jMeta 0.5
Design

Version: 0.2
Status: - Draft -

August 18, 2020

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 [MC17].

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	13000	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 form 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. Binary vs. Textual Data Formats?

The terms *binary format* and *textual format* are sometimes rather misleading and not very clear to distinguish. A textual format is clearly a format where all information, might it be parsing metadata or content, must be interpreted as text of a specified encoding. So you basically need text processing for this. In contrast, binary formats might use chunks of bytes that need to be interpreted

as integers or other numbers according to a specific byte order. However, there are also formats such as Lyrics3v2 which fully consists of textual characters, no integers in the way. Nevertheless, it has a lot of similarities with e.g. ID3v2.

So a better distinction is:

- CONTAINER-FORMATS in `jMeta` are data formats that define payload lenghts in separate parsing metadata fields; their most common representation is as binary format, but some formats such as Lyrics3v2 use text characters also for parsing metadata. Saying this, all common METADATA-FORMATS are seen as specific CONTAINER-FORMATS.
- In contrast to this, a *textual delimited data format* is a format where all bytes need to be interpreted as characters, and thus also parsing metadata is represented as strings. In addition, these formats do not have fields that specify lengths, but they use metadata delimiters to mark start end (optionally) end of a field. Examples are XML, HTML, JSON, LaTex or YAML.

3.4. 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.

3.5. 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 `DataFormats`.

3.5.1. 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.

3.5.2. 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:

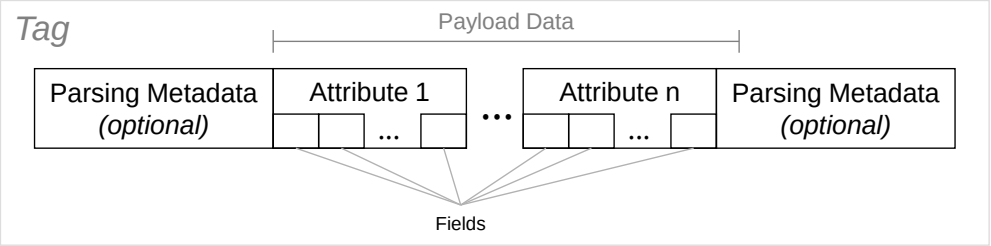

Figure 3.1.: Structure of a TAG

The most important parts of a TAG are the ATTRIBUTs.

3.5.3. 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

3.5.4. 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.

3.6. 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: Fault recognition, fault tolerance, fault correction

In addtion to "4.3 REQ 003: Fulfill specification of supported data formats", the library also has be to *fault tolerant* as much as possible. That means that specification violations and incorrect parsing metadata must be detected and - as far as possible - this should not lead to aborting parsing with an error. All violations can be protocolled and ideall automatically corrected (optional, if the user wants it).

Rationale: Old applications or other libraries write data formats sometimes not 100% compliant to the specification. In addtion, sometimes specifications are ambiguous or not accurate enough such that alternatives are possible. If there is a chance to still read the data, the user should be able to do it even in the event of faulty data.

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 encoder or decoder for multimedia content, especially not a high-performance one. 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.

4.14. EXCL 005: Scanning for data formats not supported

`jMeta` in its current version does not yet support scanning a chunk of bytes for the occurrence of a data format. This functionality might be added in later versions.

Rationale: This simplifies implementation of data format discovery for now. Furthermore, support for streaming media (where scanning is most likely to be necessary) is out-of-scope, too, see “4.10 EXCL 001: Reading media streams”.

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 [MC17].

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:

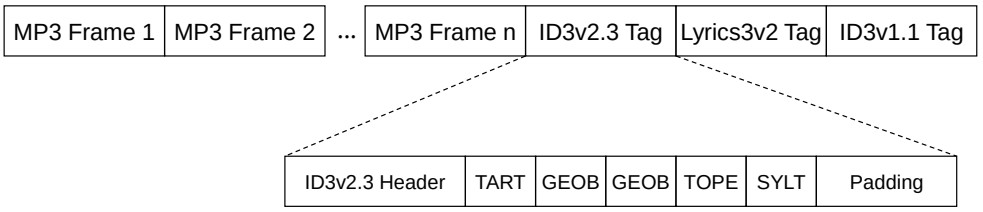

Figure 5.1.: Example 1: MP3 file with two tags

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:

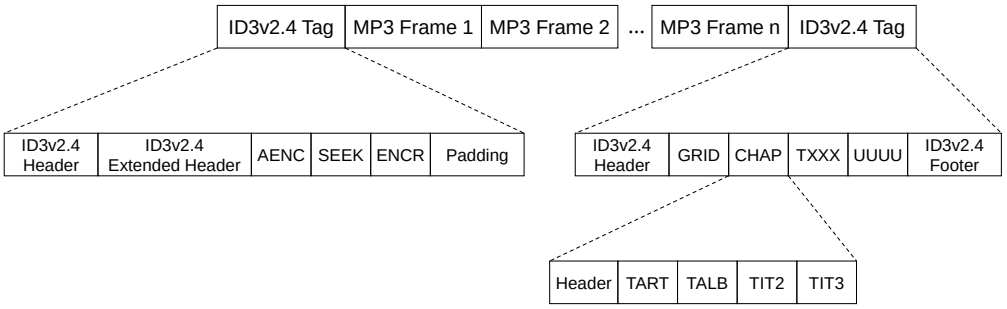

Figure 5.2.: Example 2: MP3 file with two ID3v2.4 tags

The two ID3v2.4 tags are virtually connected by a `SEEK` frame. Both have several specialties described in [MC17].

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:

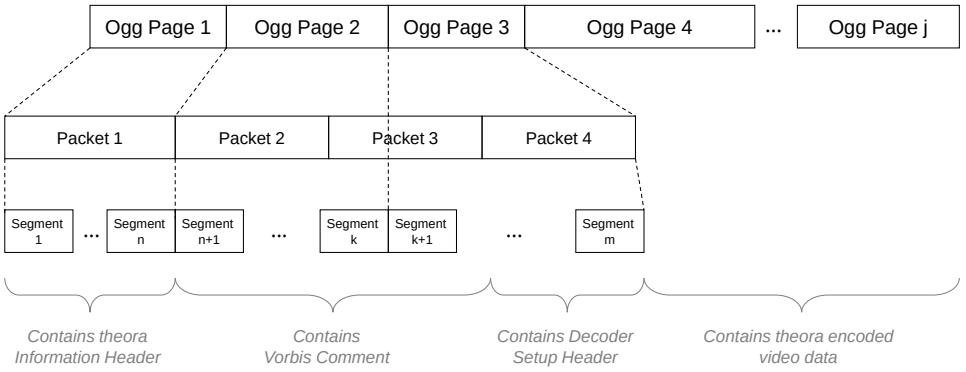

Figure 5.3.: Example 4: Ogg Bitstream with Theora and VorbisComment

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:

- The technical architecture shows the technical environment and parts of `jMeta`
- The functional architecture shows and describes the functional components of `jMeta`

6. General Design Decisions

This chapter presents all general design decisions that have a big impact on the overall architecture. It is usually very hard to revert any of those design decisions. As they generally affect the whole library, they have no direct relation to specific requirements, but build the framework to be able to fullfill those requirements.

6.1. Using Java SE 9

DES 001: **`jMeta` is based on Java SE 9**
`jMeta` is developed based on the "latest update" of Java SE 9. Migrating `jMeta` towards newer Java versions is considered on a regular basis.

Rationale: Java as programming language is well established and in wide use by a big number of world-wide developers. Furthermore, Java is platform-independent. In principle, porting `jMeta` to other platforms, to Java ME or to Java for e.g. smart phones (e.g. Android) should be less effort than porting an equivalent C/C++ library. The author has long experience with Java programming, so this should facilitate productivity. The competitor libraries in the Java environment are good, but `jMeta` can be even better. The current (state 28th of November 2017) version Java SE 9 is used, because we want to create a modern library using the newest language features, libraries and programming models in the Java world. Furthermore, it is guaranteed that we get longer support for this Java version than for the previous ones.

Disadvantages: Applications being based on Java 7, Java 8 or older are not supported by `jMeta`. This would especially effect Java EE applications where it seems not yet possible to use Java SE 9 for some application servers.

6.2. Use of 3rd Party Libraries

DES 002: **`jMeta` uses as few 3rd party libraries as possible**
`jMeta` uses mostly only the libraries offered by Java SE. In the productive code of the library, no dependencies to other 3rd party libraries are used except for logging.

Rationale: Additional dependencies could create overhead or question marks when it comes to building, shipping, testing or licensing. Furthermore, you might at one point in time reuse a well-tested and documented 3rd party library, and in the next year, it might get abandoned or bugs show up that do not get fixed. We ensure that bugs are just bugs that we can fix ourselves and we do not depend on other library developers.

Disadvantages: Sometimes this might mean "do it yourself" instead of reuse.

6.3. Component-based Library

DES 003: **`jMeta` is component-based**
`jMeta` is consisting of so-called components as defined in the follow-up sub-section. Each component has a well defined responsibility, a public API that can be used by all other components as well as a private implementation part that must not be used by other components.

Rationale: Structuring a big library into modular parts helps to manage the complexity of the library during planning, design, implementation and maintenance. By using a reasonable component structure, clear responsibilities, dependencies and decoupling of unrelated parts is achievable. Changes in implementation parts of a component do have less probability to affect other components, as they do only depend on the API of the component.

Disadvantages: Probably a little bit more overhead due to sometimes following "conventions" where an easier structure would be possible

6.3.1. Definition of the component term

A *component* in `jMeta` is *a self-contained software module with clearly defined task.* It offers *services* the can be used via a *well-defined interface (API).* These services are used both by the users of `jMeta` as well as by other components of the library.

The following figure shows a component in `jMeta` schematically:

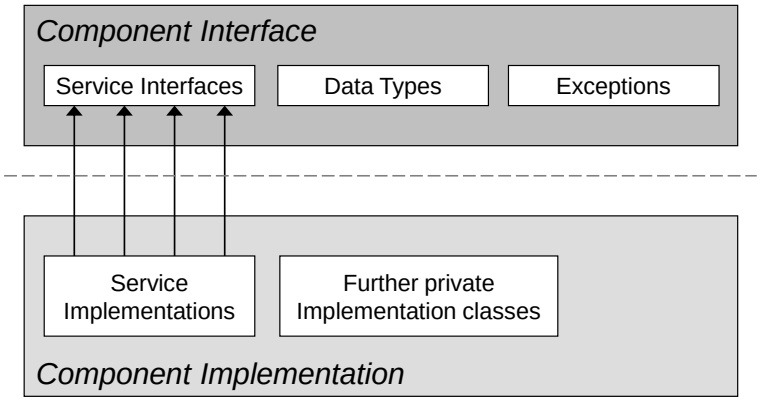

Figure 6.1.: Structure of a component

A component interface consists of one or more Java interfaces, the so-called service interfaces, exceptions as well as data types.

- *Service interfaces* offer the functional API of the component, each method corresponds to a service of the component. A single Java interface is use for a clear defined subtask of the component. Some service interface have factory methods that provide access to further service interfaces of the component.

- *Data types* are (mostly) constant Java classes that are usually used to hold data for service input and output parameters. Usually they are value types, but sometimes they are also representing a specific entity such as a medium.

- *Exceptions* are functional errors. These are mostly unchecked exceptions signaling a specific error condition, in rare cases we use unchecked exceptions if the caller can handle the situation.

In addition, a component might have data ownership for a specific kind of relevant data. This means that this data can only be read or manipulated via using the public API of this component, and by no other means.

6.3.2. Design Decision for using components

DES 004: **Components decoupled by a light-weight service locator pattern**
To ensure low coupling between components, they must know each other only via their interfaces. A component can obtain a reference to a service implementation without directly depending on the implementation class itself by using a service locator pattern, specifically the Java SE `ServiceLoader`.

Rationale: We want to decouple components. To achieve this, it must not be allowed for components to instantiate service implentations of other components directly, because service implementations are not part of the public interface of a component. A service locator exactly offers to obtain a service implementation reference without depending on it. For a library such as `jMeta`, neither JavaEE nor Spring or CDI dependency injection are valid options. Google Guice is not chosen because of DES 004. Java SE `ServiceLoader` is a very easy to use and fully sufficient facility.

Disadvantages: No disadvantages known

DES 005: **Singleton Components**
Each component offers a single *main API service interface* that itself offers access to any other service interface the component provides.
From a runtime perspective, each main API service interface thus has a single (ideally stateless) implementation, which you could refer to as "singleton".

Rationale: A main API service interfaces is purely procedural. There is just one implementation of component allowed at runtime. We just need one instance for each component at runtime, as it is stateless. This safes memory and initialization overhead.

Disadvantages: No disadvantages known

DES 006: **API and Implementation Layers**
Each component offers its services, exceptions and data types on an *API layer*. This layer is a horizontal layer cutting through the whole library and representing the public API of the library. Also per component, it is the public API of a component. Compile-time dependencies from user code to `jMeta` as well as between components are only allowed via this API layer. The implementation layer depends on the API layer and contains any private classes. An outside component and user code is not allowed to have a compile-time dependency to implementation layer classes

Rationale: Clear separation of concerns and information hiding. What implementation layer classes do is their private business, nobody from the outside world is allowed to directly work with and depend on them.

Disadvantages: No disadvantages known

DES 007: **No further subdivision of implementation layer**
It is not necessary to further subdivide the implementation layer into sublayers. Instead, each component can have an arbitrary package structure in its implementation layer code.

Rationale: We do not see any reasonable sublayers for `jMeta` that could be relevant for all components of the library.

Disadvantages: No disadvantages known

6.4. Development Environment

DES 008: **Development Environment**
As development environment, we use Maven, Eclipse and SubVersion for `jMeta`.

Rationale: Amazing toolset that leaves not much else to wish for.

Disadvantages: No disadvantages known

6.5. Multithreading

DES 009: **`jMeta` is not thread-safe**
`jMeta` is not a thread-safe library and (usually) does not use any Java APIs that are thread-safe.

Rationale: In some cases, thread-safety means performance degradation due to synchronisation points. In addition, it almost always means higher complexity. In a non-functional programming language such as Java with a lot of mutable state, it is hard to get thread-safety right. If users need to work with `jMeta` in multiple threads, they have to ensure thread-safety themselves.

Disadvantages: No disadvantages known

6.6. Architecture

DES 010: **Architecture of `jMeta`**
`jMeta`'s architecture adheres to the technical and functional architecture as descrived in chapters "7 Technical Architecture" and "8 Functional Architecture".

Rationale: See architecture discussion in detail in the next chapters

Disadvantages: See architecture discussion in detail in the next chapters

7. Technical Architecture

The technical architecture allows functional components (i.e. components offering the actual functionality of `jMeta`) to operate in their technical infrastructure. In addition, the technical architecture consists of the layers of each component. We have a component-based layered architecture as defined in "6.3 Component-based Library". The layering has already been defined in the design decisions DES 010 and DES 010. Thus, we just concentrate on a brief overview of the technical infrastructure and the technical base components here.

7.1. Technical Infrastructure

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

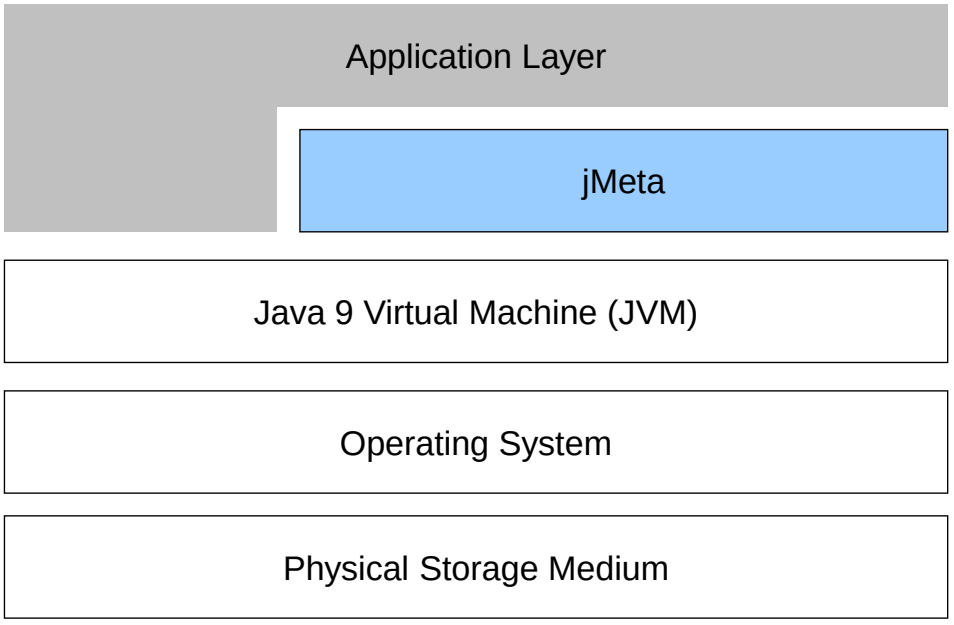

Figure 7.1.: Technical infrastructure of `jMeta`

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. Functional Architecture

The functional Architecture defines the structuring of the library into functional units, so called functional components according to "6.3 Component-based Library". For completeness, also the dependencies of the functional components with the technical base components is shown in functional architecture diagrams. In addition, the functional architecture also contains basic functional design decisions influencing the direction of the whole library.

8.1. Fundamental design decisions of `jMeta` functionality

The following design decisions are fundamental for the whole library and mostly have a cross-cutting effect. They form the basis for the functional design of all functional library components.

***DES 011:* Generic reading and writing according to a format specification**
Reading and writing different Container-Formats is done generically by a central component. A format specification describes which features a Container-Format has, especially how its Datablocks are structured and how they need to be interpreted. Thus it can be seen as a kind of generic parsing (and writing and validating) instruction of the data format.

Rationale: According to the document [MC17], Container-Formats and their more specific children Metadata-Formats have a lot of things in common, in specific the way how they are structured and parsed. These common properties can be descrubed using a generic format specification. Instead of rewriting specific parsing, writing and validation code for each new supported Container-Format, all of them can be implemented by the same generic parsing, writing and validation code that is based on the generic format specification.
This way, in the end, there should be less code, less duplication, and also a higher quality of code as it can be tested easier.
This design decision also essentially enables to meet the requirement "4.7 REQ 007: Extensibility for new Metadata and Container Formats", as this way it is much easier to extend the library with new Container-Formats.

Disadvantages: The generic code can probably not be written to support any and every possible future or existing Container-Format out of the box. Thus there might still be changes required in future. We have to carefully avoid to add format specific code to this generic core code, such that it not again becomes a complex monster, where in the end it would have been better to rewrite code for each extension. The latter risk exists if the formats do not have as much in common as assumed by this design decision. This can make testing or even understanding the generic code extensively hard.
In addition, there might be some small performance drawbacks, as generic code can of course not as heavily and specifically tuned as format-specific code can.

To mitigate the possible disadvantages of DES 011, we define the following design decision:

DES 012: **It is possible to override generic reading and writing in extensions**
`jMeta` CONTAINER-FORMAT extensions are allowed to override specific parts of reading, writing and validation code to adapt them to specifics of their data format.

Rationale: This minimizes the disadvantages of DES 012. In special cases, this can lead to easier or faster implementations. Furthermore, using such hooks, it might not be necessary to blow up the core code by specifics that are actually only relevant for one or just a few concrete formats. Those specifics can then be implemented in the corresponding format's code.

Disadvantages: It might not be easy to clearly define and communicate which parts can be overridden and which not.

To also mitigate the risk of too big and complex generic core code, we define:

DES 013: **Format specifics are only defined in extensions**
Any special features of a CONTAINER-FORMAT or METADATA-FORMAT that are no common properties/features of a lot of container formats are only defined in the corresponding extensions. That said, the generic core code never directly depends on specifics of data formats, neither syntactically (e.g. compile-time dependencs to format extension; by using if-then-else to do things different depending on the current format) or semantically (e.g. implementing a special feature for just one format, but claim it is "generic"). Only if a special feature of a format can be brought down to a common denominator for all supported formats, it is implemented generically.

Rationale: There is a strict separation between core implementation and format specifics, which allows better handling of the overall complexity.

Disadvantages: No disadvantages known

***DES 014:* High-level and low-level API**
The generic, format-independent core parts of `jMeta` form a kind of *low-level API* for parsing, writing and validating data of a CONTAINER-FORMAT quite generically down to the byte level. The end user of `jMeta` is allowed to use this part of the library directly.
However, on top of that there is a more convenient and format-specific *high-level API* the user usually uses to access data.

Rationale: According to DES 014, `jMeta` has a generic format-independent core part. This could be made private, not to be used by end users of `jMeta`. However, this would not make it possible to fullfill "4.4 REQ 004: Access to raw data via `jMeta`". At the same time, format-specific extensions need to reuse the core part. We do not want to throw the core part together with these higher-level extensions.

Disadvantages: No disadvantages known

8.2. Extensions

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 014, DES 014 and DES 014):

- A data format specification which defines DATABLOCKs and their structure and relations.
- An end-user high-level API to comfortably work with the data format, e.g. for creation of DATABLOCKs etc.; this API especially holds the data format identifier which the user can use to work with other components.
- Implementation extensions that might override generic parsing, writing and validation.

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 "4.9 REQ 009: Selective Format Choice".

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

8.3. Functional Architecture Components

The following component diagram shows the functional and technical base components of `jMeta` and their dependencies. An arrow indicates a necessary compile-time dependency. For each format extension, there is exactly one component.

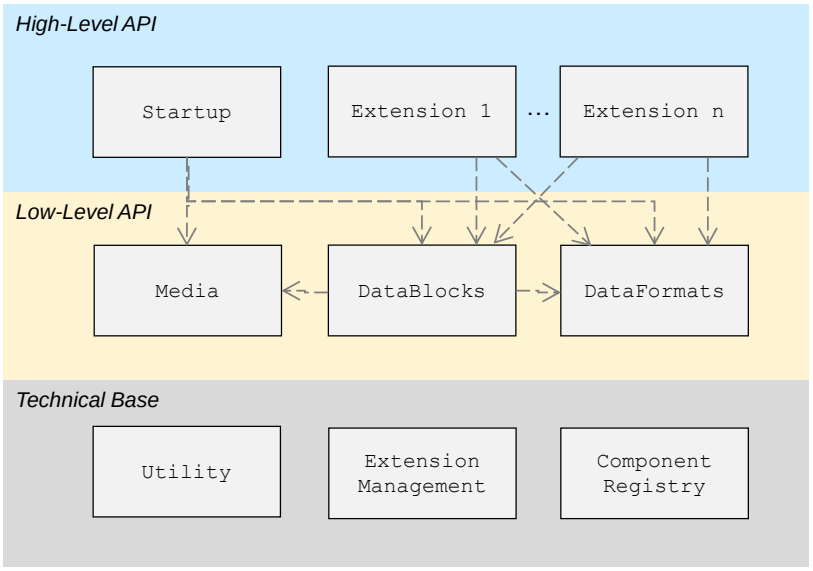

Figure 8.1.: Component diagram of `jMeta`

You can identify a kind of "component layering" (not to be confused with the architecture layering, it is just a grouping of components). At the lowest level we have the technical base components, then followed up by the low-level API components and the high-level API (according to DES 014) on top.

Dependencies to the technical base components are not shown in this diagram, as anybody can (and usually does) depend on them. Note that, due to this, we only list dependencies to technical base components in the component characteristics sections to follow if these dependencies are important and need to be specifically mentioned.

All the components displayed are just briefly described in the next sections.

8.4. Component Characteristics

All components are briefly described by a common structure that looks like this:

Component Name: The name of the component.

Task: The main responsibility of the component.

Controlled Data: Any data the component has ownership for.

Depends on <Component Name>: Occurring multiple times, once for each component it depends on with a reason why this dependency is necessary.

8.5. Component `EasyTag`

Component Name: `EasyTag`

Task: The entry point for every user to `jMeta`. It loads all extensions and provides access to all other components of the library.

Controlled Data: Keine.

Depends on `ExtensionManagement`: `EasyTag` loads all extensions by using this component.

Depends on `ComponentRegistry`: `EasyTag` uses this component to instantiate all component singletons the user asks for.

8.6. Any extension component

Component Name: Arbitrary, usually the name of the data format the extension implements

Task: Offers the high-level API to work with the data format it implements.

Controlled Data: None.

Depends on `ExtensionManagement`: It needs this dependency to the extension framework because it is an extension.

Depends on `DataFormats`: It implements a specific data format, does needs to depend on data format base classes defined in `DataFormats`.

Depends on `DataBlocks`: Extensions are allowed to override reading, writing and validation of data, thus they can depend on `DataBlocks`.

8.7. Component `DataBlocks`

Component Name: `DataBlocks`

Task: Implements generic reading and writing of Container-Formats and Metadata-Formats according to their format specification.

Controlled Data: Accessing raw Container or Tag data on byte or bit level is only allowed by using this component.

Depends on `Media`: `DataBlocks` must read and write data portions from an external medium, for this it uses `Media`.

Depends on `DataFormats`: Needs access to the data format specification by depending on `DataFormats`. This is necessary to implement the generic reading and writing.

8.8. Component `DataFormats`

Component Name: `DataFormats`

Task: Manages the generic data format definitions for Container-Formats.

Controlled Data: Has control over the data format specifications.

Depends on: No other components except technical base components.

8.9. Component `Media`

Component Name: `Media`

Task: Offers primitive to read and write all supported physical MEDIUM. Hides medium access differences for other components.

Controlled Data: Accessing data of external media is only allowed by using `Media`.

Depends on: No other components except technical base components.

8.10. Component `ExtensionManagement`

Component Name: `ExtensionManagement`

Task: Manages all extensions of `jMeta` generically, i.e. loading and verification.

Controlled Data: Descriptive data for extensions can only be obtained using `ExtensionManagement`.

Depends on: No other components except utility.

8.10.1. Component `ComponentRegistry`

Component Name: `ComponentRegistry`

Task: Implements the service locator functionality to enable components to depend only on the API of another component.

Controlled Data: Only via `ComponentRegistry`, access to metadata about components of `jMeta` is possible.

Depends on: No other components except utility.

8.10.2. Component `Utility`

Component Name: `Utility`.

Task: Offers diverse (technical) cross-functional stuff that needs to be used by most if not all other components, e.g. helper functionality to implement design-by-contract.

Controlled Data: None.

Depends on: No other components.

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

This chapter covers cross-functional aspects of the `jMeta` design which are not implemented in a single component only but cover the whole library, i.e. these are the so called *cross-cutting concerns*.

9.1. General Error Handling

Here, the general approach to error handling is presented. The basics of error handling are only slightly touched here. This section deals with errors that occur during the runtime of `jMeta`. This section does not describe concrete errors of concrete `jMeta` components. Instead, it defines guidelines used throughout `jMeta`.

9.1.1. Abnormal Events vs. Operation Errors

According to [Sied06], errors can be categorized as follows:

- *Abnormal events:* Events that should only rarely occur and require a special handling.
  - Connection to a server is lost
  - A file operation fails
  - A configuration file, table or any other data item that is expected to exist is not present
  - An external memory or the main memory lacks enough space to handle a request
- *Operation Errors:* An operation can be completed with success or error. Errors can be distinguished from abnormal events in that they are more likely or even expected to happen and that they can be handled directly by the caller of the operation.
  - Invalid user input, e.g. an input value is out of range
  - The state to call the operation is not yet established internally, the method must be called later or multiple times. This is e.g. very common in embedded programming.
  - The previous case can be seen as a precondition that must be established to call the method. However, a precondition is rather something *the user can establish* from the outside by e.g. calling another operation before.

9.1.2. Error Handling Approaches

There are several different approaches to deal with the error categories, that are sometimes even used intermixed:

- *Error codes:* In procedural system programming APIs, e.g. the Linux or Windows API, operations usually return error codes. One code is reserved with the semantics "'no error"'. Other codes are defined with special semantics that describe the kind of error.
- *Error handlers:* Some APIs allow to specify error handlers that are called back by the system in case of an error. The error handler, as the name suggests, needs to do something with the error condition. For this to work, it usually needs context information such as the kind of error that occurred.
- *Exceptions:* Mostly in object-oriented programming, exceptions are objects that usually represent errors and can contain context information about the error. They are thrown by an operation which changes the order the operations code is executed. They need to be catched somewhere in the call hierarchy of the operation, either within the operation itself or in one of its active parent callers. If they are not catched, they terminate the process the operation was running in. So throwing is the raising of the error, thereby assuming that further code in the operation is not allowed to be executed as the current state prohibits that. Catching equals handling the error. Some object-oriented programming languages like Java and C++ distinguish between checked and unchecked exceptions.

Each approach has pros and cons. In object-oriented programs, usually the exception mechanism is used.

9.1.3. Error Handling in `jMeta`

`jMeta` as an object-oriented application uses the exception error handling approach. There are no error handlers in `jMeta`. The return values of `jMeta` methods will never have the semantics of an error.

General Exception Usage Guidelines

For each special category of error, a single exception class is defined. The exception class has a meaningful name the describes the error category. The name must end with `Exception`. The class stores as much context information as necessary to be able to handle the exception. This information is passed to the constructor of the exception when throwing it. It can be requested using special getters in the exception class.

`jMeta` does not forward checked exceptions that are coming from internally used 3rd party libraries, such as the Java standard API. Instead, such errors are always wrapped by an own, meaningful `jMeta` exception.

`jMeta` cannot handle all unchecked exceptions that might occur when calling a 3rd party library. So it might happen that such an exception propagates up to the

jMeta library user. There is no means of a "'catch all"' implemented in jMeta, as there is not even a central common ground this could be implemented in.

If a jMeta exception was caused by another exception, the causing exception must be specified in the jMeta exception.

Treating Abnormal Events

In case that jMeta detects an abnormal event, it throws a jMeta unchecked runtime exception. jMeta unchecked runtime exceptions must provide information about causing exceptions and a message. The message text may further describe the exact properties of the error. By now, message texts must be given in british English.

As mentioned before, the use of jMeta might also cause 3rd party unchecked runtime exceptions that are not under control of jMeta.

How should jMeta users treat abnormal events? As the name and description of abnormal events suggest, there are usually only a few possible actions, but there might be also specialized handlings. A well-written application should not simply terminate in case of an abnormal event. It should at least have a catch all section that handles unchecked runtime errors. This section may show the error to their users in a dialog box or at least log them. In some cases, the users of that application might be able to retry the action, or choose another element on which to apply the action (e.g. in cased of a locked or read-only file). Depending on the kind of runtime error, a more sophisticated automated error handling can be set up by such an application. E.g. if a server connection had been lost, it might try to reconnect in a parallel thread, if the event happens during a database transaction, the database is of course rolled back and so on.

Treating Operation Errors

Treating operation errors is not only done with the approach that every operation error is signaled by a checked exception. Most cases where checked exceptions could be used can well be covered with the design-by-contract paradigm. Some rare cases require checked exceptions, though. This is treated in the following sections.

Design-by-Contract Design-by-contract, as originally created by Betrandt Meyer, states that each interface a client uses[1] must define a contract. The contract includes the constraints and possibilities for the user (i.e. what he must provide or which state he has to establish) as well as the duty of the interface to behave as specified. This has multiple advantages:

- The interface clearly defines how it needs to be used. It is therefore easier to understand and also easier to test. The contract is most of the time the basis for test cases of the interface.

[1] Might be a OO class or interface as well as the interface of a module etc.

- If the interface implementation directly rejects wrong input according to its precondition, it strongly supports a defensive programming style. E.g. there are no NullPointerExceptions that are thrown from just anywhere in the implemention. Instead, the user input is checked immediately and pinpointed as the source of error.

Violating the contract on the side of the user is clearly a programming error.[2] If the implementation violates the contract, it is simply not implementing its interface correctly.

Design-by-contract distinguishes:

- *Preconditions* that must be established by the user before calling a method
- *Postconditions* that the implementation must establish after a method call
- *Invariants*, i.e. state conditions for the interface that must be fulfilled before and after a method is called, but nut during a method call.

`jMeta` and Design-by-Contract :

We define:

> ***DES 015:*** **Checking preconditions**
> In the Java implementation of `jMeta`, an invalid parameter range leads to an `IllegalArgumentException`, while a precondition that can only be checked by calling a method before-hand causes a `PreconditionUnfulfilledException` if not established. Every violation of a precondition contains a british English message that describes the error using detailed argument values.
>
> **Rationale:** Fail fast is good and easy to implement
>
> **Disadvantages:** No disadvantages known

Design-by-Contract vs. Multithreading In case of multithreaded use of `jMeta`, the approach of actively checking the precondition at the beginning of each method can be outperformed. If a first thread finds that the precondition is established, but a second thread e.g. removes an item after that, leading to the precondition not being established anymore, the first thread will run into an error. The design-by-contract front-line is therefore not water-proof. However, this is not a problem of the approach itself, but a general multithreading issue. `jMeta` itself refuses to lock the code of every method for multiple threads at once to avoid decreased performance. Instead, multithreaded applications must ensure that multiple threads properly work together.

[2] In theory, with a formal approach, such programming errors could even be checked at compile-time.

Checked Exceptions Checked exceptions are only rarely used in `jMeta`, in exactly those cases where neither runtime exceptions nor preconditions are appropriate or applicable.

Whenever the caller can handle an error that is quite likely to happen, this is reflected as a checked exception. The caller catches the exception and does an appropriate handling. An example for this is an end of file condition, which may always occur and simply indicates that the client should stop reading. Another example is a class loading some data in its constructor. However, the program functions even if this data contains errors and cannot be loaded. In that case, the constructor could throw a checked exception.

9.2. Logging in `jMeta`

Logging is also used in `jMeta` as suggested by the following design decision:

DES 016: **`jMeta` maintains a log file logging the most important events and errors**
Logging is used in `jMeta` at least in `EasyTag` to protocol the startup loading of the library. In other components, logging is just used in exceptional cases, e.g. for error handling. The logging can be set on class basis by configuration done by the user, and it can also be fully disabled.

Rationale: In sufficiently complex system it is simply necessary to log status information. The user can disable and fully configure it to his needs.

Disadvantages: No disadvantages known

The question now is: What are the logging levels used?

***DES 017:* Informative output is given on INFO level, details on DEBUG and errors on ERROR level**
`jMeta` logs the following output:

- `INFO`: Every informative output related to system environment, versioning etc.
- `DEBUG`: Detailed output for progress of specific startup activities or complex operations are logged during every call.
- `ERROR`: If necessary, additional diagnostic information is logged in case of errors on ERROR level.

Rationale: There is no arbitrary but unnecessary logging. Complex background operations might require detailed log output for debugging. When errors at runtime occur within the library itself, debugging can be made easier by including addition information in logfiles.

Disadvantages: No disadvantages known

In addition, we have to define how the logging framework looks like:

***DES 018:* There is no self-made logging component, but `jMeta` uses *slf4j* wherever necessary**
Instead of a dedicated, hand-crafted and tested technical logging component which is known by every other component, and which encapsulates another logging library, `jMeta` code directly uses slf4j with a user-provided provider anywhere necessary. I.e. a direct dependency to slf4j is allowed from everywhere in the library.

Rationale: An own component wastes time and effort, in addition, every other component must somehow get an instance of it. Furthermore, innovations of the logging framework encapsulated usually remain encapsulated until you also provide it at your outside interface. slf4j is already a wrapper shielding you from details of other logging frameworks. You do not need to rewrite a logging framework today yourself, there is more precious time to spent if you do not. `jMeta` reuses the great logging providers already out there directly.

Disadvantages: No disadvantages known

The question is how to deal with technical runtime exceptions? Should they be logged, if so, where? This is clarified in the following design decision:

DES 019: **`jMeta` code logs an error message before throwing a runtime exception**
Before `jMeta` code itself throws a runtime exception, it logs a corresponding error message, preferably the exception itself, such that its message and stack trace get protocolled in the logs. This is done on ERROR level. For jMeta's own exceptions - including precondition checks - the logging is done in the exception constructor itself. A generic base class `JMetaRuntimeException` is used for this. `jMeta` code does not log anything if it throws a checked exception.

Rationale: There is no "catch all" in `jMeta` possible or reasonable, so if runtime exceptions are thrown, they are not captured in the `jMeta` logs. This is usually fine, unless `jMeta` itself throws such exceptions. We do not want to rely on the arbitrary logs of users of `jMeta`, but we want to be able to analyze what was going on in the `jMeta` code. The most important aspect of this is seeing where which exceptions originated from the library itself.
Not logging anything for checked exceptions because they are in some way expected. If they become unexpected, a JMeta runtime exception can do the logging again.

Disadvantages: Even during usual unit tests, there is logging output, i.e. probably file I/O

The last topic to handle here is whether we use tracing or not:

DES 020: **`jMeta` does not use tracing anywhere**
With tracing, we mean very fine grained log statements usually at the beginning and end of every method. This can be e.g. service facade methods only, or in the most extreme case any public, protected, default or private method called. Some applications log the input and output parameters in addition.

Rationale: Even though in rare cases tracing can give a lot of very important hints to debug a problem based on logs, it has also several drawbacks. Either you have to add it all manually. If you do, you might forget it, or you might write it in a way that is "repeating itself", making general changes harder. And, in addition, the code is cluttered with statements, probably statements such as "if logging enabled, then log". This makes the code barely readable. An alternative is to use aspect-oriented libraries. However, here it might be not always entirely clear how to only log *those specific methods* you want to trace. And there is a third-party dependency.

Disadvantages: In some cases, it might be hard to track down runtime errors if no tracing logs can be created and analyzed

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 021: **`jMeta` offers no generic configuration mechanism**
`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.

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

Disadvantages: 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 extension needs its own project and build unit. The reason is requirement "4.9 REQ 009: Selective Format Choice". This allows us to depoly individual JARs for each single extension. In addition, there is an integration testing projects for testing media that contain multiple different formats.

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 this section, the design of the component `Utility` is described. Basic task of the component is to offer generic cross-functions that are independent of metadata reading and writing, i.e. independent of the functional target of `jMeta`. As there is currently no need for detailed design for this component, nothing is described here for now.

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 021 and DES 021.

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 021, a component is a "Singleton". However, its life cycle must also be clearly:

DES 022: **`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`.

Rationale: 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).

Disadvantages: 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 023: `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.

Rationale: 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.

Disadvantages: 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 024:* 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.

Rationale: 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 "4.7 REQ 007: Extensibility for new Metadata and Container Formats". 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.

Disadvantages: No disadvantages known

How does the library identify extensions on the class path?

***DES 025:* 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.

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

Disadvantages: No disadvantages known

Regarding the configuration of extensions:

***DES 026:* 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.

Rationale: 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.

Disadvantages: No disadvantages known

Which functional interfaces do extensions provide in which way?

***DES 027:* 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.

Rationale: Although in section "8.2 Extensions", 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.

Disadvantages: No disadvantages known

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

***DES 028:* An extension is instantiated just once and then used throughout the whole application, offering a method to retrieve implementations**
According to DES 028, 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.

Rationale: 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.

Disadvantages: 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 029:* `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.

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

Disadvantages: 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 "3.6 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 030:*** **Support for random-access file access**
> `jMeta` supports the use of files as input and output medium via `Media` with access mechanism "Random Access".
>
> **Rationale:** 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.
>
> **Disadvantages:** No disadvantages known

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

***DES 031:* Support for sequential, reading byte streams**
`jMeta` supports the use of reading byte streams, i.e. `InputStream`s for input in mode "sequential access".

Rationale: `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.

Disadvantages: 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 032:* Support for random-access to byte arrays**
`jMeta` allows for random-access to byte arrays as input medium and output medium.

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

Disadvantages: No disadvantages known

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

***DES 033:* No support for writing byte streams**
`jMeta` does not support writing byte streams, i.e. `OutputStream`s.

Rationale: `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`.

Disadvantages: 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 034: **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.

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

Disadvantages: 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 035: **No locking of byte streams**
Byte streams are not locked

Rationale: 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`).

Disadvantages: 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 036: **No locking of byte arrays**
Byte arrays are not locked

Rationale: 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.

Disadvantages: 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 037:* 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.

Rationale: 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.

Disadvantages: 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 038: **`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.

Rationale: 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.

Disadvantages: 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 039: **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 of `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.

Rationale: 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!

Disadvantages: 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 040: `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*.

Rationale: 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.

Disadvantages: No disadvantages known

How is the padding requirement adressed? Like this:

***DES 041:* 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.

Rationale: 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.

Disadvantages: 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 042:* 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 042. These can be undone according to the requirements defined above.

Rationale: 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.

Disadvantages: 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 042. 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 043:* 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.

Rationale: 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.

Disadvantages: 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 043 can be achieved using a cache, as we see just a little later
- DES 043 can be implemented with a cache, i.e. anything that has been buffered should directly go into the cache for subsequent read actions
- DES 043 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 044: **`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.

Rationale:

- We provide faster repeated read access to already read data to the end-user
- This can be used for direct implementation of DES 044 in a sense of buffering when reading. i.e. the cache works as the buffer for DES 044
- Implementation of DES 044 can be done using a cache

Disadvantages: 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 044.

How to use caching to better achieve DES 044?

DES 045: **Caching is used to better mitigate the differences between `InputStreams` and files according to DES 045.**
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.

Rationale: This implements DES 045 nearly entirely, "transparent" to the user.

Disadvantages: 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 045 are heavy-weigth. If you wouldn't do anything about it to mitigate these disadvantages, then DES 045 would be nonsense, as the advantages of this approach would be dramatically overshadowed by its disadvantages.

DES 046: **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.

Rationale: 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`

Disadvantages: No disadvantages known

In addition, we require a lower limit of the cache size:

DES 047: **Minimum cache size**
Every cache has a minimum size with a sensible value, any smaller values are rejected

Rationale: Allowing caches to have sizes of e.g. 1 Byte would lead to very stupid special cases to handle or subtle bugs, and these bugs would be created by users who indeed think the cache size is given in KB instead of bytes. To avoid this, a minimum size is enforced. This allows guarantees a better average performance.

Disadvantages: No disadvantages known

What about the sizes of the individual parts kept in a cache, the *cache regions*? They should also have some size restrictions. Actually it turned out that a single configuration parameter can be used for various important things in the library, so we introduce it already here:

DES 048: **Configurable maximum read-write block size to limit data written or read at once as well as cache region size**
A parameter called **maximum read-write block size** must be configurable for the user - It is the maximum number of bytes read or written at once by e.g. flush operations or caching operations. In addition, it is also the maximum size of a single cache region.

Rationale: 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. In general, it enables a more efficient buffered reading approach. Making the cache regions exactly this maximum size is straight forward as otherwise we would need to deal with more fragmentation in average case.

Disadvantages: No disadvantages known

In DES 048 it was said that there is a minimum cache size. With the same idea, it goes that there should be a minimum for the read-write block size as well as a coupling of the two size indicators:

DES 049: **Lower bound of read-write block size, maximum cache size must be bigger than the maximum read-write block size**
Read-write block size must have a minimum value which is enforced by the API. The initially set maximum cache size set must be bigger than twice the maximum read-write block size which is also enforced.

Rationale: 1 Byte read-write block sizes would lead to awful performance and subtle special cases to handle and errors to fix. Using a sensible default and minimum scores a good average performance for every user. The other part of the design decision comes from the following observation: Buffered reading is a good idea, and a natural fit for the buffer size is the maximum read-write block size. But what if some object that needs to be part overlaps two consecutive blocks? In that case it would be possible that once you buffer the next block, the maximum cache size is exceeded and thus the previous block freed. So you end up having just the second part of the object buffered, the first one thrown out of the cache. In worst-case for streaming media this means runtime exceptions.

Disadvantages: No disadvantages known

DES 050: **Performance drawbacks of `InputStreams` are explicitly documented**
Reducing differences between `InputStreams` and files by caching in accordance to DES 050 means: You must deal with the fact that you cannot store virtually unlimited `InputStreams` in a cache. For file access, the user can also choose between `FileInputStream` and more direct access via `RandomAccessFiles`. The performance drawbacks induced by using `FileInputStream` compared to random-access - which are introduced by a unified API according to DES 050 - are explicity described in the `jMeta` documentation. The mitigation mechanisms (setting maximum cache size, disabling caching) are explicitly described with their corresponding consequences.

Rationale: 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.

Disadvantages: No disadvantages known

DES 051: **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).

Rationale: This functionality needs to be implemented and tested (additional effort). It is quite unlikely that it is actually needed when having implemented DES 051. Furthermore, when should the user free the data? A monitoring by the using code is necessary which also makes the using code more complex.

Disadvantages: No disadvantages known

After these results, the disadvantages previously listed in table 11.2 and in DES 051 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 `OutOfMemoryErrors`, we have defined DES 051 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 051. 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 051. 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 052:* 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)?

Rationale: 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 052, if a maximum size is configured. This mechanism should be sufficient for achieving a moderate heap usage.

Disadvantages: 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 053: **`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 053. However, we do not use it and implement another solution "by hand".

Rationale: 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.

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

At the end we think about a special case of caching for byte array media. One could think: This should be "disabled" as caching this anyway in-memory data would be some overhead. However, I have learnt it the hard way and we define

DES 054: **For byte array media, caching is nevertheless used**
For byte array media, caching is used the same way as for the other media types. There is no special handling for this.

Rationale: As byte arrays are already in RAM, caching is some seemingly unncessary overhead. However, it turned out during implementation that always having this "special" case was unnerving:

- Every time updating the cache when writing, one needed to check if the medium is backed by a cache or not
- When calling the cache() method, no EOM was ever thrown as caching was "disabled", no attempt to read bytes was performed, thus leading to special handling at the callers side outside of `Media`.
- A very VERY complex test case hierarchy and specific test cases where necessary to test in-memory media; this was working in the end, but for the price of a lot of "ignored" test cases (using jUnit's `Assume`) as well as a quite non-trivial test class hierarchy.

In addition, by storing the medium as ByteBuffer, one can easly create read-only views instead of duplicating the memory for the cache.

Disadvantages: A small additional overhead for caching the already cached data

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 055:* 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.

Rationale: 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.

Disadvantages: 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 056: **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.

Rationale: Necessary for implementing DES 056. 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.

Disadvantages: No disadvantages known

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

DES 057: **Explicit operation to get medium bytes**
There is an operation *getData* which returns n data bytes starting at a given offset.

Rationale: Without it there would not be any possibility to read data from a medium, as *cache* only buffers it internally without returning it.

Disadvantages: No disadvantages known

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

***DES 058: 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.

Rationale: 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.

Disadvantages: No disadvantages known

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

***DES 059: 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.

Rationale: 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.

Disadvantages: No disadvantages known

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

***DES 060:* 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.

Rationale: 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.

Disadvantages: No disadvantages known

For efficiency reasons, we define the following:

DES 061: **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.

Rationale: 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.

Disadvantages: 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 062: **`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

Rationale: 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.

Disadvantages: 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 063: **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 063, according to DES 063 aan `InputStreamMedium` and according to DES 063 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 length 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 063, 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.

Rationale: 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.

Disadvantages: No disadvantages known

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

DES 064: **If a `Medium` implementation is writable, it must also be random-access**
Every in principle writable `Medium` implementation must be random-access, too.

Rationale: 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 064.

Disadvantages: No disadvantages known

Here is a very importante note for byte array media:

DES 065: **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`.

Rationale: 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.

Disadvantages: No disadvantages known

Positions on 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 066:* 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 066 and DES 066, 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.

Rationale: 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 066).

Disadvantages: 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 067:* `jMeta` uses the interface `MediumOffset` 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 `MediumOffset` is located on the same medium behind of at the same position as this instance.
- `before`: Returns true if another `MediumOffset` is located on the same medium before the position of this instance.
- `advance`: Creates a new `MediumOffset` that is located the given number of bytes before (negative argument) or behind (positive argument) this instance. Also zero is possible as argument.

Rationale: 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.

Disadvantages: No disadvantages known

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

DES 068: **`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.

Rationale: This guarantees uniformity. However, we also want to meet the requirement "4.8 REQ 008: Read and write large data blocks". 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.

Disadvantages: 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 069: **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.

Rationale: 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.

Disadvantages: 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 `MediumOffset` 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 `MediumOffset` instances
2. Update already created `MediumOffset` instances directly for each pending change registered (see DES 069)

3. Update already created `MediumOffset` instances only when an explicit *flush* according to DES 069 occurs

Assume that `MediumOffset` instances are not updated when writing. That means the user code remembers a position of an element in form of a `MediumOffset` instance, and uses it to read or write data. If e.g. an insert operation takes place on the medium before the offset of the `MediumOffset` 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 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. The second point why we need it: If caching data, this data also refers to `MediumOffset`s. If we do not update them automatically, an additional functionality must be added to perform the updates based on the changes, increasing the complexity of the caching implementation.

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 070: **`Media` needs to automatically update `MediumOffset` instances after medium changes: `remove` and `insert`**
All `MediumOffset` 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 `MediumOffset` 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.

Rationale: `MediumOffset`s are both handed out to the user as well as used to manage cache objects. If changes to the medium happens, all `MediumOffset`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 `MediumOffset`s as invalid whenever used, or to invalidate the current cache content. Instead of doing so, we simply update all `MediumOffset`s previously created according to the kind of change, thus practically still letting them point to the same "object" as before (despite removing the medium bytes as special case). The `MediumOffset`s can thus be used further on.

Disadvantages: A central management of `MediumOffset` instances must be implemented (see DES 070).

`replace`s are discussed in the next design decision:

DES 071: **`Media` needs to automatically update `MediumOffset` instances after medium changes: `replace`**
For `replace`, the update of already created `MediumOffset`s is similar, but any offsets within the replaced region remain unchanged. In detail: Let x be the start offset of a collection of n bytes (an "object"):

- `replace` *behind the object start offset* x*:* The same as insert and remove behind the start offset, i.e. does not change it but might truncate or split an existing object starting earlier.
- `replace` *before the object start offset* x *without overlap:* For an inserting replace, The same as for `insert` before x. For a removing replace, the same as for `remove` before x without overlap. For an overwriting replace, no changes are applied to x.
- `replace` *before the object start offset* x *with overlap:* The replaced region overlaps with an existing object. Here, x remains unchanged by the replacement.

Rationale: There is no surprise for replaces behind the object start offset and those before the start offset without overlap. The only question is: If there is an overlap, i.e. x lies within the replaced region of bytes, why do we not update it? The reason is: Because of the replacement by other bytes, the object is anyway changed, the correct semantics seems to be to let the `MediumOffset`s point to x still point to x.

Disadvantages: A central management of `MediumOffset` instances must be implemented (see DES 071).

As already indicated by DES 071 the automatic updating of already created `MediumOffset` instances requires that only `Media` may create `MediumOffset` instances. These must be managed in a kind of factory to be able to automatically update them in case of writing operations.

DES 072: **`MediumOffset` instances are centrally managed by `Media` and cannot be directly created by users of the component**
The lifecycle of `MediumOffset` instances is controlled by `Media`. They are created and returned to the user via a factory method. The user has no direct access to the implementation class of `MediumOffset`.

Rationale: It is strictly required to implement DES 072. Instances that have been created by the user cannot be updated automatically, thus we have to ensure the manual creation by the user does not happen.

Disadvantages: More complex instantiation of `MediumOffset` instances.

The question *when* to update `MediumOffset` instances has still not been an-

swered yet. The following design decision clearly defines this:

DES 073: **`MediumOffset` instances are only updated after a *flush***
According to DES 073 `MediumOffset` instances are automatically updated in case of medium changes. This automatic update only happens at *flush* time.

Rationale: Assumed that `MediumOffset` 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 `MediumOffset` 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 073 requires that offsets would need to be "re-adapted" if a pending changes is undone again. Again this is additional complexity.

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

Disadvantages: No disadvantages known

This directly implies the following design decision:

DES 074: **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.

Rationale: Naturally follows from DES 074. 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.

Disadvantages: No disadvantages known

Semantics of Writing Operations

In DES 074, 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 075: 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.

Rationale:

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.

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

Before looking at the interactions between *insert*s, *remove*s and *replace*s in detail, we have to deeper think about `AMed09`. The requirement is to allow *insert*s and one of these other operations at the same offset in arbitrary order. What is the meaning behind? This is clarified by the following design decision:

***DES 076:* *insert* at the same offset as *remove* and *replace* means to insert before remove and replace, irrespective of schedule order**
No matter if you first call *insert* at offset x, then *remove* or *replace* at offset x, or the other way round: The meaning is that you want to insert new bytes before the bytes to remove or replace.

Rationale: If you want to insert something behind the replacement or removed region, the insertion offset must point to the offset of the first byte behind the replacement or removed region. Thus, calling *insert* after *remove* or *replace* cannot have the meaning "first execute the replace or remove, then insert the bytes behind the removed or replaced region". Thus, we have to ignore the sequence and ensure *insert*s at the same offsets as *remove* or *replace* are always executed first.

Disadvantages: No disadvantages known

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

***DES 077: 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, with the meaning to first insert new bytes at offset x, then removing existing bytes starting at offset x (see DES 077).
4. Step 1: *remove* at offset x, Step 2: *insert* at same offset: Both calls are allowed, with the meaning to first insert new bytes at offset x, then removing existing bytes starting at offset x (see DES 077).
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.

Rationale:

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`

Disadvantages: Then, how to implement write requirement `AMed04`? See DES 077.

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 078: 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, with the meaning to first insert new bytes at offset x, then replacing existing bytes starting at offset x (see DES 078).
4. Step 1: *replace* at offset x, Step 2: *insert* at same offset: Both calls are allowed, with the meaning to first insert new bytes at offset x, then replacing existing bytes starting at offset x (see DES 078).
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.

Rationale:

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`

Disadvantages: Then, how to implement write requirement `AMed05`? See DES 078.

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 078: 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 079: **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`.

Rationale: `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.

Disadvantages: 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 080: 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.

Rationale:

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

Disadvantages: No disadvantages known

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

***DES 081: 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

Rationale: 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`.

Disadvantages: No disadvantages known

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

***DES 082: 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.

Rationale:

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

Disadvantages: 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 083:* 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
- `MediumOffset` 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.

Rationale: According to DES 083, 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`.

Disadvantages: No disadvantages known

End medium access

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

***DES 084:* 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.

Rationale: 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.

Disadvantages: 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 `MediumOffset`, `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 085:* 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 085) is offered via interface `MediumStore` with the operations listed in table 11.3.

Rationale: 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.

Disadvantages: No disadvantages known

Operation	Description	Features and motivation random-access	Features and motivation `InputStream`
`cache(x, int n)`	Buffers n bytes according to DES 085 and DES 085 permanently in the internal cache, starting at the given offset x. 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 085 and DES 085, 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, int n)`	Reads n bytes at offset x according to DES 085, DES 085, DES 085.	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`.
`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.

Operation	**Description**	**Features and motivation random-access**	**Features and motivation `InputStream`**
`insertData(x, data)`	Implements the writing operation *insert* according to DES 085, DES 085, DES 085, DES 085, DES 085: 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, int n)`	Implements the writing operation *remove* according to DES 085, DES 085, DES 085, DES 085, DES 085: Removes n bytes at offset x. Consecutive bytes are shifted "to front", the changes first get written only with `flush()`	Removing existing metadata is a standard case in `jMeta` and must be supported.	`ReadOnlyMediumException`
`replaceData(x, int n, data)`	Implements the writing operation *replace* according to DES 085, DES 085, DES 085, DES 085, DES 085: 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`

Operation	Description	Features and motivation random-access	Features and motivation `InputStream`
`flush()`	Implements the writing operation *flush* according to DES 085: 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 085: While `insertData()`, `removeData()` and `replaceData()` only write into a temporary buffer, this call directly writes to the external medium.	`ReadOnlyMediumException`
`createMediumOffset(x)`	Creates a new `MediumOffset` instance for the given offset x according to DES 085	Is needed for random-access	Is needed for `InputStream`s
`undo(mediumAction)`	Undoes the changes of the given `MediumAction` according to DES 085, as far as it is still pending.	Is needed for random-access	`InvalidMediumActionException`
`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 085, clears the complete cache contents and other internal data structures	See DES 085	See DES 085

Operation	Description	Features and motivation random-access	Features and motivation `InputStream`
`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 085	See DES 085

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 086: **`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.

Rationale: 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.

Disadvantages: No disadvantages known

Error Handling

In general violations of the interface contract according to DES 086 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 086, DES 086, DES 086, DES 086, DES 086	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 087:* 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.

Rationale: See table

Disadvantages: No disadvantages known

11.1.3. Implementation Design

Management of `MediumOffset` instances

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

Thus, these things follow:

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

Rationale: 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 `MediumOffset` for offset x. `flush()` must keep the offset of the first insertion unchanged, as the data of this first insertion remains there. However, the offset of the second insertion must be changed, as these inserted bytes are actually located at offset $x + n_1$ after `flush()`.

Disadvantages: For many created `MediumOffset` objects, there could be a bigger memory footprint.

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

DES 089: **All instances ever created with `createMediumOffset()` 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 `MediumOffset` objects.

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

Disadvantages: No disadvantages known

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

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

Rationale: 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 `MediumOffset` 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 adding it, because the creation of a `MediumOffset` instance is usually much more commonly done than a `flush()`. Following approaches have been rejected especially:

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

Disadvantages: Finding all `MediumOffset` 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 091:* The class `MediumOffsetFactory` is used for maintaining `MediumOffset` instances**
`MediumOffsetFactory` implements the design decisions DES 091, DES 091 as well as DES 091. It offers following methods:

- `createMediumOffset()` to create `MediumOffset` instances
- `updateOffsets()` for implementation of DES 091, where a `MediumAction` instance is passed
- `getAllOffsets()` returns all maintained instances
- `getAllOffsetsInRegion()` returns all maintained instances with given offset range
- `getAllOffsetsBehindOrEqual()` returns all maintained instances with offsets bigger than the given offset
- `clear()` removes all maintained instances

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

Disadvantages: No disadvantages known

How to deal with the method `advance()` mentioned in DES 091. It creates `MediumOffset` instances. Do they have to be maintained in `MediumOffset-Factory`, too?

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

Rationale: The client code can arbitrarily use `MediumOffset.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 `MediumOffsetFactory.updateOffsets()`.

Disadvantages: Close coupling between `MediumOffset` and `MediumOffsetFactory`, as both are knowing each other.

Now we still have the issue with the memory footprint as mentioned in DES 092, as we cannot pool `MediumOffset`s referring to the same offset. Furthermore, tests so far showed that without any special measures, e.g. for only one insert (and nothing else) to process during `flush`, there are `MediumOffset`s added to the

MediumOffsetFactory. For the biggest test case having 34 different changes with a flush, it resulted in a MediumOffsetFactory with 4398 reference in case of using a cache. This is still somehow tolerable, but might not scale that good. Imagine you are editing a large file with several gigabytes, adding and removing data in each frame. This would result in probably millions of MediumOffsets being stored in the MediumOffsetFactory. Thus, there is some motivation to reduce the number of MediumOffsets maintained in a MediumOffsetFactory. What can we do? We already saw the pooling is no option. The second thing that is not possible is the following:

***DES 093:* The end MediumOffset of a MediumRegion must not be stored, but be always recalculated**
The end MediumOffset of a MediumRegion must not be stored within MediumRegion, but must always be recalculated whenever MediumRegion.calculateEndOffset() is called.

Rationale: The end MediumOffset must be managed in MediumOffsetFactory, because MediumRegion is a public class and you never know how the client uses the created end offset in his code. This would not be a reason against calculating it right at creation time of the MediumRegion.
Of course, first of all it is good style to avoid redundancy, as we anyway store start offset and size in the MediumRegion.
But the major reason for not storing it is the following case: Imagine an insert happens at an offset that is in the middle of an existing cached MediumRegion. When storing the end MediumOffset of the region, it would be shifted by the number of inserted bytes after a flush, while the start offset would not be changed. This would essentially punish the use of reduncancy: The end offset would not fit anymore to the size also stored in MediumRegion, thus leading to subtle and interesting bugs in connection with flushing changes, e.g. for the MediumRegions stored in the cache.

Disadvantages: Each call to MediumRegion.calculateEndOffset() creates a new entry to be maintained in MediumOffsetFactory

A second thing we unfortunately cannot do is the following:

DES 094: **It is not safe to create unmanaged `MediumOffsets` in internal code**
Whenever the internal implementation of scheduling or flushing requires `MediumOffset`s advanced, it might be an idea that it only uses a variant of the `MediumOffset.advance()` named `MediumOffset.advanceUnmanaged()` that does not associate the advanced `MediumOffset` instance with the `MediumOffsetFactory`.
However, this will always include the risk that some unmanaged `MediumOffset` "leaps" out to the client or somehow comes into the cache.

Rationale: It might be true that internally created `MediumOffset` for flushing and scheduling never go out to the client user of `Media`. However, still some of the `MediumOffset`s created internally using `advance` might come into the cache. We cannot afford to risk stability and reliability for things that might only occur in "corner cases". This is extremely hard to debug and puts the overall image of the library at risk.

Disadvantages: No disadvantages known

After having eliminated three possibilities for improvement already, what are the remaining options to keep the number of `MediumOffset`s maintained in `MediumOffsetFactory` to a minimum? Basically, the calls to `MediumOffset.advance()`, especially via `MediumRegion.calculateEndOffset()`, must be minimized in the implementation without sacrificing readability of the code. We define:

DES 095: `MediumRegion` **is optimized to create as few new** `MediumOffset` **as possible**
`MediumRegion` only rarely or not at all calls `MediumOffset.advance()`, thus avoiding creation of new managed `MediumOffset` instances. Specifically, a new method `calculateEndOffsetAsLong` is introduced that can be used as replacement for `calculateEndOffset` and does not create new `MediumOffset`s.

Rationale: It showed that `MediumRegion` was responsible for the greater part of `MediumOffset`s created, because it is so heavily used during flushing, offset updates and cache management. In tests we could show that for cached media, it only created six times less `MediumOffset`s using this approach.

Disadvantages: No disadvantages known

Now there is but one topic left: A special case for `MediumOffsetFactory.updateOffsets()`, as indicated in the following design decision:

DES 096: **`updateOffsets()` must not increase the `MediumOffsets` auf the `insert MediumAction` causing the shift**
According to DES 096, `insert MediumActions` shift managed `MediumOffsets` with an offset *equal to* or bigger than the insert offset by the number of inserted bytes towards higher offsets. This is necessary for *any other insert* at the same offset, but not for the causing `MediumActions` insert.

Rationale: First of all, only `MediumActions` of type `insert` are affected, as only for these also `MediumOffsets` at the same offset need to be updated.
If we would also shift the causing `MediumActions` `MediumOffset`, it would not point to the same bytes anymore. E.g. when adding the insertion bytes to the cache after updateing all offsets, we would need to recalculate the original offset again by substraction.

Disadvantages: No disadvantages known

Medium Access

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

DES 097: **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 097. 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 097, DES 097, DES 097).
- `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.

Rationale: `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.

Disadvantages: No disadvantages known

After having discussed the implementation of `MediumOffsetFactory` in the last subsection, it should be stated what kind of current positions are stored in an `MediumAccessor` implementation? This is made clear by the following design decision:

DES 098: **The current position of an `MediumAccessor` instance is not maintained in `MediumOffsetFactory`**
The current position of an `MediumAccessor` instance is not maintained in `MediumOffsetFactory`, such that it won't be changed by a `flush` of changes that would otherwise modify it.

Rationale: It might work with using a managed `MediumOffset` for storing the current position of the `MediumAccessor` implementation. This is because for stream-based media, you cannot flush, and for all random-access media the current position can be arbitrary.
However, making the current position managed would first of all confuse outside users who might expect that the current position is only modified by read and write operations of the `MediumAccessor`, and not in addition by flushing.
Second, we want to keep the memory footprint of `MediumOffset` in `MediumOffsetFactory` to a minimum. Last but not least, we want to avoid a direct dependency from `MediumAccessor` to `MediumOffsetFactory`.

Disadvantages: No disadvantages known

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 098. 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 098 and DES 098). 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 098 and DES 098). Finally, the maximum cache size can be set (see DES 098).

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 099: **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:

- `getStartOffset()`: Query start `MediumOffset` 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 `MediumOffset` 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
- `calculateEndOffset()`: Convenience method for calculating the end `MediumOffset` of the region

Rationale: 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.

Disadvantages: No disadvantages known

Maintenance of cache content is done by a helper class:

***DES 100:* 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.

Rationale: Reducing complexity of `MediumStore`

Disadvantages: 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 101:* 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.

Rationale: 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.

Disadvantages: No disadvantages known

How are the cache regions internally managed?

***DES 102:* A `TreeMap` is used for managing the cache contents, an additional `LinkedList` for freeing up according to FIFO**
A `TreeMap<MediumOffset, MediumRegion>` is used for managing the cache contents. An additional `LinkedList` is used for freeing up cache data according to FIFO.

Rationale: 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 102.

Disadvantages: 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 102. 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 103:* 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.

Rationale: 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.

Disadvantages: 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 104:* `MediumAction` has a schedule sequence number for knowing the schedule order of actions**
`MediumAction` defines a schedule sequence number (starting at 0) for distinguishing which action has been scheduled earlier. It is incremented by any scheduled action (no matter which type) by 1. The schedule sequence number is "global" for the same medium in a sense that it is simply incremented for each change, no matter what change it is and if there was a flush already or not.

Rationale: Two distinct `MediumAction` instances of type `insert` referring to the same offset cannot be sorted without this mechanism (see DES 104).

Disadvantages: 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 105:* Comparisons of `MediumActions` is done based on `MediumOffset` and - for `insert` only - 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 `MediumOffset` of `a` is smaller than the `MediumOffset` of `b`,
- Or only in the case that the `MediumOffset` belongs to a `MediumAction` of type `insert`: if `a` is equal to the `MediumOffset` 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 `MediumOffset` is bigger, or for equal `MediumOffset`s of type `insert`, if the sequence number is bigger. Furthermore it is defined: If `MediumOffset`s and sequence numbers of `insert`s are identical, then `a` is still bigger than `b` in the case that any of the other attribute of the `MediumAction`s differs.

Rationale: Sorting of `MediumAction`s should be done according to their order on the medium (i.e. their `MediumOffset`s) and only for `insert`s in creation order (i.e. their sequence number), because behaviour of consecutive operations is based on call order, according to DES 105, DES 105 and DES 105. In addition, the `MediumAction`s must be processesed in a defined order during `flush`. `equals` must return true exactly in the case that `compareTo` returns 0.

Disadvantages: No disadvantages known

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

***DES 106:* `MediumActions` are stored in a sorted data structure without duplicates**
The `MediumAction`s created before a `flush()` are held in a datastructure sorted by offset and sequence number (according to DES 106), which does not allow duplicates (e.g. `TreeSet`).

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

Disadvantages: No disadvantages known

Now regarding management of `MediumAction`s:

***DES 107:* For managing `MediumActions`, the class `MediumChangeManager` is responsible**
For managing `MediumActions`, the class `MediumChangeManager` is defined, which internally implements DES 107, with following methods:

- `scheduleInsert()` for scheduling an *insert*, implements DES 107, 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 107, 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 107, 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 107
- `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 107 and DES 107.

Rationale: 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.

Disadvantages: No disadvantages known

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

DES 108: **`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`.

Rationale: `MediumAction` needs a start `MediumOffset`, 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 108 and DES 108) and not the replacement bytes themselves. In that way, we can get a common implementation of overlap detection.

Disadvantages: 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 `MediumOffset` 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).

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 109: **`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 lead to `WRITE` actions. For all `READ` and `WRITE` actions: the number of bytes is between 0 and the (configured) maximum read-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.

Rationale: 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`.

Disadvantages: No disadvantages known

Next, we should clarify how the cache is involved in `flush()`. 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
- 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

- It is not yet clear if a specific handling of `TRUNCATE` is necessary

For the last point, we define the following:

DES 110: **`READ` operations in `flush` prefer cached data and update the cache if not already contained - We implement this using `getData`**
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.
This is exactly what getData already does - so we simply call this method. If an `EndOfMediumException` occurs during reading, this is unexpected and will lead to an `IllegalStateException` thrown.

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

Disadvantages: No disadvantages known

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

DES 111: **`INSERT` and inserting `REPLACE` operations in `flush` add their data to the cache**
Whenever `flush` processes an `INSERT` or `REPLACE` action coming from the flush plan, it adds all bytes to insert or replacement bytes from these actions to the cache.

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

Disadvantages: No disadvantages known

Very similar for `REMOVE` and `REPLACE`:

DES 112: **`REMOVE` and `REPLACE` operations in `flush` remove their removed or replaced ranges from the cache**
Whenever `flush` processes a `REMOVE` or `REPLACE` action coming from the flush plan, it removes all bytes to remove or replace from the cache.

Rationale: 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.

Disadvantages: 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 113: **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`.

Rationale: 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 `MediumOffset`s of any data cached behind the removed region.

Disadvantages: No disadvantages known

We still have the topic to update any already created `MediumOffset`s when processing `INSERT`s, `REMOVE`s and `REPLACE`s according to design decision DES 113. When do we need to perform these changes? The first part of the answer is given in the following design decision:

DES 114: `flush` **processes the flush plan in two phases: Medium access phase and cache update phase**
First of all, flush iterates all `MediumActions` in the flush plan in order and executes any `READ`, `WRITE` and `TRUNCATE` operations. So this can be called the *medium access phase*. In the second phase, all `MediumActions` are iterated again, while the cache is updated, the `MediumActions` are set to done and the `MediumOffsets` are updated in correspondence with the changes.

Rationale: Why not performing everything within the same loop, i.e. just once iterating the `MediumActions`? This would mean to e.g. first execute an `INSERT` action, and then update all references. The problem: The flush plan might still contain READ and WRITE operations which are referring the offsets on the current medium. Calling `MediumOffsetFactory.updateOffsets()` already now would also update these references, and they would be pointing to the wrong bytes! Thus, we must first process all `READ` (including adding of the read bytes to the cache), `WRITE` and `TRUNCATE` operations with their original offsets. For the caching updates in terms of `INSERT`, `REPLACE` And `REMOVE`, there are additional constraints that make it necessary to do it in the second phase. These are detailled in the next design decision.

Disadvantages: No disadvantages known

The second part of the answer when to perform `MediumOffsetFactory.updateOffsets()` comes here. Some things need to be kept in mind when updating the cache, which depends on the action type:

DES 115: **For INSERTs and the replacement bytes of a REPLACE, MediumOffsets are updated *BEFORE* the cache is updated, for REMOVEs and the replaced bytes of a REPLACE they are updated *AFTERWARDS***
For INSERTs and REPLACEs, any MediumOffsets already created and maintained in the MediumOffsetFactory are updated first, which essentially increases the offsets of any MediumOffsets behind the insertion or replacement offset by the number of bytes inserted or the number of replacement bytes. Only after this, the data newly inserted or the replacement bytes can be added to the cache.
In contrast to this, when REMOVEs and the replaced region of a REPLACE are processed, first of all, the bytes to remove or replace, if present in the cache, need to be removed from it. Only then, all MediumOffsets behind the remove or replace offset need to be decreased by the number of removed or replaced bytes (or fall back to remove or replace offset, if within the removed or replaced range).

Rationale: For INSERTs: If we add the inserted (or replacement) 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 unnecessarily and increase the complexity of cache management needed at runtime a bit.
For REMOVEs: If we would first update the MediumOffsets 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).

Disadvantages: No disadvantages known

The question for the complex `flush` operation now is: Do we need to do anything special for error handling? First of all we should think of: What could happen? In detail, there are two reasonable scenarios what could go wrong:

- There is a programming error in the library, and this leads to a runtime exception
- There is a problem during accessing the medium, may it by a HD failure or that another process accesses the medium in parallel

Both cases can be considered fatal. Would a retry make sense? Probably yes, but most often: Probably no. Should we support in performing a retry? Probably also no:

***DES 116:* No specific error handling is done in `flush` itself**
There is no try-catch block within `flush` trying to handle any runtime errors. These errors are directly thrown up to the using code. The `flush` implementation does not perform any specific measures to ensure or allow retry or "recovery" mechanisms.

Rationale: First of all, flush does not and cannot really support ACID, see DES 116. Second, the error situations one could think of are in no way expected, but refer to where unexpected, abnormal situation. In these situations, anyways the state of the overall system might be critical. We cannot foresee any possible error situations and we also cannot really pretend we can handle them correctly. The only advise for the user when an error in flush happens: If it might be a programming error, issue a bug. Otherwise, try closing and reopening the medium and redo the changes. In both cases, of course, due to DES 116, it could be that some changes have already been performed, which might leave the external medium in a corrupted state.

Disadvantages: No disadvantages known, as we hope there are really no cases where a retry would absolutely make sense

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

1. Create the flush plan according to DES 116

2. **Phase 1 - Medium access phase:** Iterate all actions of the flush plan in order, but only handle the action types mentioned below in a specific way:

   - If action = `READ`: Call `getData` at the indicated offset to read n bytes. These bytes are written in the subsequent `WRITE` action. If an `EndOfMediumException` occurs, this is considered as impossible and an `IllegalStateException` is thrown. Store the read bytes to be used by the upcoming `WRITE` action. Set the action to done afterwards.
   - 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`. Set the action to done afterwards.
   - If action = `TRUNCATE`: Execute a truncation of the medium with direct access to the medium via `MediumAccessor`. Set the action to done afterwards.

3. **Phase 2 - Cache update phase:** Iterate all actions of the flush plan in order, but only handle the action types mentioned below in a specific way:

- If action = INSERT: First call MediumChangeManager.undo on the action. If there is an existing cache region containing the insertion offset, we have to split this region now at the insertion offset. The reason is: Otherwise the already cached portion of the cached region behind the insertion offset would not be shifted by the number of insertion bytes, and the cache would get corrupted. Then call MediumOffsetFactory.updateOffsets() for the action. Finally add the insertion bytes to the cache. Here it must be noted that, as the cache add is done after updateOffsets(), the insertion offset was increased already before, and we must decrease it again to add the cached bytes at the original insertion offset!
- If action = REMOVE: First call MediumChangeManager.undo on the action. Then remove the bytes in the removal region from the cache. Finally call MediumOffsetFactory.updateOffsets() for the action.
- If action = REPLACE: First call MediumChangeManager.undo on the action. Then remove the bytes in the replaced region from the cache. Finally call MediumOffsetFactory.updateOffsets() for the action, then add the replacement bytes to the cache.

DES 117: **flush() is implemented according to the process defined above**
flush() is implemented according to the process defined above

Rationale: The flush 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 MediumOffset 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 117. Also, undo() must be executed first, otherwise updateOffsets() changes offsets which would lead to a serious problem in a TreeSet as used in MediumChangeManager to find the correct reference with contains.
The cache update is divided into two parts as mentioned in previous design decisions.
The cache management is completely done within MediumCache.addRegion(), such that maximum size and consecutive regions can be optimized there.

Disadvantages: No disadvantages known

***DES 118:* Flushing progress is logged on DEBUG level**
Flushing and probably even scheduling of changes is logged on DEBUG level

Rationale: The process of flushing is the most complex process in `Media`. If there are any subtle bugs for end-users, they might be incredibly hard to track down without proper logging.

Disadvantages: 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 [CFPPaper].

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 118
- 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 read-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 read-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 read-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 read-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 read-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 read-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 `removess`, `replaces` and `inserts`	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 and he can also change the maximum read-write block size. But how? In DES 118, 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 119:* 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.

Rationale: 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.

Disadvantages: 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 120: **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.

Rationale: 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.

Disadvantages: No disadvantages known

Medium	Parameter Name	Type	Default Value	Description
`File`, `InputStream`	`maxCacheSize`	long > 65535	1 MB	Sets the maximum cache size according to DES 120. 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`, `InputStream`, `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 120.

Table 11.6.: Configuration parameters of medium access

11.2. DataFormats Design

In this section, the design of the component DataFormats is described. Basic task of the component is to describe the structure of a container or metadata format in a way that is useful for parsing them generically.

11.2.1. Format Comparison

Before any real design decisions can be found, we have to look what the formats are that we try to generically describe with the DataFormats component. The main source of this whole chapter is [MC17].

We fully focus on the following data formats here, ignoring any other possible formats as out of scope:

- Multimedia container formats: Ogg, QuickTime, MPEG Layer I Elementary Stream (for MP3), Matroska, RIFF
- Metadata container formats: ID3v1 and ID3v1.1, ID3v2.2, ID3v2.3, ID3v2.4, VorbisComment, Lyrics3v1, Lyrics3v1, APEv1 and APEv2

As first introduction, we have to compare those formats according to their most important structural aspects. These aspects are as follows:

- Basic encoding they use: **Byte orders** and **character encodings** supported
- Building blocks: What are the **containers** in this format? Are there other sub-structures? Can they be embedded in each other? Is there a hierarchical structure with arbitrary nesting? For metadata container formats, we call the elements actually holding the metadata *attributes*, which is nothing else than a special case of a container.
- Headers and Footers: Despite the payload, are there **headers** and/or **footers** in the containers or attributes? Are they fixed or variable size?
- Lengths of containers: How is the **length of containers** or of their payload defined?
- Lengths of attributes: How is the **length of attributes** for metadata container formats defined?
- Padding: Does the format have some special way of supporting **padding**?

The following table provides all these properties for multimedia container formats, skipping attribute topics:

TODO Table for multimedia formats tab:DFcompareMult

The following table compares these properties for the metadata container formats considered:

TODO Table for metadata formats tab:DFcompareMeta

Note that ID3v2x is a very special format in multiple dimensions: It is incredibly overloaded with a lot of features you might never dare dream to use, say: over-designed. It offers e.g. support for encryption, grouping, compression of tags, embedded lyrics or audio guide codec data which basically contain spoken versions of the metadata stored and other things. On top of that, it has a strange conversion scheme called the *unsynchronization scheme*.

11.2.2. The Container Metamodel

In order to give a feasible model of the structure of a container format (which includes metadata container formats), a metamodel of the typical data structure of a container format is developed here.

The metamodel is ultimately shown in the following figure, the corresponding design decisions will follow:

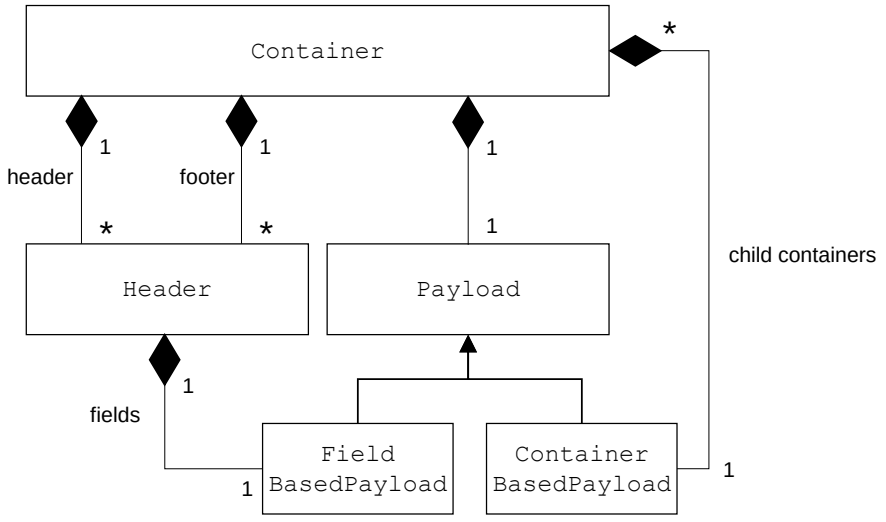

Figure 11.1.: The container metamodel

***DES 121:* Containers as basic top-level unit, containing nested containers**
All data that can be processed by `jMeta` consists of containers. A container is a sequence of bytes with a specific structure belonging to exactly one container data format. However, consecutive containers might belong to different data formats.
In addition, a container - as the name suggests - contains other data. This data might be containers of the same format again. The nesting might be arbitrarily deep. A nested container is able to contain data of other formats.

Rationale: Every format considered here has such basic building blocks, called differently in each format specification (atom, frame, page, element, item, tag etc.). The same file or media stream might consist of containers belonging to different formats. One example is an MP3 file with an ID3v2 tag. See also [MC17], part IV for more details.
Most of the supported data formats have some notion of nesting child containers (or, for metadata formats: attributes) within parent containers. Those child containers have either exactly the same stucture as the top-level containers, or they differ slightly. Some formats such as RIFF explicitly define specific containers that embed data belonging to other metadata formats.

Disadvantages: No disadvantages known

How are these containers structured in detail?

***DES 122:* A container consists of an optional header, payload and an optional footer**
Every container - no matter which data format - follows the same basic structure: One or more optional headers start the container's byte sequence. Following this, there is a sequence of bytes called the payload. At the end of the container, there are one or more optional footers. A container must either have at least a single header or at least a single footer.

Rationale: Considering table **??** and **??**, every container in every format has at least one header or one footer, including nested containers. Only ID3v23 as sole exception has a second, so-called extended header. Thus we have to support multiple headers and footers, even though this might be a singular case. But future formats might come to bring multiple headers, too. Modelling the middle part of a container as payload explicitly instead of just as list of sub-containers or fields makes sense because it allows to lazily read substructures, or entirely skip the payload.

Disadvantages: No disadvantages known

We saw that containers have a structure, and containers can live within other containers. But on the lowest level, there are leaf nodes, the fields:

***DES 123:* Fields as the atomic unit of all byte sequences**
Fields are the leaf nodes of the container metamodel: A field is a sequence of 1 to N consecutive bytes. A field has a **binary value** as well as an **interpreted value**, i.e. a human-understandable value with a specific meaning. We say that a field has a specific type, which basically describes the mapping between binary and interpreted value as well as the allowed values or format of a field.

Rationale: The term "field" is quite commonly used in binary data formats. It does not make sense to go down to individual byte or bit level. The smalles level of semantic in every binary or textual format is a field.

Disadvantages: No disadvantages known

Now towards headers and footers: Are they inherently different or basically the same? Fixed or variable size?

***DES 124:* Headers and footers as sequences of fields are structurally identical**
Headers and footers are sequences of fields, i.e. they must not nest a container. They are either variable or fixed-size. For both header and footer, the same model class called `Header` is used.

Rationale: No data format specifies a header or footer nesting a container-like data structure, they all consist of plain fields. Despite the location (before or behind the payload), there is nothing really distinguishing headers or footers from a structural perspective. Thus the same model class can be used for both. However, there is no english term for that class that seems to clearly be a grouping term for headers and footers.

Disadvantages: No disadvantages known

So far, we just introduced containers, fields, headers, footers and payload. We learned that containers might embed other containers, while headers and footers must only consist of fields. But what about the structure of the payload of a container? Of course, if the headers and footers cannot embed child containers, they must hide within the payload. Is this the only possibility? Of course not:

***DES 125:* Payload either solely consists of containers or fields**
The payload of a container either fully consists of a sequence of child containers or of a sequence of fields, but never both.

Rationale: Ultimately, we must boil payload down of fields, or put otherwise: We cannot recurse indefinitely into sub-containers. So at last, there must be a final sequence of child containers having payloads solely consisting of fields. So it is clear that we have both cases. But what about the mixed case: Should we have containers in the model that have both child fields neighbouring child containers? In rare cases such as - again - ID3v2, it would be possible to think of having the frames as child containers of the tag, followed by the padding child field of the tag behind it. However, we avoid such complexities, but payload is either ContainerBasedPayload or FieldBasedPayload. See next design decision of how we handle the ID3v2 special case.

Disadvantages: No disadvantages known

How to handle padding? If we look at the data format comparisons in **??** and **??**, we see that padding is just a special case in ID3v2 and MP3. All other formats solve it "more naturally" by embedding it in special child container. ID3v2 just adds some nullbytes at the end of the tag payload behind the frames. How to model this? There are basically three possibilities:

- The padding is a part of the payload, basically a field. This seems most intuitive, but is already excluded by the previous design decision, because then we would need to support mixed payloads, consisting both of child containers and fields.

- The padding is part of the payload, but considered as a very special child container.

- The padding is an optional part of the tag footer.

Based on these alternatives, we can really only decide the following:

***DES 126:* Padding in ID3v2 is modeled as very special child container of the tag**
Padding in ID3v2 is modeled as very special child container of the tag, where the header is the first null byte, and all follow-up bytes form the payload. Note that for this to work, `jMeta` must support containers with no (i.e. empty) payload for the very special case that the padding only consists of one byte: The header magic key identifying the padding.

Rationale: Modeling it as child field of the payload is not possible as it would contradict DES 126. Modeling as part of the footer would be odd, as not every version of ID3v2 defines a footer at all, and we would have a variable sized footer optionally starting with an arbitrary number of nullbytes, which would not be very intuitive from a library users perspective, too.

Disadvantages: The only disadvantage possibly to be identified that there could be padding just consisting of a single byte. This would mean we'd have a container with just a header, but an empty payload, which actually is quite degenerate.

The overall term for containers, headers, footers, fields and payload is a **datablock**. This term is used further on throughout this document.

There is another ID3v2 specific that we directly want to exclude here:

***DES 127:* ID3v2 transformations such as unsynchronization, compression, encryption are not handled generically**
ID3v2 is clearly overdesigned. It includes an outdated mechanism called *unsynchronization* to ensure players do not interpret ID3v2 tags as MP3 frames and try to play them. ID3v2 also defines frame-based compression and encryption possibilities.
These facilities are not handled generically in `jMeta`, they are just specifically implemented in the ID3v2 extensions.

Rationale: Unsynchronization is not needed anymore, as current players all know about ID3v2 tags. Even if it would be needed: There is no other data format know to date that defines any such transformation schemes, not even for compression or encryption.
Thus, declaring this all as generic concept is useless as only one format defines such things, and these things are even only very rarely used.

Disadvantages: No disadvantages known

11.2.3. Representing a Data Format

Data Formats need some kind of representation:

DES 128: **Class for data formats and especially for container formats**
There is a base class called `AbstractDataFormat` representing a general data format, which has a subclass called `ContainerDataFormat` that is used for all data formats listed in "11.2.1 Format Comparison". Each individual format is represented as an instance of `ContainerDataFormat`. These two classes are neither enums nor "dynamic enums", but just plain old java classes.

Rationale: We need some way to identify which data format a container belongs to. A class allows to store additional format-wide properties for logging or even with a functional notion behind. Why not using an enum to identify each supported data format? Because enums are static, and we do not want to change the enum whenever a new format comes around. And it is not clear which formats are out there by 3rd party. A dynamic enum pattern is unnecessarily complexity here, so just a plain class. Why a just an instance of `ContainerDataFormat` rather than a new subclass for each format supported? First of all, data formats must be used as tokens by end-users of the library to refer to which tags or containers they want to read. Creating a new sub-class instance everytime they want to have a token is strange. Instead, they can use a public constant instance as token. The next question: Why at all a `ContainerDataFormat` subclass? The reason is that `jMeta` might want to support even more different formats in future, e.g. XML. This can be a sibling subclass of `AbstractDataFormat`. From now on, everything described in this section is only covering container data formats.

Disadvantages: No disadvantages known

But that's not all of it. If we structure data belonging to a format into containers, headers, footers and so on according to the generic metamodel, *how exactly* is a correct datablock of a given format structured? Of course, each format individually specifies the exact structure of its containers (how many headers, footers etc., what the payload can contain), headers (which fields make up a header), fields (which type a field has, how long it is and which values it can have) and so on. Thus, we need a place where we describe the structure of all datablocks belonging to a data format in more detail, otherwise we could not parse this data generically.

We define:

DES 129: `DataFormatSpecification` **provides** `DataBlockDescriptions`**,** `DataFormatRepository` **maintains all** `DataFormatSpecifications`
The exact properties of a datablock is described by an instance of the class `DataBlockDescription`. Which properties are needed in detail is determined later in "11.2.4 `DataBlockDescription` Common Properties". The interface `DataFormatSpecification` provides access to all `DataBlockDescription`s offered by one specific data format. Last but not least there is a `DataFormatRepository` interface which basically only maintains a list of all supported data formats and their corresponding `DataFormatSpecification`s. A concrete `DataFormatSpecification` is defined in an extension and loaded when starting `jMeta`.

Rationale: For generically parsing data blocks according to a format definition, the parser code needs information about the exact structure of a data block. Thus, we need a place to maintain all data formats, their specifications and a description of each possible individual container, field etc.

Disadvantages: No disadvantages known

11.2.4. `DataBlockDescription` Common Properties

First of all, we need a clear way to identify each specific datablock that might occur in the data format. Every container type, header, footer, field and payload needs an identifier:

***DES 130:* Each datablock of a format has a specification id unique across all supported formats**
Each distinctly defined container, header, footer, field or payload block of a data format is assigned a unique specification id. Using this specification id, a parser implementation can lookup the `DataBlockDescription` of the block from its associated `DataFormatSpecification`. Specification ids are human-readble rather than generated GUIDs. The uniqueness is guaranteed by using a top-level prefix identifier that is directly checked to be unique upon all loaded data formats when loading the extension defining the data format. The ids reflect the hierarchical nature of a data format according to the container meta model, following these rules:

- Each id consists of one or more *segments* separated by a dot.
- A segment name does only contain ASCII alphanumeric characters, starting always with a lower-case letter and preferring lower-case only letters, i.e. no special characters, no blanks etc. So a datablock id has much in common with a Java package name.
- The first segment is the data format top-level prefix which equals the (single) top-level container id of the data format. So we have the implicit decision that each format starts off with a single container on top-level enclosing everything else. This is true for all data formats considered here.
- Each container consists of an arbitrary number of headers and footers as well as exactly one payload. The segment name of a container payload is always `payload`. If there is just a single header or footer, it is called header or footer respectively. Multiple headers or footers bear more specific, distinct names.
- Child fields of a header or footer have descriptive names and so on

Rationale: When parsing, the format structure of a datablock must be clearly visible and available. By an id, this format structure is clearly adressable and the corresponding `DataBlockDescription` can be obtained from a `DataFormatSpecification`. The uniquenss of the id guarantees that the parsing logic cannot unintentially try to parse a block of bytes according to the incorrect format.
The reason for not using generated unique ids such as counters or GUIDs is that we want to have memorable and recognizable names, both for library developers (e.g. during debugging) or `jMeta` end-users that need to work with datablock ids directly.

Disadvantages: No disadvantages known

Besides their id, what else might be desirable on first glance when looking at a datablock taken from a data format's specification? Some descriptive stuff comes

to mind. Second, it seems like every datablock must have exactly one type that is useful to store. So we can state:

DES 131: **Basic properties held in a `DataBlockDescription`**
Each `DataBlockDescription` instance holds the datablock id of the datablock according to DES 131. In addition, it has:

- A more human-readable name of the datablock, if available; only for display or documentation purposes
- A description of the datablock, probably taken from the format specification, preferably in English. This is meant to give more information about the purpose of the datablock, but has no other meaning.
- The `DataBlockType` of the datablock, which is: `HEADER`, `FOOTER`, `CONTAINER`, `FIELD`, as well as - according to DES 131 - `FIELD_BASED_PAYLOAD` or `CONTAINER_BASED_PAYLOAD`.

Rationale: These properties are clear from the beginning, the concrete is known statically, and it is needed to take decisions during generic parsing.

Disadvantages: No disadvantages known

We defined that datablock ids are organized hierarchically as their real blocks are organized hierarchically. However, still, it is not sufficient to somehow maintain the "hierarchy" indicated by the metamodel only in the datablock ids. Instead, we define:

DES 132: `DataBlockDescriptions` **contain a list of all child ids they might contain, where order matters**
A `DataBlockDescription` instance contains a list of all child datablock ids that form the structure of the datablock described. While headers, footers, containers and payload usually has child ids, `DataBlockDescriptions` of fields must not define child datablock ids. The list of child ids is ordered with the following meaning:

- For containers, the child id list exactly matches the correct order of headers, payload and footers that build the container
- For headers, footers and field-based payload, the child id list exactly matches the order of child fields within the header, footer or field-based
- For container-based payloads, the child id list order does not have a meaning, i.e. if multiple different container types are allowed, they might appear in any order

Rationale: The parsing code somehow needs information about what child blocks of which format to expect within a given datablock, otherwise parsing it correctly is simply impossible. Each data format clearly defines the order of fields, headers and footers. Usually, most container formats define just exactly one child container that can appear in a container-based payload. However, there are cases such as QuickTime (atom vs. QT atom container) that define multiple different container types that might appear in any possible order.

Disadvantages: No disadvantages known

We first have to define some limitations for these datablock ids that should be made clear:

DES 133: **Datablock ids and their segments MUST not be used for any logic**
Especially, no assumptions about the hierarchical relations of distinct blocks must be done based on their id. E.g. a block with specification id `id3v23.payload.talb.encoding` not necessarily is a child of a block with id `id3v23.payload.talb`, as there could be multiple `talb` containers within an ID3v23 tag. So datablock ids are only specification ids, not instance ids! Even within the same specification, a case might occur where the same chunk of data with the exact structure appears in several places, e.g. as child of the top-level container, and also as deeper nested child. Moreover, some container formats encourage to embed datablocks belonging to even completely different data formats. How do we handle such cases? In no special way. Even if a container might appear on several levels, we just define its structure once in the data format with just one id. And if a datablock of a data format is embedded in another container of a different format, it still keeps its well-defined datablock id according to its data format. So this again shows that based on the id, we can never determine the real hierarchy level or instance relations of two datablocks during parsing!

Rationale: Quite clear, otherwise we would implement a lot of bullshit code

Disadvantages: No disadvantages known

11.2.5. Generic and Concrete Containers

Data formats define different kinds of *container types*. A container type for us is defined as a data structure having a fixed number of headers and footers, where each header and footer has a strictly defined sequence of fields with strictly defined syntax and semantics, yet leaving undefined how the payload looks like. They just state that there is a payload consisting of bytes, but not how it is structured. Most often, there is just one header or just one footer. Here are some examples of container types:

- Most container formats such as RIFF or Matroska define exactly one container type. In these formats, those containers usually can be found on top-level as well as arbitrarily nested within parent containers
- Some formats that are no container formats in the narrower sense[1] such as ID3v23 use a slightly different container type for the top-level (the tag) container as for the child (the frame) containers
- Some container formats such as QuickTime even define multiple container types that can appear on the same level: First of all, there are the top-level containers called atoms; one extension of atoms are QT atoms, which just use a slightly extended header compared to the usual atoms. Both

[1] In terms of their main purpose is to contain *arbitrary* multimedia data, not only data of a specific kind.

might appear on top-level. Furthermore, as child container types, not only atoms or QT atoms might appear, but also QT atom containers (a slightly confusing term here, but this is how QuickTime names them)

Container types are the pillars of the most basic property of most well-designed container formats: They define the structures of their basic buildings blocks (i.e. their container types) just once, and specialize this basic structure whenever needed. This at first allows to write generic parsers that may skip containers whose semantics and purpose they might not be aware of, but yet they still know their structure and thus are able to determine where the next container actually starts. This allows to extend data formats by new subtypes whereas existing parsers still work.

But this is not yet enough. Most container formats go much farther than only define a generic container type that can be parsed and defines a structure. They specialize in terms of defining specific containers having a specific meaning (data content) and whose payload follows a deeper defined format. These specialized containers can be said to have the same structure as a given container type, but they fill this container type with meaning and the payload with additional structure.

We define:

DES 134: **`DataFormats` distinguishes generic and concrete containers**
`DataFormats` distinguishes generic and concrete containers:

- A container is called a **generic container** if it is just known to follow a container type's structure, but a more specialized container type is not (yet) known; more technically speaking: A generic container is a container whose detailed payload structure is not yet known in advance, but can only be determined at runtime, usually by reading some specific `id` field of the container's header or footer. Examples for generic containers are: Generic ID3v23 frame, generic (QT) atom, generic Matroska segment, generic APEv2 item and others. We can say that generic containers form the archetypes for the even more useful concrete containers. `DataFormats` must maintain `DataBlockDescription`s for generic containers.
- A **concrete container** is a container with a well-defined payload structure. It might be based on a generic container which means it entirely inherits its header and footer structure, while it might "override" specific field values in headers and footers. Examples for concrete containers are: ID3v23 tag (because its payload structure is already statically clearly defined), ID3v23 TALB frame, ID3v23 GEOB frame, moov QT atom, APEv2 title item and others. `DataFormats` must maintain `DataBlockDescription`s for concrete containers.

Of course, there are much more concrete containers than there are generic containers.

Rationale: Generic containers are necessary such that `jMeta` can basically parse data format content, yet not necessarily understanding its payload. These "unknown" containers can safely be skipped, and `jMeta` can later be extended to also understand these concrete types.
Concrete containers are necessary such that `jMeta` can read and write data with specific semantics and inner structure. Otherwise it would not be possible to write album, track, artist etc. fields to metadata container formats and so on.

Disadvantages: No disadvantages known

How exactly are these generic and concrete container `DataBlockDescription`s defined in a `DataFormatSpecification` and used at runtime?

DES 135: **DataFormatSpecifications list generic container as children, parsers resolve them to concrete containers at runtime**
Generic container types might occur on arbitrary levels of the data block hierarchy, either on top-level or as child datablocks of a CONTAINER_BASED_PAYLOAD datablock. The semantics is: The file/media stream or payload contains a sequence of containers having a specific basic structure (i.e. they have a specific container type). This type can be found to be even more specific at runtime. For this, container formats define a special id field either in the container header or footer. This id field identifies the concrete type of container. After the parser has read the id field, it can lookup if there is a more specific DataBlockDescription for this id defined in its DataFormatSpecification. If so, the generic container is changed to the concrete container, thus the payload can now be parsed differently.

Rationale: All container formats define their CONTAINER_BASED_PAYLOAD to contain a multitude of different concrete containers that all follow (usually) a single generic container type.

Disadvantages: No disadvantages known

The actual detailed parsing process is described within the design of DataBlocks.

Now we already mentioned in DES 135 that there is "usually" only a single generic container defined within a CONTAINER_BASED_PAYLOAD. However, there are also already two exceptions we can list here:

- A (QT) atom generic container having a CONTAINER_BASED_PAYLOAD could have two different generic container types as child containers: Either (QT) atoms again, or QT atom containers that have a totally different header.
- We already "artificially" introduced a second case with DES 135: An ID3v23 tag CONTAINER_BASED_PAYLOAD either contains a generic ID3v23 frame or a padding container.

If a CONTAINER_BASED_PAYLOAD might contain different types of containers in arbitrary order, how is a parser able to tell which generic container type he is handling currently? This is all based on magic keys and described in "11.2.6 Data Format Identification and Magic Keys".

11.2.6. Data Format Identification and Magic Keys

This section deals with the question how to identify that a given chunk of bytes belongs to a specific data format. This is not an easy thing because of the following aspects:

- There are different media types such as files, continuous streams and in-memory bytes

- Different strategies of reading data formats are (partly) supported in `jMeta`: Most data formats are designed for the default way of reading, i.e. **forward reading** starting from the beginning of the medium from offset zero towards higher offsets. However, sometimes **backward reading** is much more performant, especially for multimedia tags. Last but not least, you can think of **scanning** to search a chunk of bytes for the first occurrence of a known format.

- Some data formats are extremely similar to each other, either due to the fact that they are different revisions of the same family of formats (e.g. ID3v23 and ID3v24) or due to decisions of standardization boards (e.g. the ISO base multimedia format with its children JPEG 2000, MPEG-4 and others which are nearly identical to QuickTime) - how to distinguish them.

- A few data formats such as MP3 allow coexistence of multiple unrelated data formats on top-level of a medium, usually a file. An MP3 file not only contains MP3 frames, but also ID3v1, ID3v3, LyricsV3 or APE tags, all of which are entirely different data formats specified on their own. This is a special case as most other multimedia data formats such as RIFF, QuickTime, Matroska and Ogg require to only have data blocks of RIFF, QuickTime, Matroska and Ogg on top-level in the same medium.

What are the different possibilities of identifying the data format of a medium?

- *Based on the file extension:* In 80 percent of the cases, you could think of identifying a data format based on its file extension. This requires a mapping of extensions to data formats. While this could be a feasible approach, its downsides are clear: If the file has no extension (e.g. on Unix systems this could be the case), you cannot identify it. Furthermore, this approach is not feasible for streaming media and in-memory bytes.

- *Using other external hints:* You can use other external hints, e.g. external metadata (provided in separate files, in an HTTP header for streams) or let the user select or hint at the data format. Obviously, this could never be a possibility covering all cases and requires a lot of very diverse strategies to implement.

- *Based on information in the data:* You read the bytes and try to identify the data format based on the bytes in the data itself. Mostly, this translates into using magic keys which we will cover immediately or scanning. While this might seem strenuous, it is actually the only proven way to ensure proper identification that even works for all different media.

So we decide:

***DES 136:* Default identification mechanism uses magic keys**
`jMeta` uses magic keys to identify the data format of a given chunk of data bytes. Scanning is not used for identifaction for now.

Rationale: Its the only reliable way to identify a data format, and we anyway need to parse the data. It works for all media types. Regarding scanning: It is currently excluded, see "4.14 EXCL 005: Scanning for data formats not supported".

Disadvantages: No disadvantages known

Now we want to deal with the very basic and central concept common to most if not all of those data formats: **Magic Keys**. A magic key is a sequence of bytes with a fixed, well-known value, which allows to identify a block of bytes as belonging to a data format, enabling generic parsers to understand how to parse a byte sequence. Thus, it is usually located at the start of the medium. For example, an Ogg page is identified by the magic key string "OggS", encoded as ASCII. Of course, this implies that different data formats use different magic keys.

These are the main properties of magic keys we are most interested in:

- A magic key has a single fixed byte identification sequence that is usually an ASCII encoded string
- It is either starting at the beginning of a container, or at a well-defined fixed offset from the start of the container, thereby being a field in the container's header or footer
- That said, also footers have magic keys to be used for backward reading, e.g. for ID3v23, APEv2 or Lyrics3v2
- Data formats might have multiple magic keys identifying them - one example is RIFF where the magic key also identifies the byte order of the container

Let's list all magic keys of the data formats we care about in table 11.7.

Data Format	Header Keys	Footer Keys
Ogg	"OggS" at start of page header	-
Matroska	1A45DFA3h	-
RIFF	"RIFF" (big endian) or "RIFX" (little endian) at start of chunk header	-

Data Format	Header Keys	Footer Keys
AIFF	"FORM" at start of chunk header	-
QuickTime	"ftyp" or "moov" or "pnot" or "mdat" or "skip" as atom type (starting with 5th byte) in atom header	-
MP3	Eleven 1 bits at start of frame header	-
ID3v1 and ID3v1.1	"TAG" at start of tag header	-
ID3v22	"ID3" followed by version number 0200h at start of tag header	-
ID3v23	"ID3" followed by version number 0300h at start of tag header	-
ID3v24	"ID3" followed by version number 0400h at start of tag header	"3DI" followed by version number 0400h at start of tag footer
APEv1	"APETAGEX" followed by version number 1000 at start of tag header	"APETAGEX" followed by version number 1000 at start of tag footer
APEv2	"APETAGEX" followed by version number 2000 at start of tag header	"APETAGEX" followed by version number 2000 at start of tag footer
Lyrics3v1	"LYRICSBEGIN" at start of tag header	"LYRICSEND" starting with 7th byte of tag footer
Lyrics3v2	"LYRICSBEGIN" at start of tag header	"LYRICS200" starting with 7th byte of tag footer
VorbisComment	*No magic key*	-

Data Format	Header Keys	Footer Keys

Table 11.7.: Magic keys of all relevant data formats

Let us look at the odd exceptions in this list:

(1) We see that the only format not having a magic key is the VorbisComment

(2) MP3 as the only format does not have a string magic key, but rather a bit sequence

(3) RIFF as the only format defines multiple different magic keys in its header

(4) Lyrics3v2 and ID3v24 are the only formats where header and footer magic keys differ

(5) QuickTime and Lyrics3v2 are the only formats where the magic key does not start at the beginning of its header/footer, but at a fixed byte offset

(6) QuickTime is very vague about its magic key, it actually defines all top-level atoms as optional, but recommends to usually start a QuickTime file with the ftyp atom

(7) Matroska seems to have an integer magic key

Gradually, we will develop a strategy that includes all of those special cases.

