

FH-OÖ Hagenberg/HSD
SDP3, WS 2025
Übung 6



Name: Simon Offenberger / Simon Vogelhuber

Aufwand in h: siehe Doku.

Mat.Nr: S2410306027 / S2410306014

Punkte:

Übungsgruppe: 1

korrigiert:

Beispiel 1 (24 Punkte) Dateisystem-Simulation: Entwerfen Sie aus der nachfolgenden Spezifikation ein Klassendiagramm, instanzieren Sie dieses und implementieren Sie die Funktionalität entsprechend:

Ein Dateisystem für ein einfaches, eingebettetes System besteht aus Dateien, Ordner und Verweise auf Dateien, Ordner oder weitere Verweise. Ein Ordner kann Dateien, Verweise und weitere Ordner beinhalten. Dateien, Ordner und Verweise werden über einen Namen spezifiziert, der verändert werden kann.

Eine Datei hat zusätzlich folgende Eigenschaften:

- aktuelle Dateigröße in Bytes
- Größe eines Blockes auf dem Speichermedium in Bytes
- Anzahl der reservierten Blöcke

Die Größe eines Blockes und die Anzahl der reservierten Blöcke kann für jede Datei bei der Erzeugung unterschiedlich festgelegt werden. Ein nachträgliches Ändern dieser Eigenschaften ist nicht möglich!

Das Schreiben in eine Datei wird durch eine Methode `Write(size_t const bytes)` simuliert. Achten Sie darauf, dass die Datei nicht größer werden kann als der für die Datei reservierte Speicher!

Implementieren Sie zur Erzeugung von Dateien, Ordner und Verweise eine einfache Fabrik.

Implementieren Sie einen Visitor (`Dump`) der alle Dateien, Verweise und Ordner in hierarchischer Form ausgibt. Die Ausgabe soll sowohl auf der Standardausgabe als auch in einer Datei möglich sein!

Implementieren Sie einen Visitor (`FilterFiles`) der alle Dateien herausfiltert deren aktuelle Größe innerhalb eines vorgegebenen minimalen und maximalen Wertes liegt. Ein zusätzlicher Filter soll alle Verweise herausfiltern. Die Filter sollen in der Lage sein, alle gefilterten Dateien mit ihrem vollständigen Pfadnamen auszugeben! Bei der Filterung von Verweisen muss zusätzlich auch der

Name des Elementes auf das verwiesen wird ausgegeben werden.

Implementieren Sie einen Testtