```java
package at.jku.videocuttingtool.backend;

import java.io.*;
import java.nio.file.*;
import java.util.*;

import static java.util.stream.Collectors.joining;

/**
 * Backend of the Basic Video cutting tool
 */
public class Backend {
    private final List<File> files = new ArrayList<>();
    private File workingDir;

    private static final String sep = File.separator;
    private static final String ffmpeg = "lib" + sep + "ffmpeg" + sep + "ffmpeg.exe";

    /**
     * add sources that are to be used in the project
     *
     * @param sources a list of Files
     */
    public void addSources(List<File> sources) {
        files.addAll(sources);
    }

    /**
     * load all file from the working directory
     */
    public void loadFromWorkingDir() {
        List<File> files = new ArrayList<>();
        listf(workingDir, files);
        addSources(files);
    }

    /**
     * set the working directory
     *
     * @param dir the working directory
     */
    public void setWorkingDir(File dir) {
        this.workingDir = dir;
    }

    /**
     * method to export a given timeline with its export parameters
     *
     * @param export the export container with the given export parameters
     * @throws IOException          if an error occurs while cutting the clips
     * @throws InterruptedException if an error occurs while executing the ffmpeg commands
     */
```

```
 53     public void export(Export export) throws IOException,
    InterruptedException, EditMediaException {
 54         File exportDir = export.getExport().getParentFile();
 55 
 56         if(exportDir==null) {
 57             exportDir = workingDir;
 58         }
 59 
 60         if (!exportDir.isDirectory()) {
 61             return;
 62         }
 63 
 64         //generate temporary directories for the export
 65         File vDir = new File(exportDir + sep + "outV");
 66         vDir.mkdir();
 67         File aDir = new File(exportDir + sep + "outA");
 68         aDir.mkdir();
 69 
 70         String vF = export.getVideoFormat();
 71         String aF = export.getAudioFormat();
 72         File vMerged = null, aMerged = null;
 73         boolean hasVideo = false, hasAudio = false;
 74 
 75         /*
 76          * if there are more than 1 files in the video or audio timeline
 77          * merge them together
 78          */
 79         Process merge;
 80         if (export.getTimeline().getVideo().size() > 0) {
 81             if (export.getTimeline().getVideo().size() == 1) {
 82                 vMerged = export.getTimeline().getVideo().get(0).getMedia
    ();
 83             } else {
 84                 vMerged = new File(vDir + sep + "v_export." + vF);
 85                 merge = mergeMedia(export.getTimeline().getVideo(), vMerged
    );
 86                 if (merge != null && merge.waitFor() != 0) {
 87                     throw new EditMediaException(merge.getErrorStream());
 88                 }
 89             }
 90             hasVideo = true;
 91         }
 92 
 93         if (export.getTimeline().getAudio().size() > 0) {
 94             if (export.getTimeline().getAudio().size() == 1) {
 95                 aMerged = export.getTimeline().getAudio().get(0).getMedia
    ();
 96             } else {
 97                 aMerged = new File(aDir + sep + "a_export." + aF);
 98                 merge = mergeMedia(export.getTimeline().getAudio(), aMerged
    );
 99                 if (merge != null && merge.waitFor() != 0) {
100                     throw new EditMediaException(merge.getErrorStream());
101                 }
102             }
```

```
103             hasAudio = true;
104         }
105 
106         /*
107          * check if there is audio overlapping the video
108          * if there is merge the extra audio with the video and its audio
109          */
110         if (hasVideo && hasAudio){
111             String extractAudio = new Exec(ffmpeg)
112                     .addInput(vMerged.getAbsolutePath())
113                     .addArg("-y")
114                     .addOutput(aDir + sep + "v_audio." + aF)
115                     .done();
116             String mergeAudio = new Exec(ffmpeg)
117                     .addInput(aMerged.getAbsolutePath())
118                     .addInput(aDir + sep + "v_audio." + aF)
119                     .addArg("-filter_complex amerge -c:a libmp3lame -q:a
     4 -y")
120                     .addOutput(aDir + sep + "final_audio." + aF)
121                     .done();
122             String mergeAudioAndVideo = new Exec(ffmpeg)
123                     .addInput(vMerged.getAbsolutePath())
124                     .addInput(aDir + sep + "final_audio." + aF)
125                     .addArg("-map 0:v -map 1:a -c copy -y")
126                     .addOutput(export.getExport().getAbsolutePath())