First of all, we summarize and define:

***DES 137:* A top-level container defines one or several magic keys in headers or footers**
Every top-level container must define one or multiple magic keys identifying the container and data format. The magic key is either at the start (header) or at the end (footer) of a container. We introduce a class `MagicKey` for describing the properties of a magic key. It defines:

- The magic key bytes
- A bit length of the magic key
- Optional: A string representation of the magic key
- Optional: A fixed byte-length distance from start of header (positive) or end of footer (negative)

Rationale: Every top-level container must at least have one magic key, because otherwise we cannot identify it. The possibility of multiple different magic keys is implied by e.g. RIFF and QuickTime. ID3v24 defines a header magic key as well as a different footer magic key for backward reading. The bit length in contrast to a byte length is necessary for the MP3 magic key which is not a multiple of 8 bits, but 11 bits long. The fixed byte-length distance from start of header or footer is necessary for formats such as QuickTime.

Disadvantages: No disadvantages known

Having defined this, we already covered the exceptions number (2), (3), (4) and (5). Now let's tackle QuickTime:

***DES 138:* ftyp, pnot, mdat and moov as magic keys for QuickTime**
ftyp, pnot, mdat and moov are defined as the magic keys for QuickTime. If none of these atoms is found, a byte chunk cannot be identified as QuickTime.

Rationale: The vagueness of the specification is a problem, but we can reasonably handle it like that and be quite sure that in 90 percent of the cases, we can read QuickTime files properly.

Disadvantages: Some very special QuickTime byte sequence might not be readable with `jMeta`.

Now we are left with the VorbisComment special case. Unfortunately, you need external context knowledge to just know that the chunk of bytes is a Vorbis Comment. As described in [MC17, MC17]:

- In Vorbis bitstreams, it always is embedded in the second page of the stream
- In FLAC, Speex and theora bitstreams, it is always starting with the second packet of the stream, it is actually not quite clear if this is the same thing

as for Vobis, but note that Vorbis says page and the other say packet which is conceptually different in Ogg!

The VorbisComment, to say it cleary, seems quite oddly designed. We will not invent a generic solution for this, thus we define:

DES 139: **VorbisComment oddities are only handled in the Ogg and VorbisComment implementations**
The generic parsing algorithm of `jMeta` does not provide any support for data formats that have no magic keys such as VorbisComment. Instead, such oddities need to be handled in the individual implementations. We assume that this can be done quite easily in the Ogg and VorbisComment parsing implementations.

Rationale: Handling such a singular special case in a generic way is quite hard and would greatly complicate the generic parser. Implementations should be able to handle it with their special knowledge, which does not need to leak into the generic parser.

Disadvantages: No disadvantages known

We have already covered top-level containers and data format identification. But what about nested containers? Do they also need to have magic keys? In most cases, we have generic nested containers whose exact structure is only revealed at runtime during parsing, e.g. ID3v2 frames, RIFF chunks, Matroska elements and so on. Having an id that is always different, they usually do not qualify to contain magic keys. If all data formats would define a generic nested container type, all would be fine. But again, we have candidates for problems:

- QuickTime defines two different types of nested containers that have a completely different structure: atoms and QT atom containers
- ID3v2 has an odd use of padding by defining it as special null bytes behind the payload and before the footer (if any), thus he have defined to treat padding as special container, see DES 139

Here, we can use magic keys to the rescue:

DES 140: **Nested containers define magic keys if not generic**
Usually, nested containers are generic. Non-generic nested containers are allowed to define one or multiple magic keys in their headers or footers.

Rationale: QT atom containers have 10 zero-bytes at start which can be used as magic key. Likewise, padding always must start with a zero byte which is then the magic key.
Parsers reading payload bytes of a top-level container can first check for the presence of any magic key bytes of defined child containers. Only if these are not found, the single generic child container is assumed to be present.

Disadvantages: No disadvantages known

Note that there are still two problems with the formats to solve when doing it like this:

- QuickTime (again) defines atoms and QT atoms as generic container types that can also be nested containers. QT atoms start with the same structure as atoms, but add some header fields.

- When parsing child content like defined in DES 140, there is a special case for Lyrics3v2: Unfortunately, the payload size of this tag is not known in andvance, but its end is only identified by the magic key in its footer. This special case is treated in DES 140 in another section.

So let us check how to handle yet another QuickTime oddity:

***DES 141:* QT atoms can be identified at runtime**
QuickTime as only format relevant for us defines two different generic container types: Atoms and QT atoms. Thus, we define these two types of generic containers and use the generic container identification schemes to handle the situation. At runtime, if a specific id is found, the atom type is known to be an atom or QT atom, and further parsing is going on correspondly. See DES 141 for the runtime concrete container identification.

Rationale: Luckily, QT atoms just extend the header of atoms and otherwise are structure the same way. Thus this mechanism will resolve the problem.

Disadvantages: No disadvantages known

Based on these definitions, we can be more detailed about the parsing strategies:

***DES 142:* Parsing top-level containers (forward reading)**
For top-level containers, the data format identification is the first thing that happens. Each data format must define at least one top-level container which can either be generic or concrete. Each top-level container type must at least define one header magic key that indicates presence of the format. If the current bytes match the magic key, the data format is identified. Further on, if the top-level container is a generic container, the runtime identification of its concrete type is following as defined in DES 142. Data formats that require deviating behaviour (e.g. VorbisComment) can override some aspects of this behaviour.

Rationale: This is directly following from all previous design decisions

Disadvantages: No disadvantages known

For nested containers, forward reading basically works like this:

DES 143: **Parsing nested containers (forward reading)**
For nested containers, no data format identification is needed. The data format defines all possible containers that might occur in the container-based payload of the parent container. Each of these containers can either be generic or concrete. Any defined nested container might define a magic key. The parser first gets all child containers of the current payload who have a magic key defined and tries to match the current bytes with these magic key bytes. If they match, the child container is found. Further on, if the nested container is a generic container, the runtime identification of its concrete type is following as defined in DES 143. If none of the child containers with magic keys match, the parser gets the *default nested container* which can be obtained from the `DataFormatSpecification`, if set. If this is the case, the parser assumes that the next container is the default nested container, otherwise it cannot identify the next container. If a container is found and this container is generic, the runtime identification of its concrete type is following as defined in DES 143. This might e.g. reveal that an atom (default nested container for QuickTime) is actually a QT atom of a concrete type, see DES 143.

Rationale: This is directly following from all previous design decisions. We have to define a default nested container just because of QuickTime as sole exception defining two very similar generic nested container types.

Disadvantages: No disadvantages known

Now the topic of backward-reading containers is still open. Again, there is a special case here: ID3v1 has no footer, but still is actually backward-readable. The reasons: It is defined to always be at the end of the file. Even if it is not, it could still be identified, as it has a fixed length. This means we would just need to look for the header magic key at a fixed offset from the end of the tag. However, there is a rather hidden special logic that additionally needs to be applied: If you identify the presence of a container by its footer magic key, you yet have to still read backward to obtain its payload. In case of ID3v1, you have arrived at the header already, such that you actually need to read forward to get its payload.

To handle this, again, we have two alternatives: Either bring this special case into the generic layer, e.g. by adding a field to the magic key, which points to the offset for backward reading. Or we handle it only in the ID3v1 implementation. A third esoteric possibility would be to have a fake "footer" in ID3v1, such that we could just say: Whenever backward reading is necessary, just rely on footers only. But this would be very artificial and would lead to other problems. As was the case previously, we decide:

***DES 144:* Backward-reading for ID3v1 is implemented in the ID3v1 parser**
The special case for ID3v1 is not considered in the generic parser code, but it is specifically overridden for this purpose in the ID3v1-specific parsing code.

Rationale: This decision is in line with the other design decisions. We want to keep the basic concepts and generic parser code and classes clear and clean of single-format special cases.

Disadvantages: No disadvantages known

Now, finally, let us define how backward reading is working overall. It is just a feature meant to be used for tags at the end of file. You cannot backward-read MP3 frames or Ogg pages, because they have no footer, such that you would need to "scan backwards" which we do not want to do. The second thought about backward-reading: It only affects the top-level, and not nested containers. To summarize:

***DES 145:* Parsing top-level containers (backward reading)**
A prerequisite for backward-reading top-level containers is that the data format has a footer with a magic key. The user has to exyplicitly request backward reading. The parser checks if it finds a footer magic key for all backward-readable data formats, i.e. data format identification happens here. It starts reading bytes to compare at the end offset of the footer plus the magic key offset (i.e. it reuses the offset defined in the `MagicKey` class, which is negative in this case). If the read bytes match the magic key, the data format is identified. Further on, if the top-level container is a generic container, the runtime identification of its concrete type is following as defined in DES 145. Data formats that require deviating behaviour (e.g. ID3v1) can override some aspects of this behaviour. Backward reading stops as soon as no further backward-readable data format could be identified.
It is not necessary to backward-read nested child containers.

Rationale: This is directly following from all previous design decisions. The reason why nested containers do not require backward-reading: Once you know were a backward-read top-level container starts, you can again forward-read its nested children.

Disadvantages: No disadvantages known

Now we have to clarify how the parsing code can best access the information about magic keys. We decide:

DES 146: **Obtain `MagicKeys` from `DataBlockDescriptions` of containers**
For the parser code, `MagicKeys` can be obtained from a `DataBlockDescription` of a container data block. It offers a method for obtaining the backward-readable (i.e. footer) magic keys as well as a method for obtaining the forward-readable (i.e. header) magic keys.

Rationale: A magic key essentially belongs to a container. The top-level container `DataBlockDescriptions` and the nested container `DataBlockDescriptions` must anyway be available a the places where we need to parse them. By providing different methods for forward and backward reading, the intentions become very clear. Overall, it is most convenient for the parser code this way.

Disadvantages: No disadvantages known

Strongly connected to this is the question how to represent magic keys in a data format while avoiding redundancies. We already defined a specific class called `MagicKey` and methods for obtaining the magic keys for this, which is a good choice as it bundles all the relevant information (and nothing more) for the parser code. But in the end, a magic key is nothing but a header or footer field with a fixed value. Thus we define:

DES 147: **`MagicKey` instances implicitly derived from field `DataBlockDescriptions`**

The `MagicKey` instances that can be queried by the parsing code are not defined/added in parallel to the field properties anyway needed. In contrast to this, a `DataDataBlockDescription` of type `FIELD` additionally contains a flag saying if it is a magic key or not. If this flag is set to true, all other properties of the `MagicKey` can implicitly derived from the other field properties and a plausibility check can happen. If one of the following plausibility checks is failing, the `DataFormatSpecification` is considered invalid.

- A field marked as magic key must either have a fixed value, or it must be an enumerated field (i.e. with multiple alternative values).
- The header that contains the magic key field must be either be the first header or the headers before it must be fixed length headers. For footers: The footer that contains the magic key field must be either be last footer or the footers behind it must be fixed length footers. Otherwise the field does not qualify as magic key, because it might have a variable offset from the start or end of its container and the `DataFormatSpecification` cannot derive the necessary offset from start or end of container.
- For the same reason, we define for headers: The fields in front of the magic key field (if any) must all be fixed length fields. And for footers: The fields behind the magic key field (if any) must all be fixed length fields.
- There must be only at maximum one header magic key field and only at maximum one footer magic key field

Rationale: Redundancy is avoided, and it is more convenient to define a concrete `DataFormatSpecification`, as we do not need to repeat ourselves.

Disadvantages: A `DataFormatSpecification` implementation must be slightly more complex to derive and check the necessary properties of the `MagicKey` instances.

There is a special case with MP3 here: It has an odd number of bits. This makes defining the magic key as single field very hard. Instead, it needs to be defined as flag field and cannot be derived automatically:

DES 148: **MP3 magic key is added manually to the specification**
The MP3 magic key with 11 one bits is added manually to the MP3 `DataFormatSpecification`. Thus `DataBlockDescription` offers a method for adding a header or footer magic key manually. So auto-derival of magic keys is not used for MP3.

Rationale: The odd bit length does not allow to define the 11 bit field as field on its own, but it needs to use flags, and flags do not have individual default values. Thus it is not possible in the generic code to derive the magic key for MP3 automatically. Instead, the field is not tagged as magic key, but it is manually created in the MP3 extension and added using the `DataBlockDescription` method.

Disadvantages: No disadvantages known

Last but not least, we want to discuss the topic of different data format versions and their magic keys. Specifically, one could ask: Does the version number belong to the magic key of a data format or not? We define:

DES 149: **Magic keys do not include a version number**
Magic keys never include the version number of the data format, even if it directly follows or preceedes the magic key and it might appear as solution to the problem of how to distignuish similar data formats.

Rationale: First of all, a version number might be numeric, while magic keys are usually strings. Does the string representation of the version number would be slightly odd. Second and more importantly, it is actually a different field, and it could well be that a data format does not define the version field to be directly adjacent to the magic key field.

Disadvantages: No disadvantages known

11.2.7. Inheriting and Overriding Datablock Structures

TODO

11.2.8. Occurrences and Sizes

In this section, we discuss the topics of how often does a data block occur, and how long is it?

The "how often" (occurrences) part is treated as follows:

DES 150: **`DataBlockDescription` contains minimum and maximum occurrences**
For every data block, its `DataBlockDescription` can specify the number of occurrences by giving its minimum and maximum occurrences. The rules:

- Both must be set to a positive number, where only for the minimum occurrences zero is allowed
- If both are set, minimum occurrences must be smaller or equal compared to maximum occurrences

Rationale: A few formats have fields that occur not only once, but multiple times sequentially. Some fields exist whose occurrence count instead of their length is given: Vorbis comment (number of user comments), APEv2 (number of items, in addition to size), Ogg (number of segments, which is equal to size, as each segment has one byte as length).

Disadvantages: No disadvantages known

For the size of a data block, we have basically three possibilities:

- The data block has a constant, fixed size which is always the same. Most fields have this property, but also static format containers such as ID3v1.
- The data block is a variable-size field and it is terminated, see "11.2.9 Field Properties" for more details about this case. This leads to variable-size headers, footers, payload and containers that have a variable size because one of their fields is a variable-size field.
- The data block size is dynamically determined by a previously read field value or a combination of such field values. This case is further detailed in section "11.2.10 Field Functions".

For each case, we need the following:

DES 151: **`DataBlockDescription` contains minimum and maximum byte length**
`DataBlockDescription` allows to set a minimum and maximum length of the data block in bytes. Either the minimum length is set to a positive number (bigger than zero!) and the maximum length is set to "unknown length", or both are set to a positive number (bigger than zero), where the maximum length must be equal to or bigger than the minimum length.
The fixed size case is covered by setting both to the same value. Differing values have no specific influence for the parser, but can be later used for format validation of a chunk of bytes. If not set or the values differ, this is a signal that the data block is either terminated or its size needs to be determined dynamically using other field values.

Rationale: `jMeta` needs to be able to reflect both static sizes as well as dynamic sizes, and to be future-proof, it is also good to have differeng min and max sizes given for data format validation.

Disadvantages: No disadvantages known

How to deal with uncertainty? It is not always the case that lengths or counts are known. We want a constant here that clearly communicates

DES 152: **Constant for undefined or unknown length and offsets**
We define a constant in `DataBlockDescription` which has the meaning of an undefined or (yet) unknown length or offset. The value of this constant is the smallest negative long.

Rationale: This is e.g. necessary for fields with dynamic length or for relative offsets that cannot be computed due to dynamic lenghts. Having only one constant for all reduces confusion and still clearly communicates intent. Using `null` for undefined is not clearly communicating intent and is less readable. The reason for choosing the smallest negative long is: While offsets might get negative in some cases, lengths and counts never do. It is reasonable to assume that the negative relative offets we need in `DataFormats` never reach the smallest long.

Disadvantages: No disadvantages known

We just cover one additional special case here: In DES 152 and the follow-up discussion, we already noted that Lyrics3v2 is somewhat special in terms of how to determine its payload size: It cannot be done before parsing it. Instead, it is only known to end when the footer's magic key is found. Thus it is kind of "terminated". In a first design, this was handled by some kind of special magic keys, so-called "exclusion keys" that where defined to state "if you find this key, a container of the current type IS NOT present". However, we decided to redesign this to be less odd:

***DES 153:* Lyrics3v2 payload size is determined by reading all children**
Lyrics3v2 special case (no size indicator in header) is treated by reading all children until its footer. This is done for any data format where no payload size could be determined after reading the headers.

Rationale: This case is currently only known for Lyrics3v2, but the generic handling is not too complicated but rather simple, thus not requiring a custom implementation for Lyrics3v2 only.

Disadvantages: No disadvantages known

11.2.9. Field Properties

Now we look into properties that are specific to data blocks of type `FIELD`. Here, we cover other properties specific to fields. However, we do not cover the details of how field conversion (interpreted value to binary value and vice versa) is working and how some special cases of field interpretation are handled. This is a topic for the `DataBlocks` component.

We already covered sizes of data blocks (including fields) and we already defined one property of a field: It might be a magic key field, see DES 153. The fields are the leafs of the data block hierarchy, thus, ultimately, they represent concrete values. While we could decide to treat them only as raw bytes, this is counter-productive as it would make parsing less convenient. Thus we define:

DES 154: **Fields have one of the types binary, string, numeric, string list or flags**
Every field has a field type, and thus not only a **binary value**, but also an **interpreted value**. The field types are:

- **String:** The field's value is representing a string in a specific encoding.
- **Binary:** The field's value is not further interpreted, i.e. the interpreted value is identical to the binary value of the field. I.e. this could be used to represent raw encoded bytes of a codec.
- **String List:** The field's value corresponds to a list of strings.
- **Numeric:** The field's value is representing a numeric value with a specific byte order.
- **Flags:** The field's value is representing flags, i.e. a set of bit toggles, e.g. vital for parsing.

Rationale: Parsers can directly read the interpreted values of the fields they need for parsing (e.g. sizes, flags or others) without needing to implement conversion to an interpreted value themselves. Interpretation should be kept inside the format specific code - in this case in the data format specification.
All of the field types of our reference data formats can be covered by these types. The string list type is unfortunately necessary just for the sake of ID3v2.4.

Disadvantages: No disadvantages known

First, we will care about the string and string list fields:

DES 155: **String fields and string list fields support termination characters and fixed character encoding as specific properties**
A non-fixed-size string field or string list field can be terminated by a single termination character given in the field's character encoding known at parsing time. Second, a string field might have a fixed character encoding to be used. Both are optional properties.

Rationale: String fields can be terminated by termination characters, this is needed e.g. in ID3v1 and ID3v23 data formats, where the actual number of termination bytes depends on the current character encoding. Thus, it is best to define a termination character and then decide at parsing time, which bytes according to the current character encoding are expected as termination bytes. For string lists, the termination character also acts as separator between different list entries. See also DES 155.
Some specifications define fixed character encodings for specific string fields that might deviate from the data format's default character encoding.

Disadvantages: No disadvantages known

What about binary fields? Basically, they do not need any additional properties, but we want to make one thing clear about termination:

***DES 156:* Binary fields do not support termination bytes**
While string fields can define a fixed termination character, binary fields do not support termination bytes.

Rationale: The reason is simple: Within our reference formats, there is no single case where a terminated field exists that is not a string field.

Disadvantages: No disadvantages known

Numeric fields require a byte order such that their numeric value can be interpreted:

***DES 157:* Numeric fields support a fixed byte order**
Numeric fields define a fixed byte order to use during conversion. It is an optional property.

Rationale: To be convertible into a numeric value, the parsers need to have information about a byte order to use.

Disadvantages: No disadvantages known

For flags, it would be convenient to describe them in particular, e.g. which flags are there, where are they and what is their meaning. Thus we introduce a new class:

***DES 158:* Flag fields require an instance of `FlagSpecification` that describes all flags in a flags field**
The class `FlagSpecification` describes a flags field having following properties:

- The fixed byte length of the flags field
- The positions (byte and bit adress) and unique names of flags together with their specification descriptions
- A byte order

It also supports multi-bit-flags.

Rationale: It is more convenient to access the flags, in contrast to doing bitwise OR and AND yourself.

Disadvantages: No disadvantages known

One question that could arise now is: Why do we not have "enumerated fields"? Actually, we had them in a previous version. They could allow for an *arbitrary*

interpreted type, allowing the end user to map arbitrary objects to their byte representation for a field. Although this might seem very flexible and extensible, we explicitly decided to remove it:

DES 159: **No "enumerated" field type, but an enumeration property**
There is no such field type called "enumerated", but only the already mentioned field types. Instead, every field has an additional property, basically a mapping that enumerates all possible interpreted values mapped to their unique binary interpretation. This way, it is still possible to provide a fixed value set for e.g. validation or specific conversions.

Rationale: The main problem with an explicit enumerated field type is its *arbitrary* interpreted type. The runtime cannot know which type it is and requires very adventurous casting to get it done, or you need to write special code essentially duplicating the same logic for enums vs. concrete types. As we can still enumerate, this is fine. The only place where it hurts a bit are fields that are representing a Charset or a ByteOrder. Here it would be nice to have an enumerated field type as Charset or ByteOrder. However, it is simply possible to map bytes to string representations of Charsets and ByteOrders, i.e. having a String field instead.

Disadvantages: Maybe less flexibility for future demands and less support for actual object orientation

As we have these default field types, we can also define default field conversion mechanisms:

***DES 160:* Default field converters for every field type**
`DataFormats` provides default field converters used to convert the binary value of a field into an interpreted value. It first looks if there are enumerated values and maps values accordingly. If there is no matching enumerated value, the converter provides a default appropriate conversion based on provided character encoding and byte order. There are two conversion methods:

- `toInterpreted`: Converts a binary value to an interpreted value, taking current character encoding and byte order as parameters
- `toBinary`: Converts an interpreted value to a binary value, taking current character encoding and byte order as parameters

So we can implement the following default conversion mechanisms:

- String: Convert the string into bytes and vice versa according to the provided character encoding
- Binary: Identical conversion
- Numeric: Convert the number into bytes according to the provided byte order, assuming unsigned integer
- Flags: Put the binary value into a Flags instance and vice versa

Rationale: Conversion is quite standardized according to the standardizes field types, this might cover at least 80 percent of the conversion cases. Conversions are relatively simple such that it is fine to keep them in `DataFormats`.

Disadvantages: No disadvantages known

But standard converters are not enough. What happens if you need signed integers instead of unsigned? This is where a custom converter comes in:

***DES 161:* `DataFormats` allows to add custom converters for each field**
Users may add custom converters to a field upon `DataFormatSpecification` creation. A custom converter is a property of the field.

Rationale: E.g. necessary for ID3v23 (sync-safe integers), Lyrics3v2 (numbers represented as characters) and Matroska (VINTs).

Disadvantages: No disadvantages known

Finally, some fields - no matter what type - have always the same static value. We join this requirement with the notion of a default value, i.e. a value to be preferred when it is not known which would be the right value of a field.

***DES 162:* Each field has an optional default (interpreted) value**
Each field has an optional default (interpreted) value.

Rationale: The default values are needed for fields with fixed values, and might be used to fill field values in cases where they are unknown.

Disadvantages: No disadvantages known

11.2.10. Field Functions

Some fields contain values that are contributing to so-called *parsing metadata*, i.e. metadata that is required to correctly read the contents of a container. Most prominent case is a size field which e.g. contains the size of the payload in bytes. When parsing the container, it is vital to know where the container ends and the next one starts. So this context information is necessary for reading. But how to implement generic reading if different data formats have different size fields? A second question that comes up is: How to ensure the size field is consistently updated if the payload size increases? Of course we could burden this to the user of the library. But there is an idea that solves both problems at once: *Field functions*.

***DES 163:* `jMeta` uses field functions to link parsing metadata fields with the target data blocks they refer to**
To ensure consistency of container content and parsing metadata, so-called field functions are used in `jMeta`. They represent a reference of a single field to one or multiple target data blocks of any type, supporting the following categories:

- `SizeOf`: The field contains the size of a single other data block
- `SummedSizeOf`: The field contains the summed size of several other data blocks that must be consecutive
- `PresenceOf`: The field indicates that an optional data block is present or not
- `CountOf`: The field contains the number of occurrences of a data block that might have multiple occurrences
- `ByteOrderOf`: The field contains the byte order of a data block
- `CharacterEncodingOf`: The field contains the character encoding of a data block

Rationale: Formalizing connections between data blocks is at first allowing to ensure consistency and facilitates a generic parsing approach. The categories represent the most common types of parsing metadata encountered.

Disadvantages: Added complexity

Future functions might be introduced such as "CRC of", but this is yet to be planned.

How this generic parsing approach exactly works is described in "11.3.3 Container Context Information".

Here we focus on the specification aspects of field functions. First of all, let's examine some sane validity criteria for each field function. Table 11.8 is listing them.

Field function	**Validity criteria**
`SizeOf`	Must be a numeric field; must refer to exactly one target data block
`SummedSizeOf`	Must be a numeric field; must refer to at least two sibling data blocks which have the same parent; at most one of the target data blocks must have undetermined size (i.e. neither fixed size, nor terminated nor `SizeOf` function for it)
`CountOf`	Must be a numeric field; must refer to a data block of type header, footer, container or field (i.e. not to payload)
`PresenceOf`	Must be a flags field; must refer to an optional data block of type header, footer or field (i.e. not to payload or container)
`ByteOrderOf`	Must be a string field
`CharacterEncodingOf`	Must be a string field

Table 11.8.: Validity criteria for field functions

DES 164: **`jMeta` enforces the validity criteria listed in table 11.8 for field functions**
The field functions supported by `jMeta` adhere to the validity criteria listed in table 11.8. That means that these criteria are checked at specification creation time and lead to runtime exceptions if not met.

Rationale: These criteria ensure that an extension cannot use nonsense functional links which lead to arkward errors during parsing. Specifically for `SummedSizeOf`, forcing just at max one with undermined size is ensuring that there are now infinite loops during parsing.
Why only these and not more? E.g. we could enforce that `SizeOf` refers only to dynamic size targets. This is not done because there could be formats defining a size as static in current version, yet still having a `SizeOf` field to refer to.
Similar things could be said for `CountOf` and dynamic occurrences. One could force `ByteOrderOf` to refer only to numeric fields or `CharacterEncodingOf` only to string fields. However, this is not helpful as a specification might state the byte order or character encoding just once referring to all fields.

