

Name: Simon Offenberger/ Simon Vogelhuber

Aufwand in h: siehe Doku

Mat.Nr: S2410306027/ S2410306014

Punkte:

Übungsgruppe: 1

korrigiert:

Beispiel 1 (24 Punkte) Player-Schnittstelle: Sie verwenden in Ihrer Firma HSDSoft einen MusicPlayer von der Firma MonkeySoft. Die öffentliche Schnittstelle des MusicPlayers sieht folgendermaßen aus und kann nicht verändert werden:

```
1 //starts playing with the current song in list
2 void Start();
3 //stops playing
4 void Stop();
5 //switches to next song and starts at the end with first song
6 void SwitchNext();
7 //get index of current song
8 size_t const GetCurIndex() const;
9 //find a song by name in playlist
10 bool Find(std::string const& name);
11 //get count of songs in playlist
12 size_t const GetCount() const;
13 //increase the volume relative to the current volume
14 void IncreaseVol(size_t const vol);
15 //decrease the volume relative to the current volume
16 void DecreaseVol(size_t const vol);
17 //add a song to playlist
18 void Add(std::string const& name, size_t const dur);
```

Der MusicPlayer verwaltet Lieder und speichert den Namen und die Dauer jedes Liedes in Sekunden. Er kann gestartet und gestoppt werden und erlaubt das Verändern der Lautstärke. Die Lautstärke ist begrenzt mit 0 und maximal 100. Der Defaultwert für die Lautstärke liegt bei 15.

Zur Simulation liefert der Player je nach Aktion folgende Ausgaben auf der Konsole:

```
playing song number 1: Hells Bells (256 sec)
...
playing song number 4: Hawaguck (129 sec)
...
volume is now -> 70
song: Pulp Fiction not found!
stop song: Hells Bells (256 sec)
...
no song in playlist!
```

In weiterer Folge kaufen Sie einen VideoPlayer der Firma DonkeySoft mit folgender vorgegebenen Schnittstelle:

```
1 //starts playing with the current song in list
2 void Play() const;
3 //stops playing
4 void Stop() const;
5 //switches to first video in playlist and returns true, otherwise false if list is empty.
6 bool First();
7 //switches to next video in playlist and returns true, otherwise false if last song is reached.
8 bool Next();
9 //returns index of current video
10 size_t CurIndex() const;
11 //returns name of current video
12 std::string CurVideo() const;
13 //sets volume (min volume=0 and max volume=50)
14 void SetVolume(size_t const vol);
15 //gets current volume
16 size_t const GetVolume() const;
17 //adds a video to playlist
18 void Add(std::string const& name, size_t const dur, VideoFormat const& format);
19 }
```

Der VideoPlayer kann die Formate WMV, AVI und MKV abspielen. Er verwaltet Videos und speichert den Namen und die Dauer in Minuten. Er kann gestartet und gestoppt werden und erlaubt das Verändern der Lautstärke. Die Lautstärke ist begrenzt mit 0 und maximal 50. Der Defaultwert für die Lautstärke liegt bei 8.

Zur Simulation liefert der Player je nach Aktion folgende Ausgaben auf der Konsole:

```
playing video number 1: Die Sendung mit der Maus [duration -> 55 min], AVI-Format
...
playing video number 3: Freitag der 13te [duration -> 95 min], WMV-Format
...
volume is now -> 30
video: Hells Bells not found!
stop video: Pulp Fiction [duration -> 126 min], MKV-Format
...
no video in playlist!
```

Für einen Klienten soll nun nach außen folgende, unabhängige Schnittstelle zur Verfügung gestellt werden:

```
1 virtual void Play() = 0;
2 virtual void VolInc() = 0;
3 virtual void VolDec() = 0;
4 virtual void Stop() = 0;
5 virtual void Next() = 0;
6 virtual void Prev() = 0;
7 virtual void Select(std::string const& name) = 0;
```

Mit dieser Schnittstelle kann der Klient sowohl den MusicPlayer als auch den VideoPlayer verwenden. Die Methoden `VolInc()` und `VolDec()` erhöhen bzw. erniedrigen die Lautstärke um den Wert 1. `Next()` und `Prev()` schalten vor und zurück. `Select(...)` wählt ein Lied oder ein Video aus der Playliste aus.

Achten Sie beim Design auf die Einhaltung der Design-Prinzipien und verwenden Sie ein entsprechendes Design-Pattern!

Implementieren Sie alle notwendigen Klassen (auch die Music/VideoPlayer-Klassen) und testen Sie diese entsprechend!

Treffen Sie für alle unzureichenden Angaben sinnvolle Annahmen und begründen Sie diese. Verfassen Sie weiters eine Systemdokumentation (entsprechend den Vorgaben aus Übung1)!

Allgemeine Hinweise: Legen Sie bei der Erstellung Ihrer Übung großen Wert auf eine **saubere Strukturierung** und auf eine **sorgfältige Ausarbeitung**! Dokumentieren Sie alle Schnittstellen und versehen Sie Ihre Algorithmen an entscheidenden Stellen ausführlich mit Kommentaren! Testen Sie ihre Implementierungen ausführlich! Geben Sie den **Testoutput** mit ab!



HSD

FH-HAGENBERG

Systemdokumentation Projekt Music/VideoPlayer Adapter

Version 1.0

S. Offenberger, S. Vogelhuber

Hagenberg, 3. November 2025

Inhaltsverzeichnis

1	Organisatorisches	6
1.1	Team	6
1.2	Aufteilung der Verantwortlichkeitsbereiche	6
1.3	Aufwand	7
2	Anforderungsdefinition (Systemspezifikation)	8
2.1	IPlayer Interface Anforderung	8
2.2	VideoPlayer Anforderung	9
2.3	VideoPlayer Anforderung	10
3	Systementwurf	12
3.1	Klassendiagramm	12
3.2	Designentscheidungen	13
4	Dokumentation der Komponenten (Klassen)	13
5	Testprotokollierung	14
6	Quellcode	21
6.1	Object.hpp	21
6.2	Client.hpp	22
6.3	Client.cpp	23
6.4	IPlayer.hpp	27
6.5	MusicPlayerAdapter.hpp	28
6.6	MusicPlayerAdapter.cpp	29
6.7	MusicPlayer.hpp	30
6.8	MusicPlayer.cpp	32
6.9	Song.hpp	34
6.10	Song.cpp	35
6.11	VideoPlayerAdapter.hpp	36
6.12	VideoPlayerAdapter.cpp	37
6.13	VideoPlayer.hpp	38
6.14	VideoPlayer.cpp	40
6.15	Video.hpp	42
6.16	Video.cpp	43
6.17	EVideoFormat.hpp	44
6.18	main.cpp	45
6.19	Test.hpp	53

1 Organisatorisches

1.1 Team

- Simon Offenberger, Matr.-Nr.: S2410306027, E-Mail: S2410306027@fhooe.at
- Simon Vogelhuber, Matr.-Nr.: S2410306014, E-Mail: s2410306014@fhooe.at

1.2 Aufteilung der Verantwortlichkeitsbereiche

- Simon Offenberger
 - Design Klassendiagramm
 - Implementierung und Test der Klassen:
 - * Client,
 - * VideoPlayerAdapter,
 - * VideoPlayer,
 - * Video,
 - * EVideoFormat,
 - Implementierung des Testtreibers
 - Dokumentation
- Simon Vogelhuber
 - Design Klassendiagramm
 - Implementierung und Komponententest der Klassen:
 - * IPlayer
 - * MusicPlayerAdapter,
 - * MusicPlayer,
 - * Song

- Implementierung des Testtreibers
- Dokumentation

1.3 Aufwand

- Simon Offenberger: geschätzt 12 Ph / tatsächlich 11 Ph
- Simon Vogelhuber: geschätzt 9 Ph / tatsächlich 9 Ph

2 Anforderungsdefinition (Systemspezifikation)

Für die Implementierung wurden die Header von MusicPlayer, VideoPlayer und IPlayer Interface vorgegeben. Die Anforderung bestand darin einen Client eine gemeinsame Schnittstelle zum Ansprechen von MusicPlayer sowie VideoPlayer zu bieten. Die Schnittstelle soll folgende Funktionen bereitstellen.

2.1 IPlayer Interface Anforderung

- Play
 - Spielt das Video bzw. den Song des entsprechenden Players -> Ausgabe auf COUT
- VolInc
 - Diese Methode soll die Lautstärke des Players um 1 erhöhen.
- VolDec
 - Diese Methode soll die Lautstärke des Players um 1 verringern.
- Stop
 - Stoppt die Wiedergabe
- Next
 - Wechselt den aktuellen Titel auf den nächsten in der Liste
- Next
 - Wechselt den aktuellen Titel auf den vorherigen in der Liste
- Select
 - Wählt einen Titel über den Namen aus

2.2 VideoPlayer Anforderung

Folgende Anforderungen müssen die Methoden des VideoPlayers bereitstellen:

- Play
 - Spielt das Video ab -> Ausgabe auf COUT
- Stop
 - Stopt das Video -> Ausgabe auf COUT
- First
 - Wechsel auf den ersten Titel in der Playlist
 - gibt true zurück wenn dies erfolgreich ist
 - gibt false wenn kein Titel in der Playlist ist
- Next
 - Wechsel auf den nächsten Titel in der Playlist
 - gibt true zurück wenn dies erfolgreich ist
 - gibt false wenn kein weiterer Titel in der Playlist ist
- CurIndex
 - Liefert den aktuellen Index der Playlist
- CurVideo
 - Liefert den aktuellen Title als string
- SetVolume
 - Setzt die Lautstärke des Titles max 50 min 0
- GetVolume
 - Liefert die aktuelle Lautstärke
- Add
 - Fügt und erzeugt ein Video an die Playlist hinten an

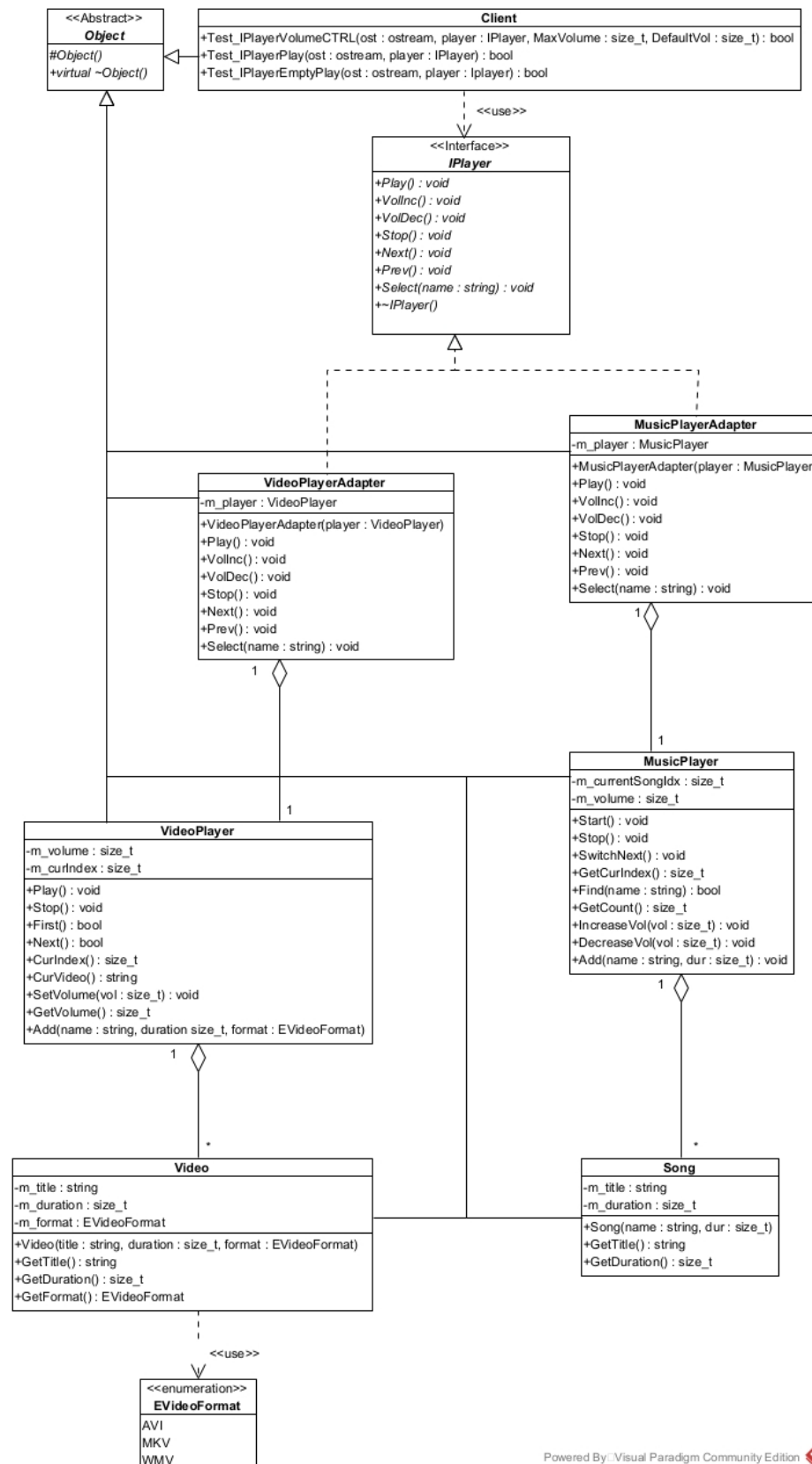
2.3 VideoPlayer Anforderung

Folgende Anforderungen müssen die Methoden des MusicPlayers bereitstellen:

- Start
 - Spielt den Song ab -> Ausgabe auf COUT
- Stop
 - Stopt den Song -> Ausgabe auf COUT
- SwitchNext
 - Wechsel auf den nächsten Titel in der Playlist am Ende wird mit den ersten fortgesetzt
- GetCurIndex
 - Liefert den aktuellen Index der Playlist
- Find
 - Sucht nach einem Titel und wählt ihn aus
 - gibt true wenn Titel gefunden wurde
 - gibt false wenn Titel nicht gefunden wurde
- GetCount
 - Gibt die Anzahl der Lieder in der Playlist zurück
- IncreaseVol
 - erhöht die Lautstärke um einen bestimmten Wert (max 100)
- DecreaseVol
 - reduziert die Lautstärke um einen bestimmten Wert (min 0)
- Add
 - Fügt und erzeugt ein Video an die Playlist hinten an

3 Systementwurf

3.1 Klassendiagramm



3.2 Designentscheidungen

Die Klassen Video und Song wurden so umgesetzt, dass diese für die Speicherung der spezifischen Daten eingesetzt werden. Hier wird in den Playerklassen ein Container von Videos bzw. Songs gespeichert. Für die Bereitstellung eines gemeinsamen Interfaces für den Client wurden Adapter für den Music- bzw. Video Player implementiert. Dieser Adapter speichern intern nur eine Referenz auf den tatsächlichen Players. Dies ermöglicht es den Player selbst als auch den Adapter simultan zu verwenden. Im Adapter müssten die Funktion der Player so angewandt und kombiniert werden, dass für beide Player über das Interface diesselbe Funktionalität zur Verfügung steht.

Die Gemeinsamen funktionen des Interfaces wurde im Client getestet. Alle anderen Klassen wurden im main getestet.

In der Übung wurde nachgefragt ob die starre Ausgabe auf cout, über einen Parameter in der Methode ausgetauscht werden kann, aber nach Absprache mit Herrn Wiesinger dürfen keine Veränderungen vorgenommen werden. Somit musste im Testtreiber cout umgeleitet werden um einen sinnvollen Testtreiber zu schreiben.

4 Dokumentation der Komponenten (Klassen)

Die HTML-Startdatei befindet sich im Verzeichnis [../doxy/html/index.html](http://doxy/html/index.html)

5 Testprotokollierung

```
1 Test VideoPlayer Adapter in Client
2
3 *****
4 TESTCASE START
5 *****
6
7 Test Volume Inc
8 [Test FAILED] Result: (Expected: true != Result: false)
9
10 Test Volume Dec
11 [Test FAILED] Result: (Expected: true != Result: false)
12
13 Test Lower Bound Volume 0
14 [Test OK] Result: (Expected: true == Result: true)
15
16 Test Upper Bound Volume
17 [Test OK] Result: (Expected: true == Result: true)
18
19 Test for Exceotion in Test Case
20 [Test OK] Result: (Expected: true == Result: true)
21
22
23 *****
24
25
26 *****
27 TESTCASE START
28 *****
29
30 Test Play Contains Name
31 [Test FAILED] Result: (Expected: true != Result: false)
32
33 Test Next
34 [Test FAILED] Result: (Expected: true != Result: false)
35
36 Test Next
37 [Test OK] Result: (Expected: true == Result: true)
38
39 Test Next
40 [Test OK] Result: (Expected: true == Result: true)
41
42 Test Next
43 [Test OK] Result: (Expected: true == Result: true)
44
45 Test Next
46 [Test OK] Result: (Expected: true == Result: true)
```

```
47
48 Test Next Wrap around
49 [Test FAILED] Result: (Expected: true != Result: false)
50
51 Test Select Video by name
52 [Test OK] Result: (Expected: true == Result: true)
53
54 Test Select Video by name not found
55 [Test OK] Result: (Expected: true == Result: true)
56
57 Test Stop Player
58 [Test OK] Result: (Expected: true == Result: true)
59
60 Test for Exception in Test Case
61 [Test OK] Result: (Expected: true == Result: true)
62
63
64 *****
65
66
67 *****
68 TESTCASE START
69 *****
70
71 Test for Message in Empty Player
72 [Test OK] Result: (Expected: true == Result: true)
73
74 Test for Exception in Testcase
75 [Test OK] Result: (Expected: true == Result: true)
76
77
78 *****
79
80 Test MediaPlayer Adapter in Client
81
82 *****
83 TESTCASE START
84 *****
85
86 Test Volume Inc
87 [Test FAILED] Result: (Expected: true != Result: false)
88
89 Test Volume Dec
90 [Test FAILED] Result: (Expected: true != Result: false)
91
92 Test Lower Bound Volume 0
93 [Test OK] Result: (Expected: true == Result: true)
94
95 Test Upper Bound Volume
```

```
96 [Test OK] Result: (Expected: true == Result: true)
97
98 Test for Exceotion in Test Case
99 [Test OK] Result: (Expected: true == Result: true)
100
101
102 *****
103
104
105 *****
106             TESTCASE START
107 *****
108
109 Test Play Contains Name
110 [Test FAILED] Result: (Expected: true != Result: false)
111
112 Test Next
113 [Test FAILED] Result: (Expected: true != Result: false)
114
115 Test Next
116 [Test OK] Result: (Expected: true == Result: true)
117
118 Test Next
119 [Test OK] Result: (Expected: true == Result: true)
120
121 Test Next
122 [Test OK] Result: (Expected: true == Result: true)
123
124 Test Next
125 [Test OK] Result: (Expected: true == Result: true)
126
127 Test Next Wrap around
128 [Test FAILED] Result: (Expected: true != Result: false)
129
130 Test Select Video by name
131 [Test OK] Result: (Expected: true == Result: true)
132
133 Test Select Video by name not found
134 [Test OK] Result: (Expected: true == Result: true)
135
136 Test Stop Player
137 [Test OK] Result: (Expected: true == Result: true)
138
139 Test for Exception in Test Case
140 [Test OK] Result: (Expected: true == Result: true)
141
142
143 *****
144
```



```
145
146 *****
147 TESTCASE START
148 *****
149
150 Test for Message in Empty Player
151 [Test OK] Result: (Expected: true == Result: true)
152
153 Test for Exception in Testcase
154 [Test OK] Result: (Expected: true == Result: true)
155
156
157 *****
158
159
160 *****
161 TESTCASE START
162 *****
163
164 Test Song Getter Duration
165 [Test OK] Result: (Expected: 123 == Result: 123)
166
167 Test Song Getter Name
168 [Test OK] Result: (Expected: Hello World == Result: Hello World)
169
170 Check for Exception in Testcase
171 [Test OK] Result: (Expected: true == Result: true)
172
173 Test Exception in Song CTOR with duration 0
174 [Test OK] Result: (Expected: ERROR: Song with duration 0! == Result:
    ↪ ERROR: Song with duration 0!)
175
176 Test Exception in Song CTOR with empty string
177 [Test OK] Result: (Expected: ERROR: Song with empty Name! == Result:
    ↪ ERROR: Song with empty Name!)
178
179
180 *****
181
182
183 *****
184 TESTCASE START
185 *****
186
187 Test Song Getter Duration
188 [Test OK] Result: (Expected: 123 == Result: 123)
189
190 Test Song Getter Name
191 [Test OK] Result: (Expected: Hello World == Result: Hello World)
```

```
192
193 Test Song Getter Format
194 [Test OK] Result: (Expected: AVI-Format == Result: AVI-Format)
195
196 Check for Exception in Testcase
197 [Test OK] Result: (Expected: true == Result: true)
198
199 Test Exception in Video CTOR with duration 0
200 [Test OK] Result: (Expected: ERROR: Video with duration 0! == Result:
    ↪ ERROR: Video with duration 0!)
201
202 Test Exception in Video CTOR with empty string
203 [Test OK] Result: (Expected: ERROR: Video with empty Name! == Result:
    ↪ ERROR: Video with empty Name!)
204
205
206 *****
207
208
209 *****
210 TESTCASE START
211 *****
212
213 Test Videoplayer Initial Index
214 [Test OK] Result: (Expected: 0 == Result: 0)
215
216 Test Videoplayer Index after First
217 [Test OK] Result: (Expected: 0 == Result: 0)
218
219 Test Videoplayer Index after Next
220 [Test OK] Result: (Expected: 1 == Result: 1)
221
222 Test Videoplayer Index Upper Bound
223 [Test OK] Result: (Expected: 4 == Result: 4)
224
225 Test Videoplayer Index after First
226 [Test OK] Result: (Expected: 0 == Result: 0)
227
228 Test Default Volume
229 [Test OK] Result: (Expected: 8 == Result: 8)
230
231 Test Set Volume
232 [Test OK] Result: (Expected: 25 == Result: 25)
233
234 Test Set Volume Max Volume
235 [Test OK] Result: (Expected: 50 == Result: 50)
236
237 Test Set Volume Min Volume
238 [Test OK] Result: (Expected: 0 == Result: 0)
```

```
239
240 Test Video Player Play
241 [Test OK] Result: (Expected: true == Result: true)
242
243 Test Video Player Stop
244 [Test OK] Result: (Expected: true == Result: true)
245
246 Check for Exception in Testcase
247 [Test OK] Result: (Expected: true == Result: true)
248
249 Test Exception in Add with empty string
250 [Test OK] Result: (Expected: ERROR: Video with empty Name! == Result:
    ↪ ERROR: Video with empty Name!)
251
252 Test Exception in Add with empty string
253 [Test OK] Result: (Expected: ERROR: Video with duration 0! == Result:
    ↪ ERROR: Video with duration 0!)
254
255
256 *****
257
258
259 *****
260 TESTCASE START
261 *****
262
263 MediaPlayer - Basic Functionality - .GetCount()
264 [Test OK] Result: (Expected: 4 == Result: 4)
265
266 MediaPlayer - Basic Functionality - .GetIndex() initial
267 [Test OK] Result: (Expected: 0 == Result: 0)
268
269 MediaPlayer - Basic Functionality - .Find() unknown song
270 [Test OK] Result: (Expected: false == Result: false)
271
272 MediaPlayer - Basic Functionality - .Find() song that exists
273 [Test OK] Result: (Expected: true == Result: true)
274
275 MediaPlayer - Basic Functionality - Song name after initial .Start()
276 [Test OK] Result: (Expected: true == Result: true)
277
278 MediaPlayer - Basic Functionality - .GetIndex() after switching
279 [Test OK] Result: (Expected: 1 == Result: 1)
280
281 MediaPlayer - Basic Functionality - Song name switching
282 [Test OK] Result: (Expected: true == Result: true)
283
284 MediaPlayer - Basic Functionality - .GetIndex() wrap around
285 [Test OK] Result: (Expected: 1 == Result: 1)
```

```
286
287 MediaPlayer - Basic Functionality - Error Buffer
288 [Test OK] Result: (Expected: true == Result: true)
289
290 MediaPlayer - Add Song without title
291 [Test OK] Result: (Expected: ERROR: Song with empty Name! == Result:
    ↪ ERROR: Song with empty Name!)
292
293 MediaPlayer - Add Song without title
294 [Test OK] Result: (Expected: ERROR: Song with duration 0! == Result:
    ↪ ERROR: Song with duration 0!)
295
296 MediaPlayer - Add Song without title
297 [Test OK] Result: (Expected: ERROR: Song with empty Name! == Result:
    ↪ ERROR: Song with empty Name!)
298
299 TEST OK!!
```

6 Quellcode

6.1 Object.hpp

```
1  /*****
2  * \file   Object.hpp
3  * \brief  common ancestor for all objects
4  *
5  * \author Simon
6  * \date   November 2025
7  *****/
8  #ifndef OBJECT_HPP
9  #define OBJECT_HPP
10
11 #include <string>
12
13 class Object {
14 public:
15
16     inline static const std::string ERROR_BAD_OSTREAM = "ERROR:_Provided_Ostream_is_bad";
17     inline static const std::string ERROR_FAIL_WRITE = "ERROR:_Fail_to_write_on_provided_Ostream";
18     inline static const std::string ERROR_NULLPTR = "ERROR:_Passed_in_Nullptr!";
19
20     // once virtual always virtual
21     virtual ~Object() = default;
22
23
24 protected:
25     Object() = default;
26 };
27
28 #endif // !OBJECT_HPP
```

6.2 Client.hpp

```

1  /*****
2  * \file   Client.hpp
3  * \brief  Client Class that uses a IPlayer Interface in order to control
4  * \brief  a Musicplayer or a Videoplayer via their adapter
5  *
6  * \author Simon
7  * \date   November 2025
8  *****/
9  #ifndef CLIENT_HPP
10 #define CLIENT_HPP
11
12 #include "Object.hpp"
13 #include "IPlayer.hpp"
14 #include <iostream>
15
16 class Client : public Object
17 {
18 public:
19     /**
20     * \brief Test Function for the Volume Control of the IPlayer interface.
21     *
22     * \param ost Ostream
23     * \param player Reference to the player
24     * \param MaxVolume Maximum Volume of the player
25     * \param DefaultVol Default Volume of the player
26     * \return true -> tests OK
27     * \return false -> tests failed
28     */
29     bool Test_IPlayerVolumeCTRL(std::ostream& ost, IPlayer& player, const size_t& MaxVolume, const size_t&
        DefaultVol) const;
30
31     /**
32     * \brief Test Play of the Player.
33     *
34     * \param ost Ostream for the Testoutput
35     * \param player Reference to player
36     * \return true -> tests OK
37     * \return false -> tests failed
38     */
39     bool Test_IPlayerPlay(std::ostream& ost, IPlayer& player) const;
40
41     /**
42     * \brief Test Play of an empty Player.
43     *
44     * \param ost Ostream for the Testoutput
45     * \param player Reference to player
46     * \return true -> tests OK
47     * \return false -> tests failed
48     */
49     bool Test_IPlayerEmptyPlay(std::ostream& ost, IPlayer& player) const;
50 };
51
52 #endif // !CLIENT_HPP

```

6.3 Client.cpp

```

1  #include "Client.hpp"
2  #include "Test.hpp"
3  #include <sstream>
4  #include <algorithm>
5
6  using namespace std;
7
8  bool Client::Test_IPlayerVolumeCTRL(std::ostream& ost, IPlayer& player, const size_t & MaxVolume, const size_t &
   DefaultVol) const
9  {
10     if (!ost.good()) throw Client::ERROR_BAD_OSTREAM;
11
12     TestStart(ost);
13
14     bool TestOK = true;
15     string error_msg = "";
16
17     try {
18
19         stringstream result;
20
21         std::streambuf* coutbuf = std::cout.rdbuf();
22
23         result << DefaultVol+1;
24         string DVol;
25
26         result >> DVol;
27
28         result.clear();
29         result.str("");
30
31         // cout redirect to stringstream
32         std::cout.rdbuf(result.rdbuf());
33
34         player.VollInc();
35
36         std::cout.rdbuf(coutbuf);
37
38         TestOK == TestOK && check_dump(ost, "Test_Volume_Inc", true, result.str().find(DVol)!=std::string::
           npos);
39
40         result.clear();
41         result.str("");
42
43         result << DefaultVol;
44
45         result >> DVol;
46
47         result.clear();
48         result.str("");
49
50         // cout redirect to stringstream
51         std::cout.rdbuf(result.rdbuf());
52
53         player.VollDec();
54
55         std::cout.rdbuf(coutbuf);
56
57         TestOK == TestOK && check_dump(ost, "Test_Volume_Dec", true, result.str().find(DVol)!=std::string::
           npos);
58
59         // cout redirect to stringstream
60         std::cout.rdbuf(result.rdbuf());
61
62         for (int i = 0; i < 200; i++) player.VollDec();
63
64         player.VollInc();
65
66         std::cout.rdbuf(coutbuf);
67
68         result.clear();
69         result.str("");
70
71
72         // cout redirect to stringstream
73         std::cout.rdbuf(result.rdbuf());
74
75         player.VollDec();
76
77         std::cout.rdbuf(coutbuf);

```

```

78
79         TestOK == TestOK && check_dump(ost, "Test_Lower_Bound_Volume_0", true, result.str().find("0") != std
            ::string::npos);
80
81         // cout redirect to stringstream
82         std::cout.rdbuf(result.rdbuf());
83
84         for (int i = 0; i < 200; i++) player.VollInc();
85
86         std::cout.rdbuf(coutbuf);
87
88         result.clear();
89         result.str("");
90
91         result << MaxVolume;
92
93         string MaxVol;
94
95         result >> MaxVol;
96
97         result.clear();
98         result.str("");
99
100        // cout redirect to stringstream
101        std::cout.rdbuf(result.rdbuf());
102
103        player.VollInc();
104
105        std::cout.rdbuf(coutbuf);
106
107
108        TestOK == TestOK && check_dump(ost, "Test_Upper_Bound_Volume", true, result.str().find(MaxVol) !=
            std::string::npos);
109    }
110    catch (const string& err) {
111        error_msg = err;
112        TestOK = false;
113    }
114    catch (bad_alloc const& error) {
115        error_msg = error.what();
116        TestOK = false;
117    }
118    catch (const exception& err) {
119        error_msg = err.what();
120        TestOK = false;
121    }
122    catch (...) {
123        error_msg = "Unhandelt_Exception";
124        TestOK = false;
125    }
126
127    TestOK == TestOK && check_dump(ost, "Test_for_Exception_in_Test_Case", true, error_msg.empty());
128
129    TestEnd(ost);
130
131    if (ost.fail()) throw Client::ERROR_FAIL_WRITE;
132
133    return TestOK;
134}
135
136bool Client::Test_IPlayerPlay(std::ostream& ost, IPlayer& player) const
137{
138    if (!ost.good()) throw Client::ERROR_BAD_OSTREAM;
139
140    TestStart(ost);
141
142    bool TestOK = true;
143    string error_msg = "";
144
145    try {
146
147        stringstream result;
148        std::streambuf* coutbuf = std::cout.rdbuf();
149
150        // cout redirect to stringstream
151        std::cout.rdbuf(result.rdbuf());
152
153        player.Play();
154
155        std::cout.rdbuf(coutbuf);
156
157        TestOK == TestOK && check_dump(ost, "Test_Play_Contains_Name", true, result.str().find("Harry_
            Potter1") != std::string::npos);

```



```
158     player.Next();
159
160     result.str("");
161     result.clear();
162
163     std::cout.rdbuf(result.rdbuf());
164
165     player.Play();
166
167     std::cout.rdbuf(coutbuf);
168
169     TestOK == TestOK && check_dump(ost, "Test_Next_", true, result.str().find("Harry_Potter2") != std::
170     string::npos);
171
172     for (int i = 0; i < 4; i++) {
173
174         player.Next();
175
176         result.str("");
177         result.clear();
178
179         std::cout.rdbuf(result.rdbuf());
180
181         player.Play();
182
183         std::cout.rdbuf(coutbuf);
184
185         TestOK == TestOK && check_dump(ost, "Test_Next_", true, result.str().find("Harry_Potter" + 2
186         + i) != std::string::npos);
187     }
188
189     player.Next();
190
191     result.str("");
192     result.clear();
193
194     std::cout.rdbuf(result.rdbuf());
195
196     player.Play();
197
198     std::cout.rdbuf(coutbuf);
199
200     TestOK == TestOK && check_dump(ost, "Test_Next_Wrap_around", true, result.str().find("Harry_Potter1"
201     ) != std::string::npos);
202
203     result.str("");
204     result.clear();
205
206     std::cout.rdbuf(result.rdbuf());
207
208     player.Select("Harry_Potter3");
209     player.Play();
210
211     std::cout.rdbuf(coutbuf);
212
213     TestOK == TestOK && check_dump(ost, "Test_Select_Video_by_name_", true, result.str().find("Harry_
214     Potter3") != std::string::npos);
215
216     result.str("");
217     result.clear();
218
219     std::cout.rdbuf(result.rdbuf());
220
221     player.Select("Harry_Potter14");
222     player.Play();
223
224     std::cout.rdbuf(coutbuf);
225
226     TestOK == TestOK && check_dump(ost, "Test_Select_Video_by_name_not_found", true, result.str().find("
227     not_found!") != std::string::npos);
228
229     result.str("");
230     result.clear();
231
232     std::cout.rdbuf(result.rdbuf());
233
234     player.Select("Harry_Potter3");
235     player.Stop();
236
237     std::cout.rdbuf(coutbuf);
```

```

236         TestOK == TestOK && check_dump(ost, "Test_Stop_Player",
237             true,
238             result.str().find("stop") != std::string::npos && result.str().find("Harry_Potter3")
                != std::string::npos);
239
240
241     }
242     catch (const string& err) {
243         error_msg = err;
244         TestOK = false;
245     }
246     catch (bad_alloc const& error) {
247         error_msg = error.what();
248         TestOK = false;
249     }
250     catch (const exception& err) {
251         error_msg = err.what();
252         TestOK = false;
253     }
254     catch (...) {
255         error_msg = "Unhandelt_Exception";
256         TestOK = false;
257     }
258
259     TestOK == TestOK && check_dump(ost, "Test_for_Exception_in_Test_Case", true, error_msg.empty());
260
261     TestEnd(ost);
262
263     if (ost.fail()) throw Client::ERROR_FAIL_WRITE;
264
265     return TestOK;
266 }
267
268 bool Client::Test_IPlayerEmptyPlay(std::ostream& ost, IPlayer& player) const
269 {
270     if (!ost.good()) throw Client::ERROR_BAD_OSTREAM;
271
272     TestStart(ost);
273
274     bool TestOK = true;
275     string error_msg = "";
276
277     try {
278         stringstream result;
279
280         result.str("");
281         result.clear();
282
283         std::streambuf* coutbuf = std::cout.rdbuf();
284
285         std::cout.rdbuf(result.rdbuf());
286
287         player.Play();
288
289         std::cout.rdbuf(coutbuf);
290
291         TestOK == TestOK && check_dump(ost, "Test_for_Message_in_Empty_Player", true, result.str().find("no"
                )!=string::npos);
292
293     }
294     catch (const string& err) {
295         error_msg = err;
296     }
297     catch (bad_alloc const& error) {
298         error_msg = error.what();
299     }
300     catch (const exception& err) {
301         error_msg = err.what();
302     }
303     catch (...) {
304         error_msg = "Unhandelt_Exception";
305     }
306
307     TestOK == TestOK && check_dump(ost, "Test_for_Exception_in_Testcase", true , error_msg.empty());
308
309     TestEnd(ost);
310
311     if (ost.fail()) throw Client::ERROR_FAIL_WRITE;
312
313     return TestOK;
314 }

```

6.4 IPlayer.hpp

```
1  #ifndef IPLAYER_HPP
2  #define IPLAYER_HPP
3  /*****
4   * \file IPlayer.hpp
5   * \brief Interface to interact with various Player (music, video)
6   * \author Simon Vogelhuber
7   * \date October 2025
8   *****/
9
10 #include <iostream>
11
12 class IPlayer
13 {
14 public:
15     /**
16      * \brief Play selected song
17      */
18     virtual void Play() = 0;
19
20     /**
21      * \brief increase volume by 1 (out of 100)
22      */
23     virtual void VollInc() = 0;
24
25     /**
26      * \brief decrease volume by 1 (out of 100)
27      */
28     virtual void VollDec() = 0;
29
30     /**
31      * \brief Stop playing Song
32      */
33     virtual void Stop() = 0;
34
35     /**
36      * \brief Skip to next song
37      */
38     virtual void Next() = 0;
39
40     /**
41      * \brief Skip to previous song
42      */
43     virtual void Prev() = 0;
44
45     /**
46      * \brief Selects a Video by Name.
47      * \param name
48      */
49     virtual void Select(std::string const& name) = 0;
50
51     /**
52      * \brief virtual Destructor for Interface.
53      */
54     virtual ~IPlayer() = default;
55 };
56
57 #endif // !IPLAYER_HPP
```

6.5 MusicPlayerAdapter.hpp

```

1  /**
2   * \file MusicPlayerAdapter.hpp
3   * \brief
4   *
5   * \author Simon
6   * \date November 2025
7   */
8  #ifndef MUSIC_PLAYER_ADAPTER_HPP
9  #define MUSIC_PLAYER_ADAPTER_HPP
10
11 #include "IPlayer.hpp"
12 #include "MusicPlayer.hpp"
13
14 class MusicPlayerAdapter :public Object, public IPlayer
15 {
16 public:
17
18     MusicPlayerAdapter(MusicPlayer & player) : m_player{ player } {}
19
20     /**
21      * \brief Play selected song
22      */
23     virtual void Play() override;
24
25     /**
26      * \brief increase volume by 1 (out of 100)
27      */
28     virtual void VollInc() override;
29
30     /**
31      * \brief decrease volume by 1 (out of 100)
32      */
33     virtual void VollDec() override;
34
35     /**
36      * \brief Stop playing Song
37      */
38     virtual void Stop() override;
39
40     /**
41      * \Skip to next song
42      */
43     virtual void Next() override;
44
45     /**
46      * \brief Skip to previous song
47      */
48     virtual void Prev() override;
49
50     /**
51      * \brief Selects a Video by Name.
52      *
53      * \param name
54      */
55     virtual void Select(std::string const& name) override;
56
57     // delete Copy Ctor and Assign Operator to prohibit untestet behaviour
58     MusicPlayerAdapter(MusicPlayerAdapter& Music) = delete;
59     void operator=(MusicPlayerAdapter Music) = delete;
60
61 private:
62     MusicPlayer & m_player;
63 };
64
65 #endif // !MUSIC_PLAYER_ADAPTER_HPP

```

6.6 MusicPlayerAdapter.cpp

```
1  #include "MusicPlayerAdapter.hpp"
2
3  void MusicPlayerAdapter::Play()
4  {
5      m_player.Start();
6  }
7
8  void MusicPlayerAdapter::VollInc()
9  {
10     m_player.IncreaseVol(1);
11 }
12
13 void MusicPlayerAdapter::VollDec()
14 {
15     m_player.DecreaseVol(1);
16 }
17
18 void MusicPlayerAdapter::Stop()
19 {
20     m_player.Stop();
21 }
22
23 void MusicPlayerAdapter::Next()
24 {
25     m_player.SwitchNext();
26 }
27
28 void MusicPlayerAdapter::Prev()
29 {
30     // The MusicPlayer does not provide a prevSong
31     // function - so we need to skip forward until
32     // we hit the previous song.
33     size_t skipSongs = m_player.GetCount() - 1;
34
35     for (int i = 0; i < skipSongs; i++)
36         m_player.SwitchNext();
37 }
38
39 void MusicPlayerAdapter::Select(std::string const& name)
40 {
41     if (!m_player.Find(name)) std::cout << "song:␣" << name << "␣not␣found!" << std::endl;
42 }
```

6.7 MusicPlayer.hpp

```

1  /*****
2  * \file   MusicPlayer.hpp
3  * \brief  MusicPlayer - A player for music!
4  * \author  Simon Vogelhuber
5  * \date   October 2025
6  *****/
7  #ifndef MUSIC_PLAYER_HPP
8  #define MUSIC_PLAYER_HPP
9
10 #include "Object.hpp"
11 #include "Song.hpp"
12 #include <vector>
13
14 using SongCollection = std::vector<Song>;
15
16 class MusicPlayer : public Object
17 {
18 public:
19     inline static const std::string ERROR_DURATION_NULL = "ERROR:_Song_with_duration_0!";
20     inline static const std::string ERROR_EMPTY_NAME = "ERROR:_Song_with_empty_Name!";
21
22     inline static const std::size_t MAX_VOLUME = 100;
23     inline static const std::size_t MIN_VOLUME = 0;
24     inline static const std::size_t DEFAULT_VOLUME = 50;
25
26     MusicPlayer() = default;
27
28     /**
29      * \brief Plays selected song
30      */
31     void Start();
32
33     /**
34      * \brief Stop playing Song
35      */
36     void Stop();
37
38     /**
39      * \brief Skip to next song
40      */
41     void SwitchNext();
42
43     /**
44      * \brief Get index of current song
45      * \return size_t of current's song index
46      */
47     size_t const GetCurIndex() const;
48
49     /**
50      * \brief Find song by name and select it
51      * \param string name name of the Song
52      * \return true if song by that name exists
53      */
54     bool Find(std::string const& name);
55
56     /**
57      * \brief Get No. Songs inside the player
58      * \return size_t count of songs inside player
59      */
60     size_t const GetCount() const;
61
62     /**
63      * \brief Increase volume by 'vol' amount
64      * \param size_t vol (volume)
65      */
66     void IncreaseVol(size_t const vol);
67
68     /**
69      * \brief Decrease volume by 'vol' amount
70      * \param size_t vol (volume)
71      */
72     void DecreaseVol(size_t const vol);
73
74     /**
75      * \brief Add song to player
76      * \param string name
77      * \param size_t dur (duration)
78      */
79     void Add(std::string const& name, size_t const dur);
80
81     // delete Copy Ctor and Assign Operator to prohibit untested behaviour

```

```
81     MediaPlayer(MusicPlayer& Music) = delete;
82     void operator=(MusicPlayer Music) = delete;
83
84 private:
85     SongCollection m_songs;
86     size_t m_currentSongIdx = 0;
87     size_t m_volume = DEFAULT_VOLUME;
88 };
89
90
91 #endif // !MUSIC_PLAYER_HPP
```

6.8 MusicPlayer.cpp

```

1  /*****
2  * \file   MusicPlayer.hpp
3  * \brief  MusicPlayer - A player for music!
4  * \author  Simon Vogelhuber
5  * \date   October 2025
6  *****/
7  #include "MusicPlayer.hpp"
8  #include <iostream>
9
10 void MusicPlayer::Start()
11 {
12     if (std::cout.bad()) throw Object::ERROR_BAD_OSTREAM;
13
14     if (m_songs.empty())
15     {
16         std::cout << "no_songs_in_playlist!" << std::endl;
17         return;
18     }
19
20     std::cout
21     << "playing_song_number_" << m_currentSongIdx << ":\n"
22     << m_songs.at(m_currentSongIdx).GetTitle()
23     << "\n" << m_songs.at(m_currentSongIdx).GetDuration() << "\n";
24 }
25
26 void MusicPlayer::Stop()
27 {
28     if (std::cout.bad())
29         throw Object::ERROR_BAD_OSTREAM;
30
31     std::cout
32     << "stop_song_number_" << m_currentSongIdx << ":\n"
33     << m_songs.at(m_currentSongIdx).GetTitle()
34     << "\n" << m_songs.at(m_currentSongIdx).GetDuration() << "\n";
35 }
36
37 void MusicPlayer::SwitchNext()
38 {
39     // increase until end then wrap around
40     m_currentSongIdx = (m_currentSongIdx + 1) % m_songs.size();
41 }
42
43 size_t const MusicPlayer::GetCurIndex() const
44 {
45     return m_currentSongIdx;
46 }
47
48 bool MusicPlayer::Find(std::string const& name)
49 {
50     if (name.empty()) throw MusicPlayer::ERROR_EMPTY_NAME;
51
52     for (int i = 0; i < m_songs.size(); i++)
53     {
54         if (m_songs.at(i).GetTitle() == name) {
55             m_currentSongIdx = i;
56             return true;
57         }
58     }
59     return false;
60 }
61
62 size_t const MusicPlayer::GetCount() const
63 {
64     return m_songs.size();
65 }
66
67 void MusicPlayer::IncreaseVol(size_t const vol)
68 {
69     if (std::cout.bad())
70         throw Object::ERROR_BAD_OSTREAM;
71
72     m_volume += vol;
73     if (m_volume > MAX_VOLUME)
74         m_volume = MAX_VOLUME;
75
76     std::cout << "volume_is_now_" << m_volume << std::endl;
77 }
78
79 void MusicPlayer::DecreaseVol(size_t const vol)
80 {

```



```
81     if (std::cout.bad())
82         throw Object::ERROR_BAD_OSTREAM;
83
84     if (vol > m_volume)
85         m_volume = MIN_VOLUME;
86     else
87         m_volume -= vol;
88
89     std::cout << "volume_is_now->" << m_volume << std::endl;
90 }
91
92 void MediaPlayer::Add(std::string const& name, size_t const dur)
93 {
94     if (name.empty()) throw MediaPlayer::ERROR_EMPTY_NAME;
95     if (dur == 0)     throw MediaPlayer::ERROR_DURATION_NULL;
96
97     m_songs.emplace_back(name, dur);
98 }
```

6.9 Song.hpp

```
1  /*****  
2  * \file   Song.hpp  
3  * \brief  Atomic Class for saving information about a song  
4  * \author  Simon Vogelhuber  
5  * \date   October 2025  
6  *****/  
7  #ifndef SONG_HPP  
8  #define SONG_HPP  
9  
10 #include "Object.hpp"  
11  
12 class Song : public Object  
13 {  
14 public:  
15  
16     // Exceptions  
17     inline static const std::string ERROR_DURATION_NULL = "ERROR:_Song_with_duration_0!";  
18     inline static const std::string ERROR_EMPTY_NAME = "ERROR:_Song_with_empty_Name!";  
19  
20     Song(const std::string& name, const size_t& dur);  
21     /**  
22      * \brief Get title of song  
23      * \return string - title of song  
24      * \throw ERROR_DURATION_NULL  
25      * \throw ERROR_EMPTY_NAME  
26      */  
27     std::string const& GetTitle() const;  
28  
29     /**  
30      * \brief Get duration of song  
31      * \return size_t - duration of song  
32      */  
33     size_t const GetDuration() const;  
34 private:  
35     std::string m_name;  
36     size_t m_duration;  
37 };  
38  
39 #endif // !SONG_HPP
```

6.10 Song.cpp

```
1  /*****  
2  * \file   Song.cpp  
3  * \brief  Atomic Class for saving information about a song  
4  * \author  Simon Vogelhuber  
5  * \date   October 2025  
6  *****/  
7  
8  #include "Song.hpp"  
9  
10 Song::Song(const std::string& name, const size_t& dur)  
11 {  
12     if (name.empty()) throw Song::ERROR_EMPTY_NAME;  
13     if (dur == 0)      throw Song::ERROR_DURATION_NULL;  
14  
15     m_name = name;  
16     m_duration = dur;  
17 }  
18  
19  
20 std::string const& Song::GetTitle() const  
21 {  
22     return m_name;  
23 }  
24  
25 size_t const Song::GetDuration() const  
26 {  
27     return m_duration;  
28 }
```

6.11 VideoPlayerAdapter.hpp

```

1  /*****
2  * \file   VideoPlayerAdapter.hpp
3  * \brief  Adapter for the Video Player in order to Implement IPlayer Interface
4  *
5  * \author Simon
6  * \date   November 2025
7  *****/
8  #ifndef VIDEO_PLAYER_ADAPTER_HPP
9  #define VIDEO_PLAYER_ADAPTER_HPP
10
11 #include "IPlayer.hpp"
12 #include "VideoPlayer.hpp"
13
14 class VideoPlayerAdapter : public Object, public IPlayer
15 {
16 public:
17
18     /**
19     * \brief Construct a VideoPlayer Adapter .
20     *
21     * \param VidPlayer Reference to the actual VideoPlayer
22     */
23     VideoPlayerAdapter(VideoPlayer & VidPlayer) : m_player(VidPlayer) {}
24
25     /**
26     * \brief Play selected song
27     */
28     virtual void Play() override;
29
30     /**
31     * \brief increase volume by 1
32     */
33     virtual void VollInc() override;
34
35     /**
36     * \brief decrease volume by 1
37     */
38     virtual void VollDec() override;
39
40     /**
41     * \brief Stop playing Song
42     */
43     virtual void Stop() override;
44
45     /**
46     * \Skip to next song
47     */
48     virtual void Next() override;
49
50     /**
51     * \brief Skip to previous song
52     */
53     virtual void Prev() override;
54
55     /**
56     * \brief Selects a Video by Name.
57     *
58     * \param name
59     */
60     virtual void Select(std::string const& name) override;
61
62     // delete Copy Ctor and Assign Operator to prohibit untested behaviour
63     VideoPlayerAdapter(VideoPlayerAdapter& vid) = delete;
64     void operator=(VideoPlayer vid) = delete;
65
66 private:
67     VideoPlayer & m_player;
68 };
69
70 #endif // !MUSIC_PLAYER_ADAPTER_HPP

```

6.12 VideoPlayerAdapter.cpp

```
1  /*****
2  * \file   VideoPlayerAdapter.hpp
3  * \brief  Adapter for the Video Player in order to Implement IPlayer Interface
4  *
5  * \author Simon
6  * \date   November 2025
7  *****/
8  #include "VideoPlayerAdapter.hpp"
9
10 void VideoPlayerAdapter::Play() {
11     m_player.Play();
12 }
13
14 void VideoPlayerAdapter::VollInc()
15 {
16     m_player.SetVolume(m_player.GetVolume() + 1);
17 }
18
19 void VideoPlayerAdapter::VollDec()
20 {
21     if (m_player.GetVolume() != 0) {
22         m_player.SetVolume(m_player.GetVolume() - 1);
23     }
24 }
25
26 void VideoPlayerAdapter::Stop()
27 {
28     m_player.Stop();
29 }
30
31 void VideoPlayerAdapter::Next()
32 {
33     // wrap around if at the end
34     if (!m_player.Next()) {
35         m_player.First();
36     }
37 }
38
39 void VideoPlayerAdapter::Prev()
40 {
41     const size_t currIndex = m_player.CurIndex();
42     if (currIndex == 0) return;
43
44     m_player.First();
45
46     while (m_player.CurIndex() < (currIndex-1)) m_player.Next();
47 }
48
49 void VideoPlayerAdapter::Select(std::string const& name)
50 {
51     size_t prev_index = m_player.CurIndex();
52     m_player.First();
53
54     while (m_player.CurVideo() != name && m_player.Next());
55
56     if (m_player.CurVideo() != name) {
57         std::cout << "video:_" << name << "_not_found!" << std::endl;
58         // switch back to the previous Video
59         m_player.First();
60         while (prev_index != m_player.CurIndex()) m_player.Next();
61     }
62 }
63
64
65 }
```

6.13 VideoPlayer.hpp

```

1  /*****
2  * \file   VideoPlayer.hpp
3  * \brief  Implementation of Video Player of the Company DonkySoft
4  *
5  * \author Simon Offenberger
6  * \date   November 2025
7  *****/
8  #ifndef VIDEO_PLAYER_HPP
9  #define VIDEO_PLAYER_HPP
10
11 #include "Object.hpp"
12 #include "Video.hpp"
13 #include <vector>
14 #include <memory>
15 #include <iostream>
16
17 // Using definition of the container
18 using TContVids = std::vector<Video>;
19
20 class VideoPlayer : public Object {
21 public:
22     // definition of Error Messages and constance
23     inline static const std::string ERROR_NO_VIDEO_IN_COLLECTION = "ERROR:_No_video_in_Player!";
24     inline static const std::string ERROR_DURATION_NULL = "ERROR:_Video_with_duration_0!";
25     inline static const std::string ERROR_EMPTY_NAME = "ERROR:_Video_with_empty_Name!";
26
27     inline static const std::size_t MAX_VOLUME = 50;
28     inline static const std::size_t MIN_VOLUME = 0;
29     inline static const std::size_t DEFAULT_VOLUME = 8;
30
31     VideoPlayer() = default;
32
33     /**
34     * \brief Starts playing the selected Video.
35     * \throw ERROR_BAD_OSTREAM
36     * \throw ERROR_FAIL_WRITE
37     */
38     void Play() const;
39
40     /**
41     * \brief Stops the selected Video.
42     * \throw ERROR_BAD_OSTREAM
43     * \throw ERROR_FAIL_WRITE
44     */
45     void Stop() const;
46
47     /**
48     * \brief Switches to the first video in the collection.
49     *
50     * \return true -> if videos are in the playlist
51     * \return false -> no video in the playlist
52     */
53     bool First();
54
55     /**
56     * \brief Switches to the next video.
57     *
58     * \return true -> switch was successful
59     * \return false -> no switch possible index at top of playlist
60     */
61     bool Next();
62
63     /**
64     * \brief returns the current index of the selected video.
65     *
66     * \return Index of the current video
67     * \throw ERROR_NO_VIDEO_IN_COLLECTION
68     */
69     size_t CurIndex() const;
70
71     /**
72     * \brief Get the name of the current video.
73     *
74     * \return String identifier of the video
75     * \throw ERROR_NO_VIDEO_IN_COLLECTION
76     */
77     std::string CurVideo() const;
78
79     /**
80     * \brief sets the volume of the player to a specified value.

```

```
81
82     * \param vol Volume is bond to VideoPlayer::MAX_VOLUME to VideoPlayer::MIN_VOLUME
83     * \throw ERROR_BAD_OSTREAM
84     * \throw ERROR_FAIL_WRITE
85     */
86     void SetVolume(const size_t vol);
87
88     /**
89     * \brief Returns the curreunt volume of the player.
90     *
91     * \return Volume of the player
92     */
93     size_t const GetVolume() const;
94
95     /**
96     * \brief Adds a Video to the VideoPlayer.
97     *
98     * \param name Name of the Video
99     * \param dur Duration of the Video in min
100    * \param format Video Format
101    * \throw ERROR_EMPTY_NAME
102    * \throw ERROR_DURATION_NULL
103    */
104    void Add(std::string const & name, size_t const dur, EVideoFormat const & format);
105
106    // delete Copy Ctor and Assign Operator to prohibit untestet behaviour
107    VideoPlayer(VideoPlayer& vid) = delete;
108    void operator=(VideoPlayer vid) = delete;
109
110 private:
111     size_t m_volume = DEFAULT_VOLUME;
112     TContVids m_Videos;
113     size_t m_curIndex = 0;
114 };
115
116 #endif // !VIDEO_PLAYER_HPP
```

6.14 VideoPlayer.cpp

```

1  /*****
2  * \file   VideoPlayer.cpp
3  * \brief  Implementation of Video Player of the Company DonkySoft
4  *
5  * \author  Simon Offenberger
6  * \date   November 2025
7  *****/
8  #include "VideoPlayer.hpp"
9
10 void VideoPlayer::Play() const {
11     if (!std::cout.good()) throw VideoPlayer::ERROR_BAD_OSTREAM;
12     if (m_Videos.empty()) {
13         std::cout << "no_video_in_playlist!" << std::endl;
14         return;
15     }
16
17     std::cout << "playing_video_number" << CurIndex();
18     std::cout << ":\n" << CurVideo();
19     std::cout << "\n[" << m_Videos.at(m_curIndex).GetDuration() << "min]" << std::endl;
20
21     if (std::cout.fail()) throw VideoPlayer::ERROR_FAIL_WRITE;
22 }
23
24 void VideoPlayer::Stop() const {
25     if (!std::cout.good()) throw VideoPlayer::ERROR_BAD_OSTREAM;
26     if (m_Videos.empty()) {
27         std::cout << "no_video_in_playlist!" << std::endl;
28         return;
29     }
30
31     std::cout << "stop:\nvideo:\n" << CurVideo();
32     std::cout << "\n[" << m_Videos.at(m_curIndex).GetDuration() << "min]" << std::endl;
33
34     if (std::cout.fail()) throw VideoPlayer::ERROR_FAIL_WRITE;
35 }
36
37 bool VideoPlayer::First()
38 {
39     if (m_Videos.empty()) return false;
40
41     m_curIndex = 0;
42
43     return true;
44 }
45
46 bool VideoPlayer::Next()
47 {
48     m_curIndex++;
49
50     if (m_curIndex >= m_Videos.size()) {
51         m_curIndex = m_Videos.size() - 1;
52         return false;
53     }
54     else {
55         return true;
56     }
57 }
58
59 size_t VideoPlayer::CurIndex() const
60 {
61     if (m_Videos.size()==0) throw VideoPlayer::ERROR_NO_VIDEO_IN_COLLECTION;
62
63     return m_curIndex;
64 }
65
66 std::string const VideoPlayer::CurVideo() const
67 {
68     if (m_Videos.size()==0) throw VideoPlayer::ERROR_NO_VIDEO_IN_COLLECTION;
69
70     return m_Videos.at(m_curIndex).GetTitle();
71 }
72
73 void VideoPlayer::SetVolume(const size_t vol)
74 {
75     if (!std::cout.good()) throw VideoPlayer::ERROR_BAD_OSTREAM;
76
77     if (vol > MAX_VOLUME) m_volume = MAX_VOLUME;
78     else m_volume = vol;
79
80     std::cout << "volume_is_now:>\n" << m_volume;

```



```
81 |  
82 |         if (std::cout.fail()) throw VideoPlayer::ERROR_FAIL_WRITE;  
83 |     }  
84 |  
85 |     size_t const VideoPlayer::GetVolume() const  
86 |     {  
87 |         return m_volume;  
88 |     }  
89 |  
90 |     void VideoPlayer::Add(std::string const& name, size_t const dur, EVideoFormat const & format)  
91 |     {  
92 |         if (name.empty()) throw VideoPlayer::ERROR_EMPTY_NAME;  
93 |         if (dur == 0) throw VideoPlayer::ERROR_DURATION_NULL;  
94 |  
95 |         m_Videos.emplace_back(name, dur, format);  
96 |     }
```

6.15 Video.hpp

```

1  /*****
2  * \file   Video.hpp
3  * \brief  Implementation of a Video
4  *
5  * \author Simon
6  * \date   November 2025
7  *****/
8  #ifndef VIDEO_HPP
9  #define VIDEO_HPP
10
11 #include "Object.hpp"
12 #include "EVideoFormat.hpp"
13
14 class Video : public Object
15 {
16 public:
17
18     // Exceptions
19     inline static const std::string ERROR_DURATION_NULL = "ERROR:_Video_with_duration_0!";
20     inline static const std::string ERROR_EMPTY_NAME = "ERROR:_Video_with_empty_Name!";
21
22     /**
23      * \brief CTOR of a Video.
24      *
25      * \param title Tilte of the Video
26      * \param duration Duration of the Video in min
27      * \param format Video Format can be of Type EVideoFormat
28      * \throw ERROR_DURATION_NULL
29      * \throw ERROR_EMPTY_NAME
30      */
31     Video(const std::string& title, const size_t& duration, const EVideoFormat& format);
32
33     /**
34      * \brief Getter of the Video Title.
35      *
36      * \return Video Title
37      */
38     const std::string & GetTitle() const;
39
40     /**
41      * \brief Getter of the Video duration
42      *
43      * \return duration of the video
44      */
45     size_t GetDuration() const;
46
47     /**
48      * \brief Getter for the String Identifier of the Format.
49      *
50      * \return String of the Video Format
51      */
52     const std::string GetFormatID() const;
53
54 private:
55     std::string m_title;
56     size_t m_duration;
57     EVideoFormat m_format;
58 };
59
60
61 #endif // !VIDEO_HPP

```

6.16 Video.cpp

```
1  /*****  
2  * \file   Video.hpp  
3  * \brief  Implementation of a Video  
4  *  
5  * \author Simon  
6  * \date   November 2025  
7  *****/  
8  #include "Video.hpp"  
9  
10 Video::Video(const std::string& title, const size_t& duration, const EVideoFormat& format)  
11 {  
12     if (title.empty()) throw Video::ERROR_EMPTY_NAME;  
13     if (duration == 0) throw Video::ERROR_DURATION_NULL;  
14  
15     m_title = title;  
16     m_duration = duration;  
17     m_format = format;  
18 }  
19  
20 const std::string & Video::GetTitle() const  
21 {  
22     return m_title;  
23 }  
24  
25 size_t Video::GetDuration() const  
26 {  
27     return m_duration;  
28 }  
29  
30 const std::string Video::GetFormatID() const  
31 {  
32     switch (m_format) {  
33         case (EVideoFormat::AVI): return "AVI-Format";  
34         case (EVideoFormat::MKV): return "MKV-Format";  
35         case (EVideoFormat::WMV): return "WMV-Format";  
36         default: return "unkown_Format";  
37     }  
38 }
```

6.17 EVideoFormat.hpp

```
1  /*****  
2  * \file   EVideoFormat.hpp  
3  * \brief  provides an enum for the Video formats  
4  *  
5  * \author Simon  
6  * \date   November 2025  
7  *****/  
8  #ifndef EVIDEO_FORMAT_HPP  
9  #define EVIDEO_FORMAT_HPP  
10  
11  enum class EVideoFormat  
12  {  
13      AVI,  
14      MKV,  
15      WMV  
16  };  
17  
18  
19  #endif // !EVIDEO_FORMAT_HPP
```

6.18 main.cpp

```

1  #include "vld.h"
2  #include "Video.hpp"
3  #include "VideoPlayer.hpp"
4  #include "VideoPlayerAdapter.hpp"
5  #include "MusicPlayer.hpp"
6  #include "MusicPlayerAdapter.hpp"
7  #include "Client.hpp"
8  #include <iostream>
9  #include <fstream>
10 #include <cassert>
11 #include <sstream>
12 #include "Test.hpp"
13
14 using namespace std;
15
16 #define WRITE_OUTPUT true
17
18 static bool TestSong(ostream& ost);
19 static bool TestVideo(ostream& ost);
20 static bool TestVideoPlayer(ostream& ost);
21 static bool TestMusicPlayer(ostream& ost);
22
23 int main(void) {
24
25     ofstream testoutput;
26     bool TestOK = true;
27
28     try {
29
30         if (WRITE_OUTPUT == true) {
31             testoutput.open("TestOutput.txt");
32         }
33
34         VideoPlayer VPlayer;
35
36         VPlayer.Add("Harry_Potter1", 160, EVideoFormat::AVI);
37         VPlayer.Add("Harry_Potter2", 160, EVideoFormat::AVI);
38         VPlayer.Add("Harry_Potter3", 160, EVideoFormat::AVI);
39         VPlayer.Add("Harry_Potter4", 160, EVideoFormat::AVI);
40         VPlayer.Add("Harry_Potter5", 160, EVideoFormat::AVI);
41         VPlayer.Add("Harry_Potter6", 160, EVideoFormat::AVI);
42
43         VideoPlayerAdapter VidAdapter{ VPlayer };
44
45         Client client;
46
47         cout << "Test_VideoPlayer_Adapter_in_Client" << endl;
48         TestOK = TestOK && client.Test_IPlayerVolumeCTRL(cout, VidAdapter, VideoPlayer::MAX_VOLUME, VideoPlayer
::DEFAULT_VOLUME);
49         TestOK = TestOK && client.Test_IPlayerPlay(cout, VidAdapter);
50
51         if (WRITE_OUTPUT) {
52             testoutput << "Test_VideoPlayer_Adapter_in_Client" << endl;
53             TestOK = TestOK && client.Test_IPlayerVolumeCTRL(testoutput, VidAdapter, VideoPlayer::
MAX_VOLUME, VideoPlayer::DEFAULT_VOLUME);
54             TestOK = TestOK && client.Test_IPlayerPlay(testoutput, VidAdapter);
55         }
56
57         VideoPlayer EmptyPlayer;
58         VideoPlayerAdapter EmptyAdapter { EmptyPlayer };
59
60         TestOK = TestOK && client.Test_IPlayerEmptyPlay(cout, EmptyAdapter);
61         if (WRITE_OUTPUT) TestOK = TestOK && client.Test_IPlayerEmptyPlay(testoutput, EmptyAdapter);
62
63
64         MusicPlayer MPlayer;
65
66         MPlayer.Add("Harry_Potter1", 160);
67         MPlayer.Add("Harry_Potter2", 160);
68         MPlayer.Add("Harry_Potter3", 160);
69         MPlayer.Add("Harry_Potter4", 160);
70         MPlayer.Add("Harry_Potter5", 160);
71         MPlayer.Add("Harry_Potter6", 160);
72
73         MusicPlayerAdapter MusAdapter{ MPlayer };
74
75         cout << "Test_MusicPlayer_Adapter_in_Client" << endl;
76         TestOK = TestOK && client.Test_IPlayerVolumeCTRL(cout, MusAdapter, MusicPlayer::MAX_VOLUME,
MusicPlayer::DEFAULT_VOLUME);
77         TestOK = TestOK && client.Test_IPlayerPlay(cout, MusAdapter);

```

```

78         if (WRITE_OUTPUT) {
79             testoutput << "Test_MusicPlayer_Adapter_in_Client" << endl;
80             TestOK = TestOK && client.Test_IPlayerVolumeCTRL(testoutput, MusAdapter, MusicPlayer::
81                 MAX_VOLUME, MusicPlayer::DEFAULT_VOLUME);
82             TestOK = TestOK && client.Test_IPlayerPlay(testoutput, MusAdapter);
83         }
84
85         MusicPlayer EmptyMPlayer;
86         MusicPlayerAdapter EmptyMAdapter{ EmptyMPlayer };
87
88         TestOK = TestOK && client.Test_IPlayerEmptyPlay(cout, EmptyMAdapter);
89         if (WRITE_OUTPUT) TestOK = TestOK && client.Test_IPlayerEmptyPlay(testoutput, EmptyMAdapter);
90
91
92         TestOK = TestOK && TestSong(cout);
93         if (WRITE_OUTPUT) TestOK = TestOK && TestSong(testoutput);
94
95         TestOK = TestOK && TestVideo(cout);
96         if (WRITE_OUTPUT) TestOK = TestOK && TestVideo(testoutput);
97
98         TestOK = TestOK && TestVideoPlayer(cout);
99         if (WRITE_OUTPUT) TestOK = TestOK && TestVideoPlayer(testoutput);
100
101         TestOK = TestOK && TestMusicPlayer(cout);
102         if (WRITE_OUTPUT) TestOK = TestOK && TestMusicPlayer(testoutput);
103
104         if (WRITE_OUTPUT) {
105             if (TestOK) TestCaseOK(testoutput);
106             else TestCaseFail(testoutput);
107
108             testoutput.close();
109         }
110
111         if (TestOK) TestCaseOK(cout);
112         else TestCaseFail(cout);
113
114     }
115     catch (const string& err) {
116         cerr << err;
117     }
118     catch (bad_alloc const& error) {
119         cerr << error.what();
120     }
121     catch (const exception& err) {
122         cerr << err.what();
123     }
124     catch (...) {
125         cerr << "Unhandelt_Exception";
126     }
127
128     if (testoutput.is_open()) testoutput.close();
129
130     return 0;
131 }
132
133 bool TestSong(ostream& ost)
134 {
135     assert(ost.good());
136
137     TestStart(ost);
138
139     bool TestOK = true;
140     string error_msg = "";
141
142     try {
143
144         Song HelloWorld("Hello_World", 123);
145
146         TestOK = TestOK && check_dump(ost, "Test_Song_Getter_Duration", static_cast<size_t>(123), HelloWorld.
            GetDuration());
147
148         TestOK = TestOK && check_dump(ost, "Test_Song_Getter_Name", static_cast<string>("Hello_World"),
            HelloWorld.GetTitle());
149
150     }
151     catch (const string& err) {
152         error_msg = err;
153     }
154     catch (bad_alloc const& error) {
155         error_msg = error.what();
156     }
157     catch (const exception& err) {

```

```

158         error_msg = err.what();
159     }
160     catch (...) {
161         error_msg = "Unhandelt_Exception";
162     }
163
164     TestOK = TestOK && check_dump(ost, "Check_for_Exception_in_Testcase", true, error_msg.empty());
165     error_msg.clear();
166
167     try {
168         Song song{ "HelloWorld", 0};
169     }
170     catch (const string& err) {
171         error_msg = err;
172     }
173     catch (bad_alloc const& error) {
174         error_msg = error.what();
175     }
176     catch (const exception& err) {
177         error_msg = err.what();
178     }
179     catch (...) {
180         error_msg = "Unhandelt_Exception";
181     }
182
183     TestOK = TestOK && check_dump(ost, "Test_Exception_in_Song_CTOR_with_duration_0", error_msg, Song::
        ERROR_DURATION_NULL);
184     error_msg.clear();
185
186     try {
187
188         Song song{ "", 12};
189
190     }
191     catch (const string& err) {
192         error_msg = err;
193     }
194     catch (bad_alloc const& error) {
195         error_msg = error.what();
196     }
197     catch (const exception& err) {
198         error_msg = err.what();
199     }
200     catch (...) {
201         error_msg = "Unhandelt_Exception";
202     }
203
204     TestOK = TestOK && check_dump(ost, "Test_Exception_in_Song_CTOR_with_empty_string", error_msg, Song::
        ERROR_EMPTY_NAME);
205     error_msg.clear();
206
207     TestEnd(ost);
208     return TestOK;
209 }
210
211 bool TestVideo(ostream& ost)
212 {
213     assert(ost.good());
214
215     TestStart(ost);
216
217     bool TestOK = true;
218     string error_msg = "";
219
220     try {
221
222         Video HelloWorld("HelloWorld", 123, EVideoFormat::AVI);
223
224         TestOK = TestOK && check_dump(ost, "Test_Song_Getter_Duration", static_cast<size_t>(123), HelloWorld.
            GetDuration());
225
226         TestOK = TestOK && check_dump(ost, "Test_Song_Getter_Name", static_cast<string>("HelloWorld"),
            HelloWorld.GetTitle());
227
228         TestOK = TestOK && check_dump(ost, "Test_Song_Getter_Format", static_cast<string>("AVI-Format"),
            HelloWorld.GetFormatID());
229
230     }
231     catch (const string& err) {
232         error_msg = err;
233     }
234     catch (bad_alloc const& error) {
235         error_msg = error.what();

```

```

236     }
237     catch (const exception& err) {
238         error_msg = err.what();
239     }
240     catch (...) {
241         error_msg = "Unhandelt_Exception";
242     }
243
244     TestOK = TestOK && check_dump(ost, "Check_for_Exception_in_Testcase", true, error_msg.empty());
245     error_msg.clear();
246
247     try {
248         Video vid{ "Hello_World",0, EVideoFormat::AVI };
249     }
250     catch (const string& err) {
251         error_msg = err;
252     }
253     catch (bad_alloc const& error) {
254         error_msg = error.what();
255     }
256     catch (const exception& err) {
257         error_msg = err.what();
258     }
259     catch (...) {
260         error_msg = "Unhandelt_Exception";
261     }
262
263     TestOK = TestOK && check_dump(ost, "Test_Exception_in_Video_CTOR_with_duration_0", error_msg, Video::
        ERROR_DURATION_NULL);
264     error_msg.clear();
265
266     try {
267         Video vid{ "",12,EVideoFormat::MKV };
268     }
269
270     catch (const string& err) {
271         error_msg = err;
272     }
273
274     catch (bad_alloc const& error) {
275         error_msg = error.what();
276     }
277     catch (const exception& err) {
278         error_msg = err.what();
279     }
280     catch (...) {
281         error_msg = "Unhandelt_Exception";
282     }
283
284     TestOK = TestOK && check_dump(ost, "Test_Exception_in_Video_CTOR_with_empty_string", error_msg, Video::
        ERROR_EMPTY_NAME);
285     error_msg.clear();
286
287     TestEnd(ost);
288     return TestOK;
289 }
290
291 bool TestVideoPlayer(ostream& ost)
292 {
293     assert(ost.good());
294
295     TestStart(ost);
296
297     bool TestOK = true;
298     string error_msg = "";
299
300     try {
301         Video HelloWorld("Hello_World", 123, EVideoFormat::AVI);
302
303         VideoPlayer VPlayer;
304
305         VPlayer.Add("Hello_World1",123,EVideoFormat::AVI);
306         VPlayer.Add("Hello_World2",124,EVideoFormat::MKV);
307         VPlayer.Add("Hello_World3",125,EVideoFormat::WMV);
308         VPlayer.Add("Hello_World4",126,EVideoFormat::AVI);
309         VPlayer.Add("Hello_World5",127,EVideoFormat::MKV);
310
311         TestOK = TestOK && check_dump(ost, "Test_Videoplayer_Initial_Index", static_cast<size_t>(0), VPlayer.
            CurIndex());
312
313         VPlayer.First();
314
315

```



```

316
317     TestOK = TestOK && check_dump(ost, "Test_Videoplayer_Index_after_First", static_cast<size_t>(0),
318                                     VPlayer.CurIndex());
319
320     VPlayer.Next();
321
322     TestOK = TestOK && check_dump(ost, "Test_Videoplayer_Index_after_Next", static_cast<size_t>(1),
323                                     VPlayer.CurIndex());
324
325     for (int i = 0; i < 100; i++) VPlayer.Next();
326
327     TestOK = TestOK && check_dump(ost, "Test_Videoplayer_Index_Upper_Bound", static_cast<size_t>(4),
328                                     VPlayer.CurIndex());
329
330     VPlayer.First();
331
332     TestOK = TestOK && check_dump(ost, "Test_Videoplayer_Index_after_First", static_cast<size_t>(0),
333                                     VPlayer.CurIndex());
334
335     TestOK = TestOK && check_dump(ost, "Test_Default_Volume", static_cast<size_t>(8), VPlayer.GetVolume
336                                     ());
337
338     std::stringstream* coutbuf = std::cout.rdbuf();
339
340     stringstream result;
341
342     // cout redirect to stringstream
343     std::cout.rdbuf(result.rdbuf());
344
345     VPlayer.SetVolume(25);
346
347     std::cout.rdbuf(coutbuf);
348
349     TestOK = TestOK && check_dump(ost, "Test_Set_Volume", static_cast<size_t>(25), VPlayer.GetVolume());
350
351     // cout redirect to stringstream
352     std::cout.rdbuf(result.rdbuf());
353
354     VPlayer.SetVolume(300);
355
356     std::cout.rdbuf(coutbuf);
357
358     TestOK = TestOK && check_dump(ost, "Test_Set_Volume_Max_Volume", static_cast<size_t>(VideoPlayer::
359                                     MAX_VOLUME), VPlayer.GetVolume());
360
361     // cout redirect to stringstream
362     std::cout.rdbuf(result.rdbuf());
363
364     VPlayer.SetVolume(0);
365
366     std::cout.rdbuf(coutbuf);
367
368     TestOK = TestOK && check_dump(ost, "Test_Set_Volume_Min_Volume", static_cast<size_t>(VideoPlayer::
369                                     MIN_VOLUME), VPlayer.GetVolume());
370
371     result.str("");
372     result.clear();
373     // cout redirect to stringstream
374     std::cout.rdbuf(result.rdbuf());
375
376     VPlayer.Play();
377
378     std::cout.rdbuf(coutbuf);
379
380     TestOK = TestOK && check_dump(ost, "Test_Video_Player_Play", true, result.str().find(VPlayer.
381                                     CurVideo()) != string::npos);
382
383     result.str("");
384     result.clear();
385     // cout redirect to stringstream
386     std::cout.rdbuf(result.rdbuf());
387
388     VPlayer.Stop();
389
390     std::cout.rdbuf(coutbuf);
391
392     TestOK = TestOK && check_dump(ost, "Test_Video_Player_Stop", true, result.str().find("stop") !=
393                                     string::npos);
394
395     }
396
397     catch (const string& err) {
398         error_msg = err;
399     }

```

```

390         catch (bad_alloc const& error) {
391             error_msg = error.what();
392         }
393         catch (const exception& err) {
394             error_msg = err.what();
395         }
396         catch (...) {
397             error_msg = "Unhandelt_Exception";
398         }
399
400         TestOK = TestOK && check_dump(ost, "Check_for_Exception_in_Testcase", true, error_msg.empty());
401         error_msg.clear();
402
403         try{
404             MediaPlayer VidPlayer;
405             VidPlayer.Add("", 123, EVideoFormat::AVI);
406
407         }
408         catch (const string& err) {
409             error_msg = err;
410         }
411         catch (bad_alloc const& error) {
412             error_msg = error.what();
413         }
414         catch (const exception& err) {
415             error_msg = err.what();
416         }
417         catch (...) {
418             error_msg = "Unhandelt_Exception";
419         }
420
421         TestOK = TestOK && check_dump(ost, "Test_Exception_in_Add_with_empty_string", error_msg, MediaPlayer::
            ERROR_EMPTY_NAME);
422         error_msg.clear();
423
424         try{
425             MediaPlayer VidPlayer;
426             VidPlayer.Add("234", 0, EVideoFormat::AVI);
427
428         }
429         catch (const string& err) {
430             error_msg = err;
431         }
432         catch (bad_alloc const& error) {
433             error_msg = error.what();
434         }
435         catch (const exception& err) {
436             error_msg = err.what();
437         }
438         catch (...) {
439             error_msg = "Unhandelt_Exception";
440         }
441
442         TestOK = TestOK && check_dump(ost, "Test_Exception_in_Add_with_empty_string", error_msg, MediaPlayer::
            ERROR_DURATION_NULL);
443         error_msg.clear();
444
445
446         TestEnd(ost);
447         return TestOK;
448     }
449
450     bool TestMediaPlayer(ostream& ost)
451     {
452         assert(ost.good());
453
454         TestStart(ost);
455
456         bool TestOK = true;
457         string error_msg = "";
458
459         // test basic functionality
460         try {
461             MediaPlayer music;
462             std::string const song1 = "How_much_is_the_Fish_-_Scooter";
463             std::string const song2 = "Die_Blume_aus_dem_Gemeindebau_-_Wolfgang_Ambros";
464             std::string const song3 = "Red_Sun_in_the_Sky_-_MaoZe";
465             std::string const song4 = "Ski-Twist_-_Hansi_Hinterseer";
466             size_t const dur1 = 300;
467             size_t const dur2 = 240;
468             size_t const dur3 = 180;
469             size_t const dur4 = 110;
470             size_t const songCount = 4;

```

```

471         music.Add(song1, dur1);
472         music.Add(song2, dur2);
473         music.Add(song3, dur3);
474         music.Add(song4, dur4);
475
476         TestOK = TestOK && check_dump(ost, "MediaPlayer_Basic_Functionality_GetCount()", music.GetCount(
477             ), songCount);
478         TestOK = TestOK && check_dump(ost, "MediaPlayer_Basic_Functionality_GetIndex()_initial", music.
479             GetCurIndex(), static_cast<size_t>(0));
480         TestOK = TestOK && check_dump(ost, "MediaPlayer_Basic_Functionality_Find()_unknown_song", music
481             .Find("not_a_real_song"), false);
482         TestOK = TestOK && check_dump(ost, "MediaPlayer_Basic_Functionality_Find()_song_that_exists",
483             music.Find(song1), true);
484
485         // for checking cout
486         std::streambuf* coutbuf = std::cout.rdbuf();
487         stringstream result;
488
489         // cout redirect to stringstream
490         std::cout.rdbuf(result.rdbuf());
491         music.Start();
492         std::cout.rdbuf(coutbuf);
493
494         TestOK = TestOK && check_dump(ost, "MediaPlayer_Basic_Functionality_Song_name_after_initial_
495             Start()", true, result.str().find(song1) != string::npos);
496         result.str("");
497         result.clear();
498
499         music.SwitchNext();
500         std::cout.rdbuf(result.rdbuf());
501         music.Start();
502         std::cout.rdbuf(coutbuf);
503
504         TestOK = TestOK && check_dump(ost, "MediaPlayer_Basic_Functionality_GetIndex()_after_switching"
505             , static_cast<size_t>(1), music.GetCurIndex());
506         TestOK = TestOK && check_dump(ost, "MediaPlayer_Basic_Functionality_Song_name_switching", true,
507             result.str().find(song2) != string::npos);
508         result.str("");
509         result.clear();
510
511         // wrap around
512         for (int i = 0; i < music.GetCount(); i++)
513         {
514             music.SwitchNext();
515         }
516
517         std::cout.rdbuf(result.rdbuf());
518         music.Stop();
519         std::cout.rdbuf(coutbuf);
520
521         TestOK = TestOK && check_dump(ost, "MediaPlayer_Basic_Functionality_GetIndex()_wrap_around",
522             static_cast<size_t>(1), music.GetCurIndex());
523     }
524     catch (const string& err) {
525         error_msg = err;
526     }
527     catch (bad_alloc const& error) {
528         error_msg = error.what();
529     }
530     catch (const exception& err) {
531         error_msg = err.what();
532     }
533     catch (...) {
534         error_msg = "Unhandelt_Exception";
535     }
536
537     TestOK = TestOK && check_dump(ost, "MediaPlayer_Basic_Functionality_Error_Buffer", true, error_msg.empty
538         ());
539     error_msg.clear();
540
541     // Add empty song
542     try {
543         MediaPlayer music;
544         std::string const song = "";
545         size_t const dur = 1;
546         music.Add(song, dur);
547     }
548     catch (const string& err) {
549         error_msg = err;
550     }
551     catch (bad_alloc const& error) {
552         error_msg = error.what();
553     }
554 }

```

```
545         catch (const exception& err) {
546             error_msg = err.what();
547         }
548         catch (...) {
549             error_msg = "Unhandelt_Exception";
550         }
551     }
552     TestOK = TestOK && check_dump(ost, "MediaPlayer_~AddSong_without_title", MediaPlayer::ERROR_EMPTY_NAME,
553                                 error_msg);
554     error_msg.clear();
555     // Add song with 0 duration
556     try {
557         MediaPlayer music;
558         std::string const song = "This_is_a_legit_song";
559         size_t const dur = 0;
560         music.Add(song, dur);
561     }
562     catch (const string& err) {
563         error_msg = err;
564     }
565     catch (bad_alloc const& error) {
566         error_msg = error.what();
567     }
568     catch (const exception& err) {
569         error_msg = err.what();
570     }
571     catch (...) {
572         error_msg = "Unhandelt_Exception";
573     }
574
575     TestOK = TestOK && check_dump(ost, "MediaPlayer_~AddSong_without_title", MediaPlayer::ERROR_DURATION_NULL,
576                                 error_msg);
577     error_msg.clear();
578     // find empty name
579     try {
580         MediaPlayer music;
581         music.Find("");
582     }
583     catch (const string& err) {
584         error_msg = err;
585     }
586     catch (bad_alloc const& error) {
587         error_msg = error.what();
588     }
589     catch (const exception& err) {
590         error_msg = err.what();
591     }
592     catch (...) {
593         error_msg = "Unhandelt_Exception";
594     }
595
596     TestOK = TestOK && check_dump(ost, "MediaPlayer_~AddSong_without_title", MediaPlayer::ERROR_EMPTY_NAME,
597                                 error_msg);
598     error_msg.clear();
599     return TestOK;
600 }
```

6.19 Test.hpp

```

1  /*****
2  * \file   Test.hpp
3  * \brief  File that provides a Test Function with a formatted output
4  *
5  * \author Simon
6  * \date   April 2025
7  *****/
8  #ifndef TEST_HPP
9  #define TEST_HPP
10
11 #include <string>
12 #include <iostream>
13 #include <vector>
14 #include <list>
15 #include <queue>
16 #include <forward_list>
17
18 #define ON 1
19 #define OFF 0
20 #define COLOR_OUTPUT OFF
21
22 // Definitions of colors in order to change the color of the output stream.
23 const std::string colorRed = "\x1B[31m";
24 const std::string colorGreen = "\x1B[32m";
25 const std::string colorWhite = "\x1B[37m";
26
27 inline std::ostream& RED(std::ostream& ost) {
28     if (ost.good()) {
29         ost << colorRed;
30     }
31     return ost;
32 }
33 inline std::ostream& GREEN(std::ostream& ost) {
34     if (ost.good()) {
35         ost << colorGreen;
36     }
37     return ost;
38 }
39 inline std::ostream& WHITE(std::ostream& ost) {
40     if (ost.good()) {
41         ost << colorWhite;
42     }
43     return ost;
44 }
45
46 inline std::ostream& TestStart(std::ostream& ost) {
47     if (ost.good()) {
48         ost << std::endl;
49         ost << "*****" << std::endl;
50         ost << "~~~~~TESTCASE_START~~~~~" << std::endl;
51         ost << "*****" << std::endl;
52         ost << std::endl;
53     }
54     return ost;
55 }
56
57 inline std::ostream& TestEnd(std::ostream& ost) {
58     if (ost.good()) {
59         ost << std::endl;
60         ost << "*****" << std::endl;
61         ost << std::endl;
62     }
63     return ost;
64 }
65
66 inline std::ostream& TestCaseOK(std::ostream& ost) {
67     #if COLOR_OUTPUT
68         if (ost.good()) {
69             ost << colorGreen << "TEST_OK!!" << colorWhite << std::endl;
70         }
71     #else
72         if (ost.good()) {
73             ost << "TEST_OK!!" << std::endl;
74         }
75     #endif // COLOR_OUTPUT
76
77     return ost;
78 }
79
80

```

```

81 inline std::ostream& TestCaseFail(std::ostream& ost) {
82
83 #if COLOR_OUTPUT
84     if (ost.good()) {
85         ost << colorRed << "TEST_FAILED_!!" << colorWhite << std::endl;
86     }
87 #else
88     if (ost.good()) {
89         ost << "TEST_FAILED_!!" << std::endl;
90     }
91 #endif // COLOR_OUTPUT
92
93     return ost;
94 }
95
96 /**
97  * \brief function that reports if the testcase was successful.
98  *
99  * \param testcase      String that indicates the testcase
100  * \param successful true -> reports to cout test OK
101  * \param successful false -> reports test failed
102  */
103 template <typename T>
104 bool check_dump(std::ostream& ostr, const std::string& testcase, const T& expected, const T& result) {
105     if (ostr.good()) {
106 #if COLOR_OUTPUT
107         if (expected == result) {
108             ostr << testcase << std::endl << colorGreen << "[Test_OK]_" << colorWhite << "Result:_"
109             << Expected:_" << std::boolalpha << expected << "_==" << "_Result:_" << result << ")" <<
110             std::noboolalpha << std::endl << std::endl;
111         }
112         else {
113             ostr << testcase << std::endl << colorRed << "[Test_FAILED]_" << colorWhite << "Result:_"
114             << Expected:_" << std::boolalpha << expected << "_!=" << "_Result:_" << result << ")" <<
115             std::noboolalpha << std::endl << std::endl;
116         }
117 #else
118         if (expected == result) {
119             ostr << testcase << std::endl << "[Test_OK]_" << "Result:_(Expected:_" << std::boolalpha <<
120             expected << "_==" << "_Result:_" << result << ")" << std::noboolalpha << std::endl <<
121             std::endl;
122         }
123         else {
124             ostr << testcase << std::endl << "[Test_FAILED]_" << "Result:_(Expected:_" << std::
125             boolalpha << expected << "_!=" << "_Result:_" << result << ")" << std::noboolalpha <<
126             std::endl << std::endl;
127         }
128 #endif
129
130         if (ostr.fail()) {
131             std::cerr << "Error:_Write_Ostream" << std::endl;
132         }
133         else {
134             std::cerr << "Error:_Bad_Ostream" << std::endl;
135         }
136     }
137     return expected == result;
138 }
139
140 template <typename T1, typename T2>
141 std::ostream& operator<< (std::ostream& ost, const std::pair<T1,T2> & p) {
142     if (!ost.good()) throw std::exception{ "Error:_bad_Ostream!" };
143     ost << "(" << p.first << ", " << p.second << ")";
144     return ost;
145 }
146
147 template <typename T>
148 std::ostream& operator<< (std::ostream& ost, const std::vector<T> & cont) {
149     if (!ost.good()) throw std::exception{ "Error:_bad_Ostream!" };
150     std::copy(cont.cbegin(), cont.cend(), std::ostream_iterator<T>(ost, "_"));
151     return ost;
152 }
153
154 template <typename T>
155 std::ostream& operator<< (std::ostream& ost, const std::list<T> & cont) {
156     if (!ost.good()) throw std::exception{ "Error:_bad_Ostream!" };
157     std::copy(cont.cbegin(), cont.cend(), std::ostream_iterator<T>(ost, "_"));
158     return ost;
159 }
160
161 template <typename T>
162 std::ostream& operator<< (std::ostream& ost, const std::deque<T> & cont) {
163     if (!ost.good()) throw std::exception{ "Error:_bad_Ostream!" };
164     std::copy(cont.cbegin(), cont.cend(), std::ostream_iterator<T>(ost, "_"));
165     return ost;
166 }

```

```
156         if (!ost.good()) throw std::exception{ "Error_bad_Ostream!" };
157         std::copy(cont.cbegin(), cont.cend(), std::ostream_iterator<T>(ost, "\n"));
158         return ost;
159     }
160
161     template <typename T>
162     std::ostream& operator<< (std::ostream& ost, const std::forward_list<T> & cont) {
163         if (!ost.good()) throw std::exception{ "Error_bad_Ostream!" };
164         std::copy(cont.cbegin(), cont.cend(), std::ostream_iterator<T>(ost, "\n"));
165         return ost;
166     }
167
168 #endif // !TEST_HPP
169
```