127                     .done();
128 
129             Process p;
130             if ((p = Runtime.getRuntime().exec(extractAudio)).waitFor() !=
    0) {
131                 throw new EditMediaException(p.getErrorStream());
132             }
133             if ((p = Runtime.getRuntime().exec(mergeAudio)).waitFor() != 0
    ) {
134                 throw new EditMediaException(p.getErrorStream());
135             }
136             if ((p = Runtime.getRuntime().exec(mergeAudioAndVideo)).waitFor
    () != 0) {
137                 throw new EditMediaException(p.getErrorStream());
138             }
139         } else if (hasVideo) {
140             Files.copy(vMerged.toPath(),export.getExport().toPath(),
    StandardCopyOption.REPLACE_EXISTING);
141         } else if (hasAudio) {
142             Files.copy(aMerged.toPath(),export.getExport().toPath(),
    StandardCopyOption.REPLACE_EXISTING);
143         }
144 
145         delDir(vDir);
146         delDir(aDir);
147     }
148 
149     /**
150      * merge a set amount of media files
151      * can either be audio or video
```

```
152      *
153      * @param media a List of @see{@link Clip} Objects
154      * @param merge output directory+file for exported merged file
155      * @return a Process to be executed be the caller
156      * @throws IOException          if an error occurs while cutting the clips
157      * @throws InterruptedException if an error occurs while executing the ffmpeg commands
158      */
159     private Process mergeMedia(List<Clip> media, File merge) throws IOException, InterruptedException, EditMediaException {
160         File dir = merge.getParentFile();
161         String format = merge.getName().substring(merge.getName().lastIndexOf('.'));
162         if (!dir.isDirectory() || format.isEmpty()) {
163             return null;
164         }
165 
166         /*
167          * apply the start and end time if available to the clip
168          * -> cut the clip to size
169          */
170         File tmp = new File(dir + sep + "ffmpeg_merge.txt");
171         StringBuilder files = new StringBuilder();
172         for (Clip c : media) {
173             String partName = dir.getAbsolutePath() + sep + "part_" + c.getPos() + format;
174             Optional<Process> p = cutClip(c, partName);
175             if (p.isPresent()) {
176                 if (p.get().waitFor() != 0) {
177                     throw new EditMediaException(p.get().getErrorStream());
178                 }
179                 files.append("file '").append(partName).append("'\n");
180             } else {
181                 files.append("file '").append(c.getMedia().getAbsolutePath()).append("'\n");
182             }
183         }
184         Files.write(tmp.toPath(), files.toString().getBytes());
185 
186         String concat = new Exec(ffmpeg)
187                 .addArg("-f concat -safe 0")
188                 .addInput(tmp.getAbsolutePath())
189                 .addArg("-c copy -y")
190                 .addOutput(merge.getAbsolutePath())
191                 .done();
192 
193         return Runtime.getRuntime().exec(concat);
194     }
195 
196     /**
197      * @param clip   a single @see{@link Clip} object that should be cut
198      * @param export the complete filepath (incl. file location/filename) of the exported clip
```

```
199      * @return an Optional Process to be executed be the caller
200      * this is empty if no cut parameters for the clip are set
201      * @throws IOException if an error occurs while cutting the clips
202      */
203     private Optional<Process> cutClip(Clip clip, String export) throws
    IOException {
204         if (clip.getStart().isEmpty() && clip.getEnd().isEmpty()) {
205             return Optional.empty();
206         }
207 
208         Exec cut = new Exec(ffmpeg);
209 
210         if (!clip.getStart().isEmpty()) {
211             cut.addArg("-ss " + clip.getStart());
212         }
213 
214         cut.addInput(clip.getMedia().getAbsolutePath())
215                 .addArg("-c copy");
216 
217         if (!clip.getEnd().isEmpty()) {
218             cut.addArg("-to " + clip.getEnd());
219         }
220 
221         cut.addArg(" -y ");
222         cut.addOutput(export);
223 
224         return Optional.of(Runtime.getRuntime().exec(cut.done()));
225     }
226 
227     /**
228      * method to save a project
229      *
230      * @param tm the @see{@link Timeline} object to save
231      * @param to the file to save to
232      * @throws IOException if an error while writing to the file occurs
233      */
234     public void saveProject(Timeline tm, File to) throws IOException {
235         try (BufferedWriter out = new BufferedWriter(new FileWriter(to
    ))) {
236             out.write(tm.getVideo().stream()
237                     .map(Clip::toString)
238                     .collect(joining("\n")));
239             out.write("\n->\n");
240             out.write(tm.getAudio().stream()
241                     .map(Clip::toString)
242                     .collect(joining("\n")));
243         }
244     }
245 
246     /**
247      * method to load from a saved project file
248      *
249      * @param from the project file to load from
250      * @return the loaded @see{@link Timeline} instance
251      * @throws IOException if an error occurs while reading from the given
```

```
251  file
252       */
253      public Timeline loadProject(File from) throws IOException {
254          Timeline tm = new Timeline();
255          try (BufferedReader in = new BufferedReader(new FileReader(from
     ))) {
256              Iterator<String> lines = in.lines().iterator();
257              while (lines.hasNext()) {
258                  String next = lines.next();
259                  if (next.equals("->")) break;
260                  tm.addVideo(Clip.parse(next));
261              }
262              while (lines.hasNext()) {
263                  tm.addAudio(Clip.parse(lines.next()));
264              }
265          }
266          return tm;
267      }
268
269      /**
270       * list all files in a give directory recursively
271       *
272       * @param dir   the directory to return the files from
273       * @param files a reference to a List object to save the files into
274       */
275      private static void listf(File dir, List<File> files) {
276          File[] fList = dir.listFiles();
277          if (fList != null) {
278              for (File file : fList) {
279                  if (file.isFile()) {
280                      files.add(file);
281                  } else if (file.isDirectory()) {
282                      listf(file, files);
283                  }
284              }
285          }
286      }
287
288      /**
289       * delete a complete directory with all its contents
290       *
291       * @param dir the directory to be deleted
292       * @throws IOException thrown if an error occurs while deleting the
     directory
293       */
294      private static void delDir(File dir) throws IOException {
295          Files.walk(dir.toPath())
296                  .sorted(Comparator.reverseOrder())
297                  .map(Path::toFile)
298                  .forEach(File::delete);
299      }
300
301      /**
302       * getter for the source files
303       * @return list of files
```

```
304       */
305      public List<File> getSources() {
306          return files;
307      }
308  }
```

```java
package at.jku.videocuttingtool.backend;

import com.sun.deploy.util.StringUtils;

import java.io.File;
import java.util.Arrays;

/**
 * Class for a media file,
 * which can include Timestamps [from - to].
 * The media will be cut at the specified timestamps
 */
public class Clip implements Comparable<Clip> {
    private int pos;
    private final File media;
    private String start;
    private String end;

    /**
     * @param media original uncut media link
     * @param pos   the position of the clip in the timeline
     * @param start start time (relative to the media) of cut media
     *              hh:mm:ss.xxx
     *              if "" -> no cut in front
     * @param end   length of the cut media in the timeline
     *              hh:mm:ss.xxx
     *              if "" -> whole cut media will be placed at pos
     */
    public Clip(File media, int pos, String start, String end) {
        this.media = media;
        this.pos = pos;
        this.start = start;
        this.end = end;
    }

    /**
     *
     * @param media original uncut media link
     * @param pos   the position of the clip in the timeline
     */
    public Clip(File media, int pos){
        this.media = media;
        this.pos = pos;
        this.start = "";
        this.end = "";
    }

    @Override
    public String toString() {
        return StringUtils.join(Arrays.asList(
                "" + pos,
                media.getAbsolutePath(),
                start,
                end
        ), ";");
```

```java
    }

    @Override
    public int compareTo(Clip o) {
        return Integer.compare(pos, o.pos);
    }

    /**
     * generate a Clip object from a loaded file-entry
     * @param from line entry from the loaded file
     * @return parsed Clip object
     */
    public static Clip parse(String from) {
        String[] split = from.split(";");
        String start = "", end = "";

        if (split.length < 2) {
            return null;
        } else if (split.length == 3) {
            start = split[2];
        } else if (split.length == 4) {
            start = split[2];
            end = split[3];
        }

        return new Clip(
                new File(split[1]),
                Integer.parseInt(split[0]),
                start,
                end
        );
    }

    public int getPos() {
        return pos;
    }

    public File getMedia() {
        return media;
    }

    public String getStart() {
        return start;
    }

    public String getEnd() {
        return end;
    }

    public void setStart(String start) {
        this.start = start;
    }

    public void setEnd(String end) {
        this.end = end;
```

```
111     }
112
113     public void setPos(int pos) {
114         this.pos = pos;
115     }
116 }
117
```

```java
package at.jku.videocuttingtool.backend;

import java.io.BufferedReader;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.util.stream.Collectors;

/**
 * Exception class for the FFMPEG Error Streams
 */
public class EditMediaException extends Exception{
    private final String msg;

    /**
     * @param errorStream the error stream the execution of the ffmpeg commands return
     */
    public EditMediaException(InputStream errorStream){
        BufferedReader err = new BufferedReader(new InputStreamReader(errorStream));
        this.msg = err.lines().collect(Collectors.joining("\n"));
    }

    public String getMsg() {
        return msg;
    }

    @Override
    public String toString() {
        return "EditMediaException{" +
                "msg='" + msg + '\'' +
                '}';
    }

    @Override
    public void printStackTrace() {
        System.out.println(msg);
    }
}

```

```java
package at.jku.videocuttingtool.backend;

/**
 * Class to build a ffmpeg command
 */
public class Exec {
    private final StringBuilder build = new StringBuilder();

    /**
     * @param ffmpeg location of the ffmpeg.exe
     */
    public Exec(String ffmpeg) {
        build.append(ffmpeg);
    }

    /**
     * add an input to the command
     *
     * @param in the input file /file location
     * @return the new instance of this command object
     */
    public Exec addInput(String in) {
        build.append(" -i \"").append(in).append('\"');
        return this;
    }

    /**
     * add an parameter to the ffmpeg command
     *
     * @param arg parameter, that can be only one, but also multiple sequential ones
     * @return the new instance of this command object
     */
    public Exec addArg(String arg) {
        build.append(' ').append(arg);
        return this;
    }

    /**
     * add a output file to the command
     *
     * @param out the file /file location
     * @return the new instance of this command object
     */
    public Exec addOutput(String out) {
        build.append(" \"").append(out).append('\"');
        return this;
    }

    /**
     * get the completed command
     *
     * @return the completed command as a String
     */
    public String done() {
```

```
55          return build.toString();
56      }
57  }
58
```

```java
package at.jku.videocuttingtool.backend;

import java.io.File;
import java.nio.file.Paths;

/**
 * Container Class for an export
 */
public class Export {
    private final Timeline timeline;
    private final File export;
    private final String videoFormat;
    private final String audioFormat;

    /**
     * @param timeline    timeline instance which should be exported
     * @param export      the directory+filename to export to
     * @param videoFormat the format / codec of the exported video
     * @param audioFormat the format / codec for the audio of the exported video
     */
    public Export(Timeline timeline, File export, String videoFormat, String audioFormat) {
        this.timeline = timeline;
        this.export = export;
        this.videoFormat = videoFormat;
        this.audioFormat = audioFormat;
    }

    public Timeline getTimeline() {
        return timeline;
    }

    public File getExport() {
        return export;
    }

    public String getVideoFormat() {
        return videoFormat;
    }

    public String getAudioFormat() {
        return audioFormat;
    }
}

```

```java
package at.jku.videocuttingtool.backend;


import java.util.*;

/**
 * Container class for the video and audio tracks of the timeline
 */
public class Timeline {
    private final List<Clip> video = new ArrayList<>();
    private final List<Clip> audio = new ArrayList<>();

    public void addVideo(Clip video) {
        this.video.add(video);
    }

    public void addAudio(Clip audio) {
        this.audio.add(audio);
    }

    public void addVideo(List<Clip> video) {
        this.video.addAll(video);
    }

    public void addAudio(List<Clip> audio) {
        this.audio.addAll(audio);
    }

    public List<Clip> getVideo() {
        video.sort(Clip::compareTo);
        return video;
    }

    public List<Clip> getAudio() {
        audio.sort(Clip::compareTo);
        return audio;
    }
}

```