Disadvantages: No disadvantages known

Now we are just left to discuss the complexities of the `SizeOf` field function and why we need a separate `SummedSizeOf` field function. The problem with the `SizeOf` field function is that in a lot of formats, the size is not only referring to exactly the payload or just another field, but in a myriad of cases to multiple objects which we call data blocks - here are those dreadful examples from our reference formats:

- The ID3v2.3 and ID3v2.4 tag size refers to the tag payload size (including padding) *plus* the size of the extended header, if any
- The ID3v2.3 extended header size is the size of up to 3 fields that follow up the size field itself within the extended header
- The ID3v2.4 extended header size is the size of the entire extended header including the size field itself
- For RIFF and AIFF, a chunk's size field refers to the size of the form type field (fixed size of 4 bytes) *plus* the size of the actual chunk's payload
- In QuickTime file format, the atom size is the size of the whole atom considered as container, i.e. header *plus* payload
- In APEv2, the size field in header or footer is the size of the payload *plus* the footer's or header's size, if present
- In APEv1 and APEv2, the item size is the size only of the value, i.e. not including key or flags
- For Lyrics3v2, the tag size includes the payload size *plus* the size of the header

If each and every format would use the very same strategy to determine the size of its payload, it would be nice. Instead, we see that nearly every format has its special case. Not to mention that MP3 and Ogg do not have size fields at all but rely on a more complex way of calculating the payload size based on combining multiple fields in the container header.

How to handle this overwhelming variety? First of all, from a specification point of view, we have to state:

DES 165: **`SummedSizeOf` field function in addition to `SizeOf` to support multiple consecutive target blocks**
A `SummedSizeOf` field function refers to at least two data blocks that - however - must be strictly consecutive according to their specification. This has the meaning: This field is the *summed size of all* the referenced data blocks, if present.

Rationale: This is cleary required from a data format perspective as outlined above. The question is why not to use `SizeOf` both for single and multiple fields? The special case turns out to simplify implementation.

Disadvantages: Of course, field function handling will get more complex due to this special case.

Other implications of this complexity are dealt with when designing the actual parsing process.

11.2.11. Header and Footer Properties

11.2.12. Container Properties

11.2.13. Payload Properties

11.2.14. Builder API

From all that has been said previously, it might already be clear that a `DataFormatSpecification` is a complex thing: We have data blocks of very different types, each type with its individual properties and corresponding validity aspects, organized hierarchically. If we want user to create this, this can get very nasty. Actually, in a first version, there was just a plain old java interface for creating a `DataFormatSpecification`: A lot of boiler-plate code, creating and populating collections, wiring parents and children, filling attributes that have no meaning for a type with their default values, filling up values such as lenghts for parents and children redundantly etc.

So this is not only a pain to write, but also a pain to read and maintain. Thus we introduced a builder API as a quantum leap in contrast to the original API:

DES 166: **Builder API for creating valid `DataFormatSpecifications` and `DataBlockDescriptions`**
`DataFormats` provides a comfortable builder API for building a `DataFormatSpecification` from its individual `DataBlockDescriptions`. It uses method names for providing data block type and if it is generic or not, method parameters for adding the local id, name and description, and nested methods for setting other properties and adding children to a data block. It removes a lot of boiler-plate code, ensures the user only sees the properties that are really valid for the given type, and wires parents and children as well as their ids correctly.

Rationale: The user can add a new `DataFormatSpecification` much more conveniently. Furthermore, the code remains short and readable (thus maintainable), still communicating only what happens without any noise.
In addition, we noticed the following interesting benefits of the builder API:

- The user is not required to directly depend on the implementation of `DataFormatSpecification`
- Cloning of data blocks can be implemented and provided in an easier way

Disadvantages: The builder API of course needs to be built and maintained, it is a complex structure of interdependentend classes and interfaces with a lot of generics magic.

11.3. `DataBlocks` Design

In this section, the design of the component `DataBlocks` is described. Basic task of the component is to implement the actual generic parsing of data of a supported data format in a well-performing, yet flexible and extensible way.

11.3.1. Basic Concept for Reading

In section "11.2.3 Representing a Data Format", we discussed the approach that a data format is described in terms of a *data format specification*, which basically is a description of how data is represented as a chunk of bytes in the data format. It breaks up such a chunk into different types of so-called data blocks that form a well-defined hierarchy (see "11.2.2 The Container Metamodel"). Given such a specification and data block hierarchy, we define the following design decisions:

DES 167: **Data block instance classes defined in `DataBlocks`**
The instances for the different data block types are represented as Java classes in the `DataBlocks` component which use the same names as in figure 11.1.

Rationale: As chunks of bytes are structured according to the metamodel and the data format specification describes them as such, it is quite clear the have to be instance of corresponding classes during parsing. These instances are also returned to the user and provide a clear, better-to-understand view on the data. Btw., having a metamodel but not transforming it into a class model seems to be quite nonsense.

Disadvantages: No disadvantages known.

DES 168: **Specification-driven Read Approach**
Reading is heavily based on the description of the data format in its data format specification. The descriptions of lengths and types are used as basic assumption for parsing, especially magic keys for first identification.

Rationale: It is quite clear that otherwise, the data format specification would be quite useless and we would implement some data-format-specific and thus non-generic code again.

Disadvantages: Is it flexible enough? If we stick to a stiff specification, how to ensure that it is flexible enough for different data formats?

DES 169: **Reading and writing with `Media`**
Reading and writing is done using `Media`, including use of its caching features.

Rationale: Again this is very clear.

Disadvantages: No disadvantages known.

Given these basic design decisions, we are ready to sketch a rough draft of how reading basically works. We first summarize the very fundamentals that we know this far:

- Data is represented as linear chunks of bytes whose structure is defined by the data format specification
- The top-level data blocks for reading are containers
- It is not clear which data format we have in front of us
- A data block might get large, as we use long as data type (see DES 169)

Given the first two observations, we define the following:

***DES 170:* Iterator approach for reading**
`jMeta` provides an iterator pattern for iterating top-level containers.

Rationale: As the basic structure of a chunk of data format bytes consists of linear disjoint containers and we need to do forward-reading, an iterator is a quite natural choice. Lists or other collection data types would be insufficient, because they would suggest that there is a finite number of containers. Iterators are a good fit for streaming, allowing virtually "unlimited" streams of containers.

Disadvantages: No disadvantages known.

Now we can define the basis approach of forward reading, i.e. what is done if we read a top-level container? Of course, first it is not clear which data format we have - identification is necessary, already introduced in "11.2.6 Data Format Identification and Magic Keys". Second, it might happen anytime that we hit end of medium, either expectedly or unexpectedly. The basic flow is defined in the following design decision and shown in figure 11.2.

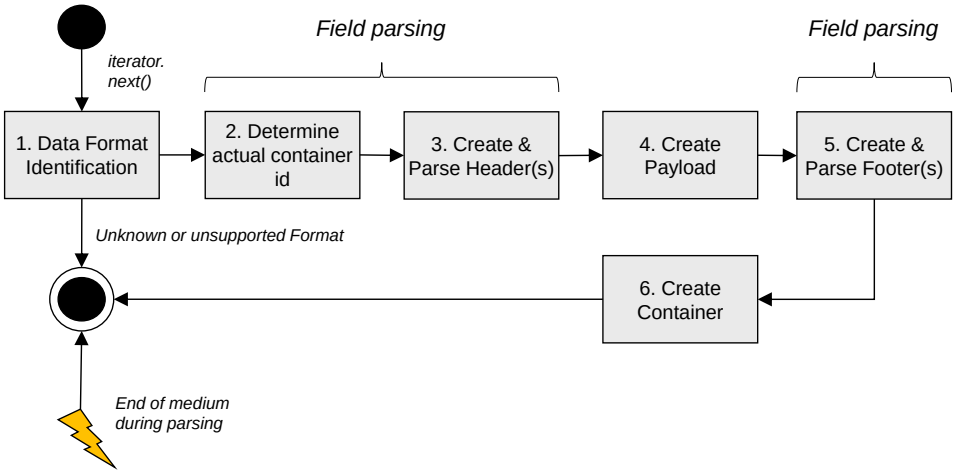

Figure 11.2.: Steps for reading data forward

***DES 171:* Steps of forward reading of top-level containers**
Forward reading of a single top-level container works as follows - whenever `next()` on the iterator is called to read the first or next container:

1. First, we do data format identification, i.e. finding out whether the data chunk ahead belongs to a data format the library supports - see "11.2.6 Data Format Identification and Magic Keys" for details. If we cannot identify the data format, the process stops here with an exception.
2. After identifying the data format, we need to identify which concrete container type and id we have, i.e. determining its structure and actual id at runtime; this is especially necessary for data formats supporting generic containers which is most of the container data formats out there, or multiple different container types.
3. Then, as we forward-read, the headers of the container are parsed, which means understanding their content. This includes determining if there are other headers or footers, and also the length of the container's payload.
4. After heaving determined the payload length, the payload object itself is created.
5. Then, if there are any footers, they are at least created and ready to be parsed.
6. Finally, the container is created which consists of the previously created headers, footers and payload

It might happen that during this process, an expected or unexpected end of medium occurs. Expected it can only be during data format identification, in all other cases there seems to be a corruption in the data or parallel changes, as the parsing metadata indicates a size which is not matching reality.
During steps 2, 3 and 5, fields need to be parsed which is a special discipline discussed later.

Rationale: The process is aligned at the structure of a container and typical for forward reading.

Disadvantages: No disadvantages known.

Btw:

***DES 172:* Steps of forward reading for nested containers**
These are nearly identical except that data format identification is unnecessary.

Rationale: Nested containers are other than the fact that the data format is already known in no way different to top-level containers.

Disadvantages: No disadvantages known

11.3.2. Buffering, Caching and Lazy Reading

`DataBlocks` uses the features of `Media` for caching as described in "11.1.1 Caching". These features are also used for buffering, i.e.

***DES 173:* General buffering using maximum read-write block size is done - for each overlap buffer the next block**
Most of the read operations in `DataBlocks` do a buffering before accessing data. They always buffer at most maximum read-write block size of bytes. If a buffering call detects that the start offset of buffering plus maximum read-write block size exceeds current buffered data, it buffers the next block starting behind the currently buffered data with at most maximum read-write block size of bytes. This decision strongly bases on the critical design decision DES 173 for a sensible handling of objects (e.g. fields) overlapping two consecutive byte blocks read.

Rationale: Reading byte by byte or field by field would be nonsense from performance point of view. The maximum read-write block size is a natural fit for buffering. The approach of buffering the next block when an overlap is detected is sensible as it leads to less cache fragmentation. The previously cached consecutive block will not fall out of the cache thanks to design decision DES 173.

Disadvantages: No disadvantages known.

Now the question arises: Where to buffer exactly to ensure that there is no "buffer gap"?

***DES 174:* Buffering at start of container and field reading**
It turned out to be fully sufficient if we do the following:

- Buffer at start of container identification/reading: Whenever it is checked if a container with a given type and id during container identification phase is done, we first buffer read-write block size of bytes. As this size has a lower bound (see DES 174) with a sensible default, this way all usual headers are already buffered, leading to speeding up data format identification for top-level containers.
- Buffer at start of field reading: Whenever a single field of a header, footer or payload is read, buffering is done. Due to DES 174, any overlapping bufferings will actually read the next block in addition, ensuring that always as sensible amount of future bytes is pre-buffered. This case also covers field based payload.

Rationale: Thus, even if we have unusually long headers, footers or field-based payload, there is never a buffering gap.

Disadvantages: No disadvantages known

A very basic mechanism of container data formats is that they allow seeking and skipping based on containers, that means the decoders might just need to check a container header to find out if they are interested in the data it contains, and if not simply go on reading with the follow-up container. For this, most if not all data formats specify the size of the container in its header or footer. To fully support this, we define some kind of lazyness as follows:

***DES 175:* Lazy payload**
Any field-based or container-based payload is lazy by default, i.e. its bytes are first read only when explicitly requested by the user. A sole exception is in place for stream-based media (see upcoming design decision).

Rationale: This way, we do not read unnecessary data the user might never look at, especially allowing to skip over large containers without risking out of memory and long read times, just perfectly meeting requirement "4.8 REQ 008: Read and write large data blocks".

Disadvantages: No disadvantages known

As streams are non-random-access media, the question arises what we do with the bytes of a stream. It is no good idea to skip the payload byte here - what if the user wants to closely look at them? There is no other choice than to still cache them. The same is true for rather "exotic" formats such as Lyrics3v2 where the payload size is not contained in a header, but only in the footer.

***DES 176:* Fully cache payload for stream-based media and formats without size field**
For stream-based media, the whole payload data must be fully read to at least have a chance that the user can look at it later. It might happen that the maximum cache size is exceeded in which case the first bytes of the payload might fall out of the cache.
For formats not having a static payload size and not having a size indicator in its header, the payload must be fully read in entirety to know its size. This also leads to potentially filling up the cache.

Rationale: There is no other way for streams than grabbing the bytes when looking at them - streams are just one time read.
For the special case of formats not having a payload size in their header, see DES 176.

Disadvantages: No disadvantages known

11.3.3. Container Context Information

In section "11.2.10 Field Functions", we already mentioned the *parsing metadata* that is vital for understanding and reading a container. This information is tagged as such by each extension in the form of field functions that link fields to target blocks they refer to. This information is of course relevant for generically parsing a container at runtime. As soon as you encounter e.g. a field that contains the size of the container's payload during parsing, you have to store this size somewhere to be accessible later when actually parsing the payload.

Here, we summarize all the parsing metadata known at one point in time during container parsing as the *container context information*. However, the container context information shall not only be a simple map storing the already known field functions, but it is intended to be more:

DES 177: **`ContainerContext` provides sizes, counts, byte orders and character encoding of all data blocks in a container; it is available in each data block; one instance per container**
A helper class `ContainerContext` provides sizes, counts, byte orders and character encoding of *all* data blocks in the current container. That means it not only stores the field functions refering to some of them, but also knows about static sizes, static occurrences, default byte orders and character encodings etc. It is an attribute of each data block such that it can be accessed from most places during as well as after parsing. The details of size determination are described in "11.3.3 Size Determination", the details of count determination in "11.3.3 Count Determination".

Rationale: Having just one place that not only stores field function values but also knows in a more abstract way how to determine sizes, counts, byte orders and character encodings allows for better separation of concerns. This also allows to pass around corresponding objects to everywhere you need this information in contrast to having this logic only in a reader class that is not availabe anywhere else after parsing.

Disadvantages: No disadvantages known

One question that comes up here: How about parsing metadata available in the top-level container, but needed in a child container? We define:

DES 178: **`ContainerContext` contains a reference to the direct parent container's context and searches the parent for metadata if none is found in itself**
`ContainerContext` contains a reference to the direct parent container's context (if any). If it cannot come up with a size, count, byte order or character encoding itself, it asks the parent for such metadata.

Rationale: This way, parsing metadata only defined in the top-level container headers can be transported to children deeper in the hierarchy.

Disadvantages: No disadvantages known

Size Determination

How to determine the sizes of data blocks before parsing? It is clearly necessary to know the size of a container in advance to be able to parse or skip it. Unfortunately, all data formats seem to follow a distinct strategy of how to specify sizes as indicated around design decision DES 178.

The cases we have to handle are summarized as follows:

***DES 179:* Cases for size determination**
The easiest **Case 1:** Static size indicated in the specification. If a data block's specification indicates a static size, this size is taken during parsing.
Otherwise, the data block has a dynamic size which breaks down to the following cases:

Case 2: The data block's size is directly indicated by another field that must be parsed before (usually in header or footer) - Note that this size might include the size of other blocks in addition, see DES 179 and the discussion around for more details. Here, "directly" means that the field is a numeric value greater or equal to the size, and we do not need to calculate something.

Case 3: The data block's size is indirectly indicated by other fields that must be parsed before (usually in header or footer). "Indirectly" means that the field is not just single or summed size contained in a field, but there is a more complex calculation of combining multiple field values to calculate the size. This is e.g. the case for the size of the MP3 container payload or the Ogg page payload and packet part sizes.

Case 4: The data block is payload consisting of fields or containers, and there is no size indicator at all. However, it is clear how to determine the sizes of all children of the payload block, and thus the size of it is the sum of the sizes of all its children. This is e.g. the case for Lyrics3v2 tag payload.

Case 5: The data block is a terminated field, i.e. its size must be determined by finding its termination. The handling of this case is defined in "11.3.4 Terminated Fields".

Case 6: None of the other cases applies, but at least the overall remaining size of the parent data block is known. Take the remaining size of the parent data block as size of the data block.

`jMeta` supports exactly those cases. Despite case 5, they are implemented in the `ContainerContext`, see "11.3.3 Container Context Information".

Rationale: The cases come from analyzing different data formats `jMeta` is required to support, so it is sufficient to consider only those (if any others exist). Why is case 5 not implemented in `ContainerContext`? The reason is that finding termination bytes requires parsing itself, so it must be implemented in the reader.

Disadvantages: No disadvantages known

Case 1 is already clear how to deal with, case 5 is treated later in "11.3.4 Terminated Fields". Thus we are left to deal with cases 2 to 4. First of all, we consider case 3:

DES 180: **Complex size calculation from fields done in custom implementation via `SizeProvider`**
Data format extensions are allowed to register a custom `SizeProvider` implementation which calculates the size of a data block as required by the format. This is used for Ogg, ID3v2.3 extended header size and MP3.

Rationale: More flexibility also for yet other future formats.

Disadvantages: No disadvantages known

Case 4 is handled as follows:

DES 181: **Determine payload size by reading all children**
In case 4 of DES 181, the size of the payload is calculated by reading all children, may it be containers or fields, and summing up their sizes. If at least one of their sizes could not be determined, this is a runtime exception.

Rationale: Straightforward and not too complex in terms of "lazyness". If the data formats design is fucked up, the implementation cannot easily work around it.

Disadvantages: No disadvantages known

Now we have case 2 left. This one is adressed with using field functions - potentially referring to multiple consecutive blocks as indicated by DES 181. We clarify how it works by summing up the whole progress of size determination in total:

***DES 182:* Size determination process**
Whenever the size of a single data block needs to be determined, the following steps are performed in the given order of precedence:

1. (Outside of `ContainerContext`) If the data block is a terminated field without fixed size, search for the termination character (case 5)
2. Otherwise if the current data format has a registered `SizeProvider` (see DES 182), first ask this instance if it can provide a defined size (includes also case 3)
3. Otherwise if the data block has a static size according to its specification, take this size (case 1)
4. Otherwise if the current `ContainerContext` has a `SizeOf` field function, take the size indicated by the `SizeOf` field (part 1 of case 2)
5. Otherwise if the current `ContainerContext` has a `SummedSizeOf` field function for multiple data blocks and the sizes of all other blocks are known, take the `SummedSizeOf` field's value minus the summed size of the other data blocks (part 2 of case 2) - How to avoid an infinite loop here? This has been treated in design decision DES 182 and is also discussed with an example after this design decision.
6. Otherwise if the data block is a concrete data block that is derived from a generic data block, it is checked if there is a known size for the generic id
7. Otherwise if the parent `ContainerContext` has a `SizeOf` field for it, take the size of the parent `ContainerContext`
8. Otherwise if the data block is a field or payload and the remaining parent size is known, take the remaining direct parent size as its size (case 6)
9. (Outside of `ContainerContext`) Otherwise if the data block is a payload block, read all its children and sum up their sizes (case 4)

Rationale: The order of steps clearly focusses on least complexity first and tries to avoid unnecessary lookups by first considering fixed sizes and custom size providers.

Disadvantages: No disadvantages known

Some subtle point in determining sizes of single data blocks whose `SummedSizeOf` field is the sum of multiple consecutive data blocks is: If we want to determine the size of a single one, we have to determine the size of all the others, sum this size and subtract it from the value stored in the `SummedSizeOf` field. If for some stupid reason there are multiple data blocks included with a dynamic size, this could end up in an infinite loop such as for ID3v2.3: Size of payload needs to be determined, thus determine the size of the extended header first. However, the

size of the extended header is also dynamic. The implementation would see that it would need the size of the payload to calculate the size of the extended header and thus call the first function again... To avoid this crap, we already stated the validity criteria in DES 182.

Count Determination

Here we discuss how the determination of the number of occurrences of a data block (i.e. count determination) works. First of all, we can state that the number of payload containers is always 1, so no need to consider this case. For all other types of data blocks, there can be 0 to multiple occurrences.

The process to handle is summarized as follows:

***DES 183:* Count determination process**
Whenever the count of a single data block needs to be determined, the following steps are performed in the given order of precedence:

1. If the current data format has a registered `CountProvider`, first ask this instance if it can provide a defined size
2. Otherwise if the data block has a static number of occurrences according to its specification, take this count
3. Otherwise if the data block is optional, i.e. either is present once or not at all, and if the current `ContainerContext` has a `PresenceOf` field function, take 1 as count if its presence is indicated by the flags, 0 otherwise
4. Otherwise if the current `ContainerContext` has a `CountOf` field function, take the count indicated by the `CountOf` field
5. Otherwise if the parent `ContainerContext` has a `CountOf` field for it, take the size of the parent `ContainerContext`

Rationale: The order of steps clearly focusses on least complexity first and tries to avoid unnecessary lookups by first considering fixed counts and custom count providers.

Disadvantages: No disadvantages known

11.3.4. Field Parsing

Field parsing is the most complex part of reading data format bytes. It requires answers to following questions:

- How to determine the size of the field? - This has been answered already in "11.3.3 Size Determination", except the case of terminated fields which we treat here

- How to determine the number of occurrences of a field, especially the case of optional fields? - This has been answered already in "11.3.3 Count Determination"
- Which byte order (for numeric fields) and character encoding (for string or enumerated fields) needs to be used when? - This has been answered already in "11.3.3 Container Context Information"
- How to convert the binary field value to an interpreted one?
- How to use caching during field parsing?

Terminated Fields

Terminated fields are those fields with a dynamic length that is not given by the value of another field, but these fields are delimited by a sequence of termination bytes or characters.

Terminated fields are worst-case for parsing: First of all you never know when it ends, and second you have to deal with strings which brings different encodings into play.

Let's deal with the first problem, the unknown length. Of course, in the "usual" case a terminated field is still a field and thus relatively small. We will only rarely encounter terminated fields longer than several KB. In order to be fully generic, one needs to deal with the worst case of fields having up to `Long.MAX_VALUE-1` as size. Having said this, also block-wise reading is required as we do not want to risk out-of-memory conditions by soaking too much bytes into memory. One could come up with the idea of using some upper bounds for reading data, i.e.:

- **Case 1:** The current total medium size as well as the exact size of a parent of the field is known - E.g. for ID3v23 text frames, the frame size and thus payload size is known, but there is a terminated text payload field; for file and byte array media, the medium length is also known.
- **Case 2:** The current total medium size is unknown, yet the exact size of a parent of the field is known - Take an ID3v23 text frame on an input stream medium as an example.
- **Case 3:** The current total medium size is known, but the exact size of a parent of the field is unknown - Take an APEv2 item header on a file or byte array medium as an example. Here, the item key is terminated, and the size of the parent header is unknown as it is only determined by the sum of the field sizes and not - in addition - by any size indicators.
- **Case 4:** Neither current total medium size nor the exact size of a parent of the field is known - So APEv2 item header on an input stream medium is an example.

***DES 184:* Block-wise reading for finding field termination**
Block-wise reading is necessary in order to find field termination. According to DES 184, we do not read more than the configured maximum read-write block size in bytes from the medium at once. Actually, we *always* read the configured maximum read-write block size and do not take into consideration any additionally known upper bounds such as the remaining parent byte count or remaining medium length (if available). If an end of medium is encountered during reading a block of data, we scan for the termination within the actually read bytes. Otherwise the next block is read for scanning for termination bytes and so on. However, we can at least take into account any available bounds for finding out if we need to search further or we can skip the process: If the already read byte count is bigger than the remaining parent byte count, we should skip as a data corruption might be present.

Rationale: Block-wise reading avoids out-of-memory conditions and is a compromise between reading too much and reading too few bytes at once. Of course this compromise comes with the cost of added complexity. We mitigate this complexity by not coming up with further "optimizations" such as only reading up to a known upper bound like the medium end or parent end. In the end, both might not be known, especially in case 4, thus anyway we have to do a "blind" reading in this case.

Disadvantages: Added complexity, but in conformance with DES 184. The complexity is adressed as specified in the rationale.

Regarding strings and their encoding: First of all, we remember that only string and string list fields can be terminated and thus it is fine to talk about *termination characters* instead of termination bytes (see DES 184 and DES 184). How should we parse terminated fields to detect the termination character? We could do it either:

1. by converting the character to a byte sequence and then searching the termination

2. by converting the bytes to a string and then trying to find the termination character or

Both approaches have their serious drawbacks. **Approach (1)** requires us to know where a character starts and ends such that we can correctly match termination bytes only at character borders. For instance UTF-16 encodes all characters always as two bytes. It would be wrong to try to detect termination bytes at odd offsets. Instead we have to ensure to search for them only at even offsets.

Approach (2) suffers from a similar problem that there is no nice way to handle block-wise split strings: If we read block-wise, it might be that a single character spanning multiple bytes is just torn apart at a block border. E.g. UTF-8 is a variable length encoding where - depending on the first byte - the total number

of bytes of a character is given. If a converter from byte to string encounters a torn apart character at the end of the byte sequence, it might fail with some kind of decoding runtime error, so we have to handle this separately. However, Java's `CharsetDecoder` seems just about right for this task. ID3v2.4 supports UTF-16LE, UTF-16BE, ISO-8859-1 as well as UTF-8 so we have to handle such cases.

We define:

DES 185: **Character-based termination detection**
We use character-based termination detection (approach 2) for finding termination characters and do not use termination-byte-based detection (approach 1)

Rationale: This is possible due to the fact that only string and string list fields can be terminated (see DES 185 and DES 185). Approach 1 seems even harder as we need to re-implement parsing of strange encodings while approach 2 at least provides an out-of-the box solution in the form of `CharsetDecoder`.

Disadvantages: No disadvantages known

No we can summarize the process of finding termination of a field:

***DES 186:* Determining the end of a terminated field**
As input for the algorithm, we get the current medium offset, the character encoding of the bytes as well as the termination character to find and the maximum read-write block size. The steps are as follows:

1. Read $N :=$ read-write block size bytes from the medium at the current offset
2. If end of medium occurs, take the actually read bytes and mark the algorithm for termination as no further input is available
3. Search the input buffer for the termination character by actually decoding bytes according to the character encoding using `CharsetDecoder`
4. If there should be a character overlapping a block border, use the facilities of `CharsetDecoder` to overcome the situation
5. If a termination character is found, mark the algorithm for termination
6. Otherwise:
   a) If the remaining parent byte count is known and the already read byte count is bigger than this value, mark the algorithm for termination
   b) Otherwise increment the current medium offset by the maximum read-write block size and goto step 1

Rationale: This process is efficient in terms of avoiding unknown steps and using only sparse memory. It is as simple as it can get without risking incorrectness.

Disadvantages: No disadvantages known

A Need for Lazy Fields?

In this section, we want to investigate if we need a kind of lazy field (similar to lazy payload as described in DES 186). The idea: What if a field is large and we know this in advance? Should we read all field bytes into memory or rather not?

First of all, we shall check if there are cases that clearly come to mind here:

- Ogg contains segments within its payload that only get a max size of 255, so no need here for lazy fields; also the ogg header is quite harmless
- APEv2, Lyrics3v2, ID3v1 and ID3v2 only define small fields
- MP3 payload fields might not be longer than 998 bytes, so no lazyness required here
- RIFF, QuickTime and Matroska might have arbitrary large payload, and thus arbitrary large fields to read

It looks like that it is unavoidable that large fields might occur. However, there is still an easy way to circumvent this and define a maximum size, and no field lazyness:

***DES 187:* Fields must only have a maximum size of `Integer.MAX_VALUE`, no lazy field mechanism**
There is no special case for large fields in `jMeta`. Instead, we simply define that the maximum length of a field must not exceed `Integer.MAX_VALUE`. This is already checked during specification validation. If a need arises to support unstructured payload longer than this, the specification can simply define a field with maximum size `Integer.MAX_VALUE` and more than one occurrence.

Rationale: Avoids unnecessary complexity due to yet another special case.

Disadvantages: In theory, if fields actually reach `Integer.MAX_VALUE` and the user actually reads the field, an OOM might occur.

11.3.5. Backward Reading

The feature of reading backward is just a convenience and performance improvement functionality. It has a lot of differences compared to forward reading, i.e. it is not only just "reversed forward reading". The reason is that APIs do not know some kind of "backward read mode", i.e. file I/O and array APIs only allow to read forward starting at offset x. The problem with this is that you do not know where x is during backward reading, because e.g. x is the starting point of a dynamic length data block, but you are currently positioned at its end and have no idea where it starts.

Nevertheless, some data formats support backward reading by providing a so-called footer. The reason for this feature is that multimedia files are easier to modify at their end than at the beginning: Wen inserting a tag of length n at the beginning of the file, you actually have to read all bytes of the file, write them to offsets $x + n$, and then write the tag bytes at offset 0. This requires a lot of I/O and can be expensive, especially for large files. Appending tags at the end of file avoids this. To ensure that parsers can identify such end-of-file tags, footers have to be defined and present. From the supported formats, Lyrics3v2, ID3v2.4 and APEv2 support footers. ID3v1 can be read backwards, too, as it is of static length.

However, first of all, we have to note down:

***DES 188:* Streaming media cannot be read backwards**
`jMeta` cannot read stream-based media backwards and reacts with a runtime exception if a user tries to do that

Rationale: Random access media allow starting reading from any point, streams by design do only start at the beginning.

Disadvantages: No disadvantages known

If we do backward reading, the first question that might come up is: How do we actually do it? Quite similar to forward reading but with some important differences:

***DES 189:* Backward reading uses footer magic keys**
When backward reading, `DataBlocks` tries to find a footer magic key at a fixed position in front of the current offset. If it can find it there, this is a signal that a container of the given data format is present. The implementation tries to extract a static payload size from the footer(s). Then it reads the payload using this static size by going that much bytes backward. Finally, it reads any header(s) in front. If start of medium is encountered during this, this is quite similar to an end of medium unexpected error.

Rationale: This allows to reuse some parts of forward-reading and thus avoids to invent something totally different.

Disadvantages: No disadvantages known

DES 189 did mention a basic approach and also footer(s) and header(s). But how to deal with dynamic length headers or footers or unknown length payload? The problem here is that you only know where a footer, payload or header ends, but not where it actually starts if it is dynamic or unknown length. For now, we state:

DES 190: **Only static length headers, payload and footers are supported for backward-reading**
As of the current release, `jMeta` cannot deal with dynamic length headers or footers during backward reading. It throws a runtime exception if the presence of such a dynamic-length header is detected. The same is done if the lenght of payload could not be determined. However, `jMeta` fully supports multiple static-length headers or footers as well as optional ones, with quite the same mechanisms as for forward-reading.

Rationale: This is e.g. the case for ID3v2.4: If there is an extended header (which has dynamic length) there is no way to find out where it starts during backward reading except you search for the actual main header starting point. This would somehow not exactly fit with the forward-read approach. There is an option to implement it later or for the extensions to overwrite the default implementation in a more flexible way.

Disadvantages: No disadvantages known

One observation that is important to grasp comes when you think about parsing the content of a container that has already been backward-read:

DES 191: **Payload content of a backward-read container is forward-read**
As soon as we have fully read a container backwards, we of cause know where its payload starts and ends. Thus, we can of course read the payload *forward* instead of backward. The same is of course true for headers and footers during backward-reading: As we know where they start, we wouldn't want to read their fields somehow backwards. Instead we simply reuse the existing code that forward-reads fields.

Rationale: We can reuse a lot of code for forward-reading instead of coming up with new code. Decreases complexity.

Disadvantages: No disadvantages known

Finally, we look at a special case of backward reading: ID3v1.

***DES 192:* ID3v1 can be backward-read as it is static size, this is done in the extension**
As ID3v1 has a static size of 128 bytes, we can always check if there is the header's magic key at -128 bytes from the current offset. If so, we've found the tag during backward reading. However, as ID3v1 does not have a footer but only a header, this special case has to be implemented in a custom implementation of the extension and not in the generic centralized code.

Rationale: ID3v1 frequently occurs at the end of a medium and thus it makes sense to support this.

Disadvantages: No disadvantages known

11.3.6. Basic Concept for Writing

Here, we care about the basics for writing. Writing data is an intgral part of the `DataBlocks` component. The question is: How can a user manipulate data? What is already clear is that we cannot write to stream-based media and the attempt will lead to a runtime exception (see "11.1.2 The public API of medium access").

What can be already noted is that we will of course use the features of `Media`:

***DES 193:* Using `Media` with two-staged write protocol**
`DataBlocks` uses the `Media` API to perform the writing. Thus, it also has to adhere to the two-stage write protocol as described in "11.1.1 Two-Stage Write Protocol". That means that first, changes are done as needed, and then the user needs to perform an actual flush of the changes in a second step. For read-only media, an exception is thrown only when trying to flush changes.

Rationale: Otherwise we needn't do the fuss in `Media`. Two-stage writing is sensible as usually one change is not done isolated but leads to a series of other changes. The user can safely do the changes and then "commit" at a sensible time, leading to even a better performance.

Disadvantages: No disadvantages known

However, if we do it like that, one question immediately arises: What does the user "see" after having done changes but not yet having flushed them? For instance, the user may add an extended header to an ID3v2.3 tag or remove a frame from it. If the user then iterates the frames, will the removed frame still be returned or not?

DES 194: **The user sees the dirty state of the containers when calling getter methods with just one exception (see next design decision)**
Once the user has done changes on a `MediumStore` using the writing API of `DataBlocks`, these changes will be visible and active in the reading API of `DataBlocks`, even if they are not yet flushed. Removed blocks won't be returned anymore, inserted blocks will be returned, and changed field values will also be returned. The user would not see these changes using another `MediumStore` to access the medium. However, this is not possible due to the locking concept of `Media`.

Rationale: This seems to entirely contradict with DES 194 that states: `Media` read methods always return the currently persisted state. However, the written data is still available in terms of added data blocks.
It would be quite strange for the user if data blocks previously inserted would not be returned again when reiterating the blocks. This could cause confusion for the users and would e.g. not offer the possibility to undo changes if the user did not keep a reference to an insterted block.

Disadvantages: No disadvantages known

Here comes the sole exception:

DES 195: **Retrieval of raw bytes always only returns currently flushed state**
Retrieval of raw bytes always only returns currently flushed state. That means it will still return bytes for currently removed data blocks, but it won't return bytes for inserted or new data blocks.

Rationale: Read methods in `Media` only return the medium state (see DES 195), because doing things otherwise would have made it all even more complex. Likewise, also here, instead of building castles in the sky just for the rare case that a user might find it convenient to get the bytes according to the not-yet-persisted dirty state, we won't waste much time. Javadocs could state this behaviour easily.

Disadvantages: No disadvantages known

In section "11.2.10 Field Functions" we introduced the so-called field functions as a way to indicate dependencies between fields and other data blocks. However, just modeling this dependency without using it in anyway is of course nonsense. The main driver for introducing this concept is consistency: If you have a size field indicating the size of payload, and if that payload changes, we want the size field to change consistently without manual intervention of the user. Thus, we have to implement this feature during writing. This goes hand in hand with the two-stage write protocol decided in DES 195.

***DES 196:* Fields with functions are consistently updated during writing**
Whenever a user changes a datablock that is referenced by another field with a field function, the field's value must be updated according to the kind of change: `SizeOf`, `SummedSizeOf`, `CountOf` and `PresenceOf` are updated once a user writes to data blocks these field functions refer to. The fields themselves need to be read-only to ensure consistency. On the other hand, `CharacterEncodingOf` and `ByteOrderOf` fields can be written, leading to an inverse change to all data blocks they refer to: If you change the fields value, the byte order or character encodings of the target data blocks must change automatically, leading to a potential change to any numeric or string target fields. The two-stage write protocol of DES 196 minimizes the chance that there is an inconsistent state where the field value changes but not the referred data blocks, because the change is first getting only scheduled consistently, and later written in a "single" operation.

Rationale: Consistency and user convenience

Disadvantages: Of course this makes the implementation more complex and in the first place requires the concept of field functions.

A second inference from using a two-stage write protocol is the possibility of a "rollback" - which MUST not be confused with ACID in any way:

***DES 197:* Undo possibilities of unflushed changes are necessary**
It needs to be necessary that the user can undo all changes he has done before a flush. However, it must be clearly communicated in the API and documentation that this whole thing is in no way ACID! Thus we do not actually call it "rollback". Flushing changes can go wrong in the middle, leaving part of the changes already written to the medium. Likewise, "rollback" could fail in between, but will at least not messing up the medium, as nothing has been written yet. See DES 197 for further details.

Rationale: If having a two-stage write protocol, this desire comes to mind. If I can mark something for change, why can I not undo it? The second thing is field functions (DES 197): If we change fields according to other changes, then being able to undo these connected changes also in a consistent way is necessary. Another argument is that we have some possibility to do that in the undo functionalities of `Media`, see DES 197.

Disadvantages: Implementing this of course requires some effort: You need to ensure that every change is properly undone.

Having talked about consistency, it comes to mind: Whenever a user changes something or creates something new, there are a myriad of possibilities why this change is utterly against the data formatt's specification. A field value could be

invalid, the size of something could be too big or small, the type or structure of the thing to add could be tried to be added at the complete wrong place etc. How do we deal with such things?

***DES 198:* Strict validation of specification conformance as early as possible**
For every writing action, `DataBlocks` strictly checks the validity of the things done against the data format specification. If the change violates properties of the data block changed or its parent, this is clearly and directly communicated with a runtime exception. The details are defined in later individual design decisions.

Rationale: Fail fast is a good principle, here even more. We want to avoid the user to write any crap to the medium that cannot be re-read, as it might be utter nonsense.

Disadvantages: No disadvantages known

Already now, it clearly emerges that the writing API will be fundamentally different from the reading API:

***DES 199:* Writing methods are scattered in the API in contrast to reading**
There is a fundamental asymmetry between reading and writing: While reading is done using an iterator approach and a core class that does the reading, writing is done by methods associated with the datablocks themselves. For extensibility this means that there is no single class the user can subclass to extend the writing functionality.

Rationale: Reading is done step by step at once in a linear process, while writing means to manipulate the individual objects that have previously been read or created.

Disadvantages: Writing is harder to extend, but currently there seems to be no need to do so

11.3.7. The State of a Datablock

If `jMeta` would only support reading of datablocks, a datablock state would probably not be necessary. However, if datablocks are modified and as we have a two-stage write protocol, states become indeed a necessity, as actions depend on the current state of a datablock. We can think of the following states:

State	Description

State	Description
NEW	The datablock has been newly created or persistently removed from the medium. That means it is not attached to a parent datblock and does not have an offset.
PERSISTED	The datablock is persisted on a medium and thus has an offset. There are no modifications in the datablock that are not yet flushed. This is the state of a datablock after reading.
INSERTED	The datablock has been newly inserted into the medium and attached to a parent, but this change was not yet flushed.
REMOVED	The datablock has been removed from the medium and detached from a parent, but this change was not yet flushed.
MODIFIED	The datablock has been modified, but this change was not yet flushed.

Table 11.9.: Datablock states

***DES 200:* A datablock in jMeta has one of the states listed in table 11.9.**
A datablock in jMeta has one of the states listed in table 11.9.

Rationale: Depending on the state, some actions are legal or illegal, or need to be performed differently. Further on, the user must have transparency what has been done with a block so far, so the state is a good indication. Other states than the ones mentioned are not required. A separate MODIFIED is needed due to fields whose values can be changed.

Disadvantages: No disadvantages known

The questions remaining now are: How do states exactly change? And what about the hierarchical dependencies between datablocks? The latter question needs to be investigated first. What if a parent of a datablock is removed, does state of the child change correspondingly?

***DES 201:* If a datablock changes state, all its descendant blocks change to the same state**
Any change to a datablock leads to the same state change in all its children and further descendants.

Rationale: Obviously, all descendants are contained within the bounds of the datablock. So if it becomes `NEW`, `PERSISTED`, `INSERTED`, `REMOVED` or `MODIFIED`, all its descendants also do.

Disadvantages: No disadvantages known

What about a change in a child or other descendant, does it have any effect on its parent?

***DES 202:* Modification, insertion or removal of a descendant leads to all its parents transitioning to `MODIFIED` state**
A datablock changes into the `MODIFIED` state once any of its descendants are modified, inserted or removed.

Rationale: A change in any child leads to an actual change of data contained in the datablock. It would be misleading if the API would still say the datablock is `PERSISTED`, no matter there are unflushed changes in a child. Furthermore, implementing things such as flushing or undo will probably be easier if you can tell which top-level data blocks were modified in some way instead of needing to search all children until the leaf nodes of the hierarchy to find out.

Disadvantages: No disadvantages known

Now we have all ingredients to come up with a suitable state transition diagram, without actual method calls yet, see figure 11.3.

11.3.8. Writing Actions

In this section, we clarify in detail which write actions there are and how they need to work.

Creating New Data blocks

First of all, in order to insert new data blocks, the user must be able to create new data blocks. The natural place to go is a factory:

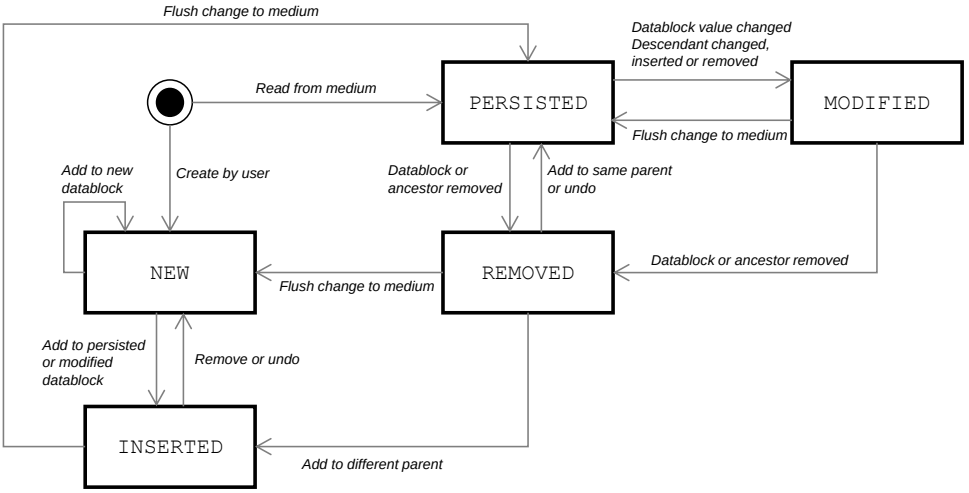

Figure 11.3.: Datablock state transitions

***DES 203:* A factory to create new data blocks**
A factory instance accessible by the user is required to create new instances of containers, fields, headers, footers or payload. The created data blocks have the state `NEW`. The user specifies the id of the data block to create. The factory ensures that the given id belongs to the expected data format and has the correct type. Further validations are only done during insertion only. Of course for the user being able to specify an id during creation, every id must be publicly available in the API of the data format extension.

Rationale: A factory is a class that can create a family of products. This is clearly the case for data blocks. Different implementations can be created in future, if necessary. There is no direct dependency from user code to concrete data block classes. Some validations are done here already according to DES 203.

Disadvantages: No disadvantages known

Another question to be answered is about the lifecycle of the factory. Should it be stateless and singleton or is there any use in having multiple instances?

***DES 204:* One factory instance per medium**
There is exactly one instance of the factory per medium and reader.

Rationale: The only reason for this is that thus the create methods may have less parameters, as we can pass reader, specification and `MediumStore` to the factory via constructor instead of passing them every time.

Disadvantages: No disadvantages known

Inserting Datablocks

NEW datablocks are inserted in their parent or at the top-level of the medium, respectively:

DES 205: **Insert methods for headers, footers and fields in their parent data block**
To insert new headers, footers or fields, there are corresponding insert methods that do the job in the corresponding datablock class. They take an index that must be valid.

Rationale: You anyway need to have the parent instance, thus it makes sense to perform insertion in the parent itself, as the children are also anyway part of the parent.

Disadvantages: No disadvantages known

DES 206: **Insert methods for containers in container iterators**
For both top-level as well as payload container iterators, there is an insert method in the iterator inserting a new container before the next container. That means to insert at a specific position, users first need to iterate to it. Containers can be inserted/appended at the end of the medium as soon as hasNext returns false.

Rationale: This is quite similar to the insertion of headers, footers and fields.

Disadvantages: No disadvantages known

Now what about payload datablocks? For these insert methods wouldn't make much sense. Thus we define:

DES 207: **Simple setter method for payload datablocks**
Payload children can be set with a setter method

Rationale: There is always just exactly one payload in a container

Disadvantages: No disadvantages known

One thing for parent datablocks containing lists of children: Can a user modify the returned lists?

***DES 208:* Lists of headers, footers and fields are unmodifiable for the user, but are internally updated by the insert methods**
A user can retrieve the child headers, footers or fields as a list. However, it is prevented that the user adds new children or otherwise modifies theses lists. Insert methods take care to also insert the child into the internal lists, if necessary.

Rationale: This way, there is no way in messing up the internal lists, and it is quite clear that the insert methods are the intended way to go.

Disadvantages: No disadvantages known

In DES 208 it was said that specification validity should be ensured as early as possible, for insertion that means:

***DES 209:* Specification validations during insertion**
The following checks against the specification are performed when inserting a new child, and if these checks fail, a runtime exception is thrown:

- Is the datablock id of the inserted block indeed a child of this datablock?
- Can it be inserted at the given position, i.e. is it specified to be located there?
- Is there already a datablock of the same type, and maximum occurrences are already reached?
- Is the parent read-only?
- Is the maximum parent size exceeded with this insertion (e.g. is the parent fixed size)?
- Is the inserted child of the expected datablock type, e.g. indeed a header, footer or field as indicated by the insert method called?
- Is the inserted child complete in a sense: Does it have all mandatory children?

Rationale: In accordance to DES 209

Disadvantages: No disadvantages known

If all these checks are done, further validations regarding the parent and child state are necessary:

***DES 210:* Required datablock states**
The datablock to insert must have the state `NEW`. The parent datablock must have one of the states `NEW`, `PERSISTED`, `MODIFIED` or `INSERTED`, i.e. it must not be `REMOVED`. If one of these conditions is not met, a runtime exception is thrown.

Rationale: Allowing other states than `NEW` for the to-be-inserted block would lead to strange use cases such as "remounting" an already persisted datablocks which we want to avoid. It is thus clear to the user: If I want to insert a data block, I have to newly create it or at least remove and flush it before. The parent state must not be `REMOVED`, because what sense would it make to add new children to a parent block which is entirely removed anyway on next flush?

Disadvantages: No disadvantages known

The next question regarding states is: How do states change after the insert?

***DES 211:* State changes caused by an insert**
If the parent block is `NEW`, the inserted datablock remains `NEW`. Otherwise the inserted block and all its descendants are changed into state `INSERTED` while the parent becomes `MODIFIED`, if not yet in that state.

Rationale: The states thus are quite conceivable.

Disadvantages: No disadvantages known

As described in DES 211, we need to adress fields with field functions during insertion. How this is done is described in the following design decision:

***DES 212:* Changes in field functions fields during inserts**
If a datablock is inserted into a parent of any state, it needs to be checked which fields refer to the size, count or presence of the block itself or one of its ancestors. For each affected field function field, the following is done:

- `SizeOf` and `SummedSizeOf`: Add the newly inserted block's size to the current interpreted value of the field
- `PresenceOf`: Set the flag indicating the presence to the value indicated by the field function
- `CountOf`: Increment the current interpreted value of the field by 1

Rationale: Required by DES 212

Disadvantages: No disadvantages known

Finally, we need to think about how the insertion of bytes really happens. First of all, let us think about what we need to achieve after the insert:

- Each inserted data block (i.e. the actually inserted one and all its descendants) needs to have an associated `MediumOffset` which is the insertion offset for all of them at insert time, and gets updated to its final offset during a flush.
- One or several inserts of concrete bytes must be scheduled in `Media` in correct order.

One could come up with two distinct strategies to achieve these goals:

Option 1: Insert bottom-up (fine-grained insertions) - Only fields as leafs of the hierarchy are actually inserted one by one in correct order

Option 2: Insert top-down (coarse-grained insertions) - At the level of insertion (container, header, field etc.), all leaf fields are collected and their binary representations are merged to form larger chunks which are then inserted as needed.

We can come up with pros and cons for both approaches:

	Advantages	**Disadvantages**
Option 1: Insert bottom- up (fine-grained insertions)	+ Field offsets after flush need not be calculated, as DES 212 takes effect	− Offsets for parent data blocks need to be determined and assigned
	+ Merging of data and potential memory doubling not necessary	− Probably imperformant writing, as a lot of small byte sequences (field-by-field) are written
Option 2: Insert top- down (coarse-grained insertions)	+ Better write performance as we can write large chunks instead of only small ones	− Offsets of child data blocks need to be determined and assigned
		− Complex merging requiring tree traversal and most probably doubling of memory and copying of bytes

Table 11.10.: Pros and cons of two insertion approaches

This already indicates we have a winner:

***DES 213:* Insertion uses Medias insert scheduling, done for each field in field order**
We choose **Option 1: Insert bottom-up (fine-grained insertions)**: When inserting a datablock into a parent, for each field contained in the data block (or for itself, if it is a field), its binary value is inserted at the insertion offset using Medias `insertData` method (see section "11.1.2 The public API of medium access"). Thereby the fields are visited in their specification order. The `MediumAction` instance returned by `insertData` is stored in each field.

Rationale: There is no other reason for the existence of insert scheduling. Why do we persist only fields and no other types of datablocks? Clearly, a container and every other data block in the end consists of fields in a defined order as the leafs of the data block hierarchy. The fields contain interpreted values that can be converted to binary values. These binary values are then ultimately persisted on the medium. Determine parent offsets is much easier than determining child offsets as a parent data block always starts at the same offset as its first leaf field. If necessary, `Media`'s flush can be changed to ensure small consecutive write actions are not written one by one but rather merged (up to reaching maximum read write block size) before being written.
Doing it in field order is clearly indicated by DES 213, as the insert offset for all new fields will be the same, their order must be clearly indicated by the call order of `insertData`. Doing it in another order thus would mess up the structure of the datablock, as fields are not persisted in the correct order as indicated in the data format specification.

Disadvantages: See table 11.10, disadvantes of option 1, but note the mitigations we noted down in the rationale.

A follow-up question is: Should we allow additional inserts in the middle of an already inserted data block? E.g. imagine you have inserted a container, but now after insertion you want to add another optional header to it. Is it allowed?

***DES 214:* Insert into an already `INSERTED` ancestor is rejected**
It is not allowed to insert a new data block into an already `INSERTED` ancestor.

Rationale: This reduces complexity, as otherwise one would need to first undo all inserts of fields that come behind the newly inserted child, and then reinsert to ensure they have the right order after a flush (compare DES 214).

Disadvantages: Of course this is less convenient for the user, as he must ensure that such "late" insertions do not happen, but he only inserts fully assembled data blocks before a flush.

At first glance, it seems utterly inlikely that a user of `jMeta` would want to insert a data block larger than int size (i.e. 2.15 gigabytes). However, a data block may have theoretically long size. Thus, however unlikely this might be, we

have to cover this case:

***DES 215:* Inserting a data block longer than max int size by multiple inserts in `Media`**
Inserting a data block being longer than maximum integer size is done by issuing multiple inserts in `Media`. The datablock thus has to keep multiple `MediumActions`.

Rationale: All `Media` methods taking a size are using int as data type to ensure only single `ByteBuffers` need to be handled, see section "11.1.2 The public API of medium access". And as our data blocks could have max long size, we need to cover this case, too. Anyway, it might not be unlikely anymore that a user comes with huge data blocks, as RAMs also increase year by year.

Disadvantages: We have to manage multiple `MediumActions` in a data block, probably leading to more complexity for undo operations and the like.

Removing Data Blocks

Removal of data blocks can be designed in a very similar way to insertion.

`PERSISTED`, `MODIFIED`, `INSERTED` or `NEW` datablocks are removed in their parent or at the top-level of the medium, respectively. Here, one could argue that also having a remove method on the child itself could be valid, as a data block has at most one parent. But we go for the same approach as for insertion:

***DES 216:* Remove methods for headers, footers and fields in their parent data block**
To remove headers, footers or fields, there are corresponding insert methods that do the job in the corresponding datablock class. They take an index that must be valid.

Rationale: We are in line with insertion design and this is the way of least surprise. Having a remove method in the child would only confuse API users. Anyway, there are not so much commonalitities that a single method would make sense, i.e. removing of a header might differ from removal of a field.

Disadvantages: No disadvantages known

DES 217: **Remove methods for containers in container iterators**
For both top-level as well as payload container iterators, there is a remove method in the iterator removing a container before the next container. That means to remove at a specific position, users first need to iterate to it. Likewise, the iterator's remove method can be designed to also the container remove method.

Rationale: This is quite similar to the removal of headers, footers and fields.

Disadvantages: No disadvantages known

Payload is simply a mandatory datablock and thus cannot be removed:

DES 218: **No remove method for payload datablocks**
Payload children cannot be removed

Rationale: There is always just exactly one payload in a container and it is mandatory.

Disadvantages: No disadvantages known

For the lists of children in parents, the very same design decision holds true as for insertions, see DES 218, meaning that remove methods manage also removing the child from the internal lists, if necessary.

In DES 218 it was said that specification validity should be ensured as early as possible, for removals that means:

DES 219: **Specification validations during removal**
The following checks against the specification are performed when removing a child, and if these checks fail, a runtime exception is thrown:

- Is the parent read-only?
- Is the minimum parent size reached with this removal (e.g. is the parent fixed size)?
- Is the minimum number of occurrences reached, e.g. is the data block present just once and not optional?

Rationale: In accordance to DES 219

Disadvantages: No disadvantages known

If all these checks are done, further validations regarding the parent and child state are necessary:

***DES 220:* Required datablock states**
The datablock to remove as well as its parent must have one of the states `NEW`, `PERSISTED`, `MODIFIED` or `INSERTED`. That means neither the block to remove nor its parent must have been already removed. If one of these conditions is not met, a runtime exception is thrown.

Rationale: For all of these states, it would be a valid need for a user to remove a block. We could even be tolerant in accepting removes for already removed blocks.

Disadvantages: No disadvantages known

The next question regarding states is: How do states change after the remove?

***DES 221:* State changes caused by a removal**
If the parent block or the to-be-removed block is `NEW`, it remains `NEW`. If the removed block is `INSERTED`, it becomes `NEW`. Otherwise the removed block and all its descendants are changed into state `REMOVED`. The parent becomes `MODIFIED`, if it was `PERSISTED` before, otherwise it remains in its previous state.

Rationale: The states thus are quite conceivable.

Disadvantages: No disadvantages known

As described in DES 221, we need to adress fields with field functions during removal. How this is done is described in the following design decision:

***DES 222:* Changes in field functions fields during removals**
If a datablock is removed from a parent of any state, it needs to be checked which fields refer to the size, count or presence of the block itself or one of its ancestors. For each affected field function field, the following is done:

- `SizeOf` and `SummedSizeOf`: Subtract the removed block's size from the current interpreted value of the field
- `PresenceOf`: Set the flag indicating the absence to the value indicated by the field function
- `CountOf`: Decrement the current interpreted value of the field by 1

Rationale: Required by DES 222

Disadvantages: No disadvantages known

Finally, we need to think about how the removal of bytes really happens. For insertions, we said that it would be best to have one low-level `Media` insert for

each inserted field as these contain the bytes. For removals, we do it the other way round:

***DES 223:* Remove uses `Medias` remove scheduling, done for the actually removed data block**
A remove is scheduled for the data block that is removed over its whole size.

Rationale: In contrast to insertions, we do not need to state which bytes to remove by having these, but you just can say *how much* bytes to remove (see section "11.1.2 The public API of medium access"). It is more efficient to issue just one such remove instead of hierarchically traversing all fields and issueing separate removes.

Disadvantages: No disadvantages known

Removes of children that are already removed are not possible (see DES 223). However, at least it could happen that a user first removes a child, then the parent. This is allowed:

***DES 224:* Removing an ancestor of an already `REMOVED` datablock is allowed**
It is allowed to remove an ancestor of an already `REMOVED` block.

Rationale: Removing a region from the medium fully encompassing an already removed region is also an allowed use case in `Media` and leads to the first remove being cancelled (see DES 224).

Disadvantages: No disadvantages known

Quite in accordance with insertion (DES 224), we have to handle the case of removal of data blocks larger than int size:

***DES 225:* Removing a data block longer than max int size by multiple removes in `Media`**
Removing a data block being longer than maximum integer size is done by issuing multiple removes in `Media`. The datablock thus has to keep multiple `MediumActions`.

Rationale: All `Media` methods taking a size are using int as data type to ensure only single `ByteBuffers` need to be handled, see section "11.1.2 The public API of medium access". And as our data blocks could have max long size, we need to cover this case, too. Anyway, it might not be unlikely anymore that a user comes with huge data blocks, as RAMs also increase year by year.

Disadvantages: We have to manage multiple `MediumActions` in a data block, probably leading to more complexity for undo operations and the like.

Modifying Datablocks

Modifying data blocks mostly means modifying the value of a field. Clearly this is because all other data blocks are consisting of children, and these can be modified. Here we define the conditions for modifying field values.

For modification, the user can use the following:

DES 226: **Setter for setting the interpreted as well as binary value of a field**
For fields, there is a setter for the binary value to directly set the bytes of the field as well as a setter for the interpreted value, i.e. "human-readable" value. Calling the interpreted value setter will immediately transform the value to bytes and vice versa.

Rationale: This is quite fair to allow both representations to be changed.

Disadvantages: No disadvantages known

Now let us clarify the state handling of modifications of fields:

DES 227: **Fields in all states but `REMOVED` can be modified**
The only state rejecting modification with a runtime exception is `REMOVED`.

Rationale: Modifying a field's value that is anyway getting removed makes no sense, and would lead to runtime exceptions because of DES 227.

Disadvantages: No disadvantages known

In line with that, let's cover the state changes:

DES 228: **State changes caused by a modification**
If the parent block or the to-be-modified field is `NEW`, `MODIFIED` or `INSERTED`, both parent and field remain so. If the modified field and its parent are in state `PERSISTED`, both change into `MODIFIED` state.

Rationale: The states thus are quite conceivable.

Disadvantages: No disadvantages known

Field value changes of course also underly validation against specification:

***DES 229:* Specification validations during field modification**
The following checks against the specification are performed when modifying a field, and if these checks fail, a runtime exception is thrown:

- Is the field read-only?
- Is the minimum or maximum field size reached with this change (e.g. is it fixed size)?
- Is the field value one of the allowed enumerated values?
- Can the field's interpreted value be converted to binary (and vice versa) without error?

Rationale: In accordance to DES 229

Disadvantages: No disadvantages known

Quite the same way as for insertions and removals, we need to adress fields with field functions during modifications of a field. However, this has two aspects to it:

***DES 230:* Changes in field functions fields during field modification**
If a field is modified and its size changes by this modification, it needs to be checked which fields refer to the size of the field itself or one of its ancestors. For each affected field function field, the following is done for `SizeOf` and `SummedSizeOf`: Subtract or add the field size change from the current interpreted value of the field.

Rationale: Required by DES 230

Disadvantages: No disadvantages known

***DES 231:* `CharacterEncodingOf` and `ByteOrderOf` fields lead to inverse changes**
If a field defines a `CharacterEncodingOf` or `ByteOrderOf` field function, it can be modified itself to reflect a different character encoding or byte order. Of course, to ensure consistency of the data according to DES 231, the affected target blocks might need to be changed then: They need to be reformatted to match the newly set character encoding or byte order.

Rationale: See DES 231

Disadvantages: No disadvantages known

Finally, we have to think about how modifications of fields are brought to the

medium:

DES 232: **Field modifications use `Medias` replace scheduling**
Changing the value of a field schedules a replace in `Media` as long as the field is not `NEW` or `INSERTED`.

Rationale: Clearly the way to go as `PERSISTED` and `MODIFIED` fields clearly already are present on the medium, and thus modifying their value means to replace their previous value with something new. This also easily covers subsequent modificaitons of the same field: Simply issue the same replace multiple times, always specifying the old size as to be replaced and the new bytes (see DES 232).

Disadvantages: No disadvantages known

For `NEW` and `INSERTED` fields, it holds:

DES 233: **For fields still `NEW`, no replace is scheduled yet, for `INSERTED` fields, another insert is scheduled**
If a `NEW` field is modified, it means it anyway hasn't yet a scheduled `MediumAction`, so we also won't be scheduling any replace when modifying the field's value. For `INSERTED` fields, it holds that there is already a scheduled insert in accordance to DES 233. This `MediumAction` needs to be undone, and instead a new insert with the new data needs to be scheduled.

Rationale: These two states are different than the others and thus require the special handling.

Disadvantages: No disadvantages known

As unlikely as it seems now, again we have to handle the case of field modifications where the new field size exceeds integer maximum value:

***DES 234:* Modifying a field to longer than max int size by multiple replaces in `Media`**
Modifying a field being longer than maximum integer size is done by issuing multiple replaces in `Media`. The datablock thus has to keep multiple `MediumActions`.

Rationale: All `Media` methods taking a size are using int as data type to ensure only single `ByteBuffers` need to be handled, see section "11.1.2 The public API of medium access". And as our data blocks could have max long size, we need to cover this case, too. Anyway, it might not be unlikely anymore that a user comes with huge data blocks, as RAMs also increase year by year.

Disadvantages: We have to manage multiple `MediumActions` in a data block, probably leading to more complexity for undo operations and the like.

Flushing

Flushing all changes is unfortunately not exactly done by just calling flush of `Media`.

First of all, we have to think of where the flush actually happens:

***DES 235:* Writing of all changes done in top-level container iterator**
The top-level container iterator offers a method to flush all previously made changes onto the medium. It is clear that this operation is not ACID as already pointed out in DES 235.

Rationale: The top-level container iterator is the starting point of working with a medium and the logical place to have the flushing method(s).

Disadvantages: No disadvantages known

Flushing in `DataBlocks` needs to do the following:

***DES 236:* Steps of flushing**
First of all, we call `MediumStore.flush()` to actually write all changes to the medium. Second, we set the states of all previously `INSERTED` or `MODIFIED` data blocks to `PERSISTED` again, and the states of all previously `REMOVED` data blocks to `NEW`.

Rationale: Flushing the store actually invalidates all `MediumActions` (see DES 236) and finally determins the actual `MediumOffsets` of each data block (see DES 236). Also cleaning up the states is clearly indicated. The order of operations is as such to at least have some basic sanity in case of failure: If flushing fails in the middle, i.e. some changes have been written and others not, at least the `MediumAction` and `MediumOffsets` up to the point of failure are valid and these updates are present in the data blocks. Only their states would not be correct. Even "retrying" would then probably be possible with just a few lines of special handling.

Disadvantages: No disadvantages known

Undo all Changes

According to DES 236, it has to be possible to undo changes done previously using insertions, removals or modifications. This especially becomes interesting if multiple changes are done before a flush.

First of all, we have to think of where the "undo all" actually happens:

***DES 237:* Undoing all changes is done in top-level container iterator**
The top-level container iterator offers a method to undo all previously made changes. It is clear that this operation is not ACID as already pointed out in DES 237.

Rationale: The top-level container iterator is the starting point of working with a medium and the logical place to have the undo all method(s).

Disadvantages: No disadvantages known

Now we not only need an undo all, but also the possibility to undo single changes with the following rules:

DES 238: **Undoing changes for `MODIFIED` and `INSERTED` data blocks**
For `MODIFIED` and `INSERTED` data blocks, there is an undo operation that undoes the change in `Media` for fields or traverses all field children for non-fields and calls undo. During this process, for all `MODIFIED` fields, the previous field's value is re-read from the medium to be restored in place. For all `INSERTED` fields, the values remain unchanged. For previously `MODIFIED` data blocks, their state is set to `PERSISTED`, for previously `INSERTED` data blocks, their state is set to `NEW`.

Rationale: We have to somehow restore the initial state of the fields when undoing.

Disadvantages: No disadvantages known

DES 239: **Undoing changes for `REMOVED` data blocks**
For `REMOVED` data blocks, the undo operation can only be called if the data blocks parent is not `REMOVED`, otherwise it fails. Then it undoes the `MediumAction` on the data block it is called and sets the states of all data blocks to `PERSISTED`.

Rationale: We have to somehow restore the initial state of the fields when undoing.

Disadvantages: No disadvantages known

However, a bigger complexity arises when assuming multiple changes done on the same data block before the removal. We first have to state clearly:

DES 240: **Undo is rather a reset as it does only reset to the *initial* state**
When undoing, the operation does not restore the *previous* state, but the *initial* state of the data block. E.g. first inserting a new child, then removing it: Undo does not restore the child on first call and remove it again on the second. So undo uses no history or stack of any kind to behave like undo in an editor, e.g.

Rationale: Otherwise we would need to keep tracke of a change history of some sort for each data block which would make things overly complicated. It is also not very likely that users actually need incremental undo functionalities. Rather, they just need the possibility to reset the initial state as read from the medium.

Disadvantages: No disadvantages known

Still, even if we now made our lives easier, there is one case that requires a re-read of fields again:

***DES 241:* Undo removal of fields that were previously modified**
If the user modifies a field previously read from the medium, the fields value (interpreted and binary) is changed to the new value. If the user than removes the field itself or its ancestor, it is set to `REMOVED`, still keeping the modified value. If the user now calls undo, the field would be `PERSISTED`, but with a value that does not match the fields actual value on the external medium. To mitigate this, whenever a `MODIFIED` field gets `REMOVED`, we re-read the fields actual value from the medium.

Rationale: This is the most efficient way to handle this situation. Another possibility would have been to "simple" re-read all field values from the medium once a user undoes a remove, but this is very inefficient.

Disadvantages: No disadvantages known

11.3.9. Datablock Event Handling

We can see that insertion, removal or modification of a data block lead to a lot of things happening:

- State changes in the data block itself
- State changes in its ancestors as well as descendants
- Field function field updates

In addition, we need some way to know which data blocks were updated to be able to either flush or undo them all at once.

These things can all be handled by a single measure:

***DES 242:* Event handling mechanism for processing data block changes**
Data block changes emit events that are propagated to any interested party, i.e. those being registered for events. This includes ancestors, descendants as well as the container context and the top-level container iterator.

Rationale: Changes do not only have local effect in the data block itself but need to cause changes at some other places, too. Instead of directly couple all those places together, we opt for a decoupling via event handling: A set of listeners registers to the event emitter and is notified if something changes. The listener decides if the event is of interest and processes it correspondingly. This decouples emitters and receivers way better, yet even allowing extending receivers to add new functionality in future.

Disadvantages: Slight increase of complexity by introducing corresponding methods and classes for event handling

One question we should answer is: How do the events look like, i.e. which data is contained? We can state that all events are somehow related to data blocks. Thus we define:

DES 243: **Class `DataBlockEvent` contains the event type and causing data block**
We introduce a new class `DataBlockEvent` that contains:

- **The event type:** One of `INSERTED` (data block has been inserted), `PERSISTED` (data block has been persisted due to a flush), `REMOVED` (data block has been removed), `MODIFIED` (data block has been modified), `RESET` (data block has been reset, i.e. undone), `FLUSHED` (all changes where flushed), `RESET_ALL` (all changes where reset, i.e. undone)
- **The event time stamp:** States the point in time the event occurred
- **The causing data block:** The data block that has caused the event, null for `FLUSHED` and `RESET_ALL`

Rationale: Except flush and reset, we only have events related to data blocks. The receiver needs to know who caused the event to do the right thing, as well as what exactly has happened with the block.

Disadvantages: No disadvantages known

Another question is: How are events exactly propagated?

DES 244: **Broadcast bus with all data blocks, container contexts and the top-level iterator as listeners**
A broadcast mechanism with a bus is used, all data blocks and container contexts as well as top-level iterators are registered as listeners. Every event causes a broadcast on the bus to all other receivers. They filter the event to decide if they need to do something and otherwise ignore it.

Rationale: We have a central point for registration of listeners, notification as well as monitoring. Broadcasts are fine because all of the events are relevant for multiple, sometimes even all listeners. It is also more flexible and easier to extend than rigid solutions.

Disadvantages: Some listeners are unnecessarily informed about an event they do not care about. Second, it is not clear how we ensure that events are not repeated or lead to too much follow-up events.

Finally, we shall ensure that an event does not lead to a flood of follow-up events:

***DES 245:* A state change in a data block alone is not enough for an event**
Whenever a child is `MODIFIED`, `INSERTED` or `REMOVED`, a corresponding event is sent that lead to corresponding changes of states also in ancestors and descendants. However, for these indirect changes, no additional event is sent to the bus.

Rationale: Avoids flooding of bus and listeners with secondary events and thus excludes the need for special case handling. Also, this might reduce the possibility of endless event loops.

Disadvantages: No disadvantages known

11.3.10. API Design

Operation	Description
`DataBlock .getId()`	Gets the `DataBlockId` of the `DataBlock`
`DataBlock .getSequenceNumber()`	Gets sequence number, which is the occurrence index in its parent
`DataBlock .getOffset()`	Returns the `MediumOffset` this `DataBlock` is currently located on the external medium or at least scheduled to be inserted at this position. Returns null if the `DataBlock` is still `NEW` or `INSERTED`.
`DataBlock .getSize()`	Returns the total size of the `DataBlock` in bytes, if currently known, or `DataBlockDescription.UNDEFINED` otherwise
`DataBlock .getState()`	Gets the current state of the `DataBlock`.
`DataBlock .getBytes(ofs, n)`	Gets the DataBlock's bytes starting at the given global offset with the given length. Both offset and length must be within the actual start offset plus total size range. The bytes are only returned if the `DataBlock` is in state `PERSISTED`, `REMOVED` or `MODIFIED`, and then taken from the external medium. For fields, this method also returns bytes for other states, which are the bytes actually set by the user.
`DataBlock .getContainerContext()`	Returns the `ContainerContext` of the next higher ancestor `Container` this `DataBlock` is associated with (either as it is the `Container` itself or its descendant). Returns null for `NEW` `DataBlock`s.
`DataBlock .getParent()`	Returns the direct parent of the `DataBlock` if it is not `NEW`. Otherwise it returns null. For top-level `DataBlock`s, this method also returns null.

Operation	Description
`DataBlock .initParent(DataBlock)`	Initializes the parent `DataBlock`, i.e. must only be called once.
`DataBlock .initContainerContext(ContainerContext)`	Initializes the `ContainerContext`, i.e. must only be called once.
`DataBlock .reset()`	Resets all changes within this `DataBlock` itself and all its descendends, i.e. resets any modifications, removals and inserts.

Table 11.11.: Operations of the `DataBlock` interface

Operation	Description
`ContainerIterator .hasNext()`	Tells whether there is a next `Container` available that can be retrieved using `next()`.
`ContainerIterator .next()`	Returns the next `Container` instance if available or throws a runtime exception if not. It thus also advances the iterator.
`ContainerIterator .remove()`, `ContainerIterator.removeContainer()`	Schedules the current `Container` for removal. If done before any calls to `next()`, this removes the first `Container`, if any. If there is no next `Container`, a runtime exception is thrown.
`ContainerIterator .insertContainer(Container)`	Inserts a `NEW Container` in front of the current `Container`. This method is allowed to be called even if `hasNext()` returns false in order to append a new `Container` at the end of the current payload or medium.

Table 11.12.: Operations of the `ContainerIterator` interface

Operation	Description
`TopLevelContainerIterator .close()`	Closes the use of the iterator for top-level container retrieval and writing. Once this is done, the instance must not be used anymore. It also closes the access to the underlying medium and cleans up any further resources.
`TopLevelContainerIterator .writeAllChanges()`	Flushes all previously made changes (removals, inserts and modifications) to the external medium.
`TopLevelContainerIterator .resetAllChanges()`	Resets all previously made changes (removals, inserts and modifications) as if they never happened.

Table 11.13.: Operations of the `TopLevelContainerIterator` interface

Operation	Description
`ContainerContext .getDataFormatSpecification()`	Returns the DataFormatSpecification this context belongs to.
`ContainerContext .getContainer()`	Returns the `Container` this context belongs to.
`ContainerContext .getParentContainerContext()`	Returns the parent `ContainerContext` or null if it refers to a top-level `Container`.
`ContainerContext .addFieldFunctions(Field)`	During reading: Adds all field functions of a given field (if any) to the `ContainerContext`
`ContainerContext .getSizeOf(DataBlockId, int)`	Tries to determine the size of the indicated `DataBlockId` with the given sequence number or `DataBlockDescription.UNDEFINED` if it cannot be determined.
`ContainerContext .getOccurrencesOf(DataBlockId)`	Tries to determine the number of occurrences of the indicated `DataBlockId` or `DataBlockDescription.UNDEFINED` if it cannot be determined.
`ContainerContext .getByteOrderOf(DataBlockId, int)`	Tries to determine the byte order of the indicated `DataBlockId` with the given sequence number or `null` if it cannot be determined.
`ContainerContext .getCharacterEncodingOf(DataBlockId, int)`	Tries to determine the character encoding of the indicated `DataBlockId` with the given sequence number or `null` if it cannot be determined.

Table 11.14.: Operations of the `ContainerContext` interface

Operation	Description
`Container .getHeaders()`	Returns the `Header`s of this `Container`. Unmodifiable list.
`Container .getPayload()`	Returns the `Payload` of this `Container`.
`Container .getFooters()`	Returns the `Footer`s of this `Container`. Unmodifiable list.
`Container .setPayload(Payload)`	Sets the `Payload` of this `Container`. Must be a `NEW Payload` that actually belongs to this `Container`. The previous `Payload` is replaced by the new one.
`Container .insertHeader(int, Header)`	Inserts a `NEW Header` into this `Container`. The given index must be valid. Must be a `NEW Header` that actually belongs to this `Container` and must not exceed the maximum occurrences of this header type.
`Container .insertFooter(int, Footer)`	Inserts a `NEW Footer` into this `Container`. The given index must be valid. Must be a `NEW Footer` that actually belongs to this `Container` and must not exceed the maximum occurrences of this header type.

Table 11.15.: Operations of the `Container` interface

Operation	Description
`FieldSequence .getFields()`	Returns the `Field`s of this `FieldSequence`. Unmodifiable list.
`FieldSequence .insertField(int, Field)`	Inserts a `NEW Field` into this `FieldSequence`. The given index must be valid. Must be a `NEW Field` that actually belongs to this `FieldSequence` and must not exceed the maximum occurrences of this header type.

Table 11.16.: Operations of the `FieldSequence` interface, thus also for `FieldBasedPayload`, `Header` and `Footer`

Operation	Description
`Field .getInterpretedValue()`	Returns the interpreted value of the `Field`.
`Field .getBinaryValue()`	Returns the binary value of the field as `ByteBuffer`.
`Field .setInterpretedValue(T)`	Sets a new interpreted value for the Field. Must be a valid value and must not exceed the field's maximum size.
`Field .setBinaryValue(ByteBuffer)`	Sets a new binary value for the field. Must be a valid value and must not exceed the field's maximum size.

Table 11.17.: Operations of the `Field` interface

Operation	Description
`ContainerBasedPayload .getContainerIterator()`	Returns the `ContainerIterator` for iterating the child containers.

Table 11.18.: Operations of the `ContainerBasedPayload` interface

Operation	Description
`SizeProvider .getSizeOf(DataBlockId, int, ContainerContext)`	Determins a custom size of the given `DataBlockId` and sequence number or `DataBlockDescription.UNDEFINED` if it cannot be determined.
`CountProvider .getCountOf(DataBlockId, ContainerContext)`	Determins a custom count of the given `DataBlockId` or `DataBlockDescription.UNDEFINED` if it cannot be determined.

Table 11.19.: Operations of the `CountProvider` and `SizeProvider` interfaces

Operation	Description
`DataBlockEventBus .registerListener(DataBlockEventListener)`	Registers a new `DataBlockEventListener`.
`DataBlockEventBus .publishEvent(DataBlockEvent)`	Notifies all currently registered `DataBlockEventListener`s that a new event has occurred. The order of notification is undefined and listeners must not rely on it.
`DataBlockEventListener .dataBlockEventOccurred(DataBlockEvent)`	Is called once whenever a new `DataBlockEvent` has occurred on the `DataBlockEventBus`.

Table 11.20.: Operations of the `DataBlockEventBus` and `DataBlockEventListener` interfaces

Operation	Description
`DataBlockFactory.createPersistedContainer(DataBlockId, int, MediumOffset, ContainerContext, DataBlockReader)`	
`DataBlockFactory.createPersistedHeader(DataBlockId, int, MediumOffset, ContainerContext, DataBlockReader)`	
`DataBlockFactory.createPersistedFooter(DataBlockId, int, MediumOffset, ContainerContext, DataBlockReader)`	
`DataBlockFactory.createPersistedPayload(DataBlockId, int, MediumOffset, ContainerContext, long, DataBlockReader)`	
`DataBlockFactory.createPersistedField(DataBlockId, int, MediumOffset, ContainerContext, ByteBuffer, DataBlockReader)`	
`DataBlockFactory.createNewContainer(DataBlockId)`	
`DataBlockFactory.createNewHeader(DataBlockId)`	
`DataBlockFactory.createNewFooter(DataBlockId)`	

Operation	Description
`DataBlockFactory.createNewPayload(DataBlockId)`	
`DataBlockFactory.createNewField(DataBlockId, T)`	

Table 11.21.: Operations of the `DataBlockFactory` interface

TODO: DataBlockAccessor + DataBlockService

Bibliography

[CFPPaper] An algorithm for writing scheduled modifications to finite byte sequences
Jens Ebert, December 2017
TODO
11.1.3

[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

[MC17] Metadata Compendium - Overview of Popular Digital Metadata Formats
Jens Ebert, January 2017
TODO
I, 5, 5.2, 8.1, 11.2.1, 11.2.2, 11.2.6

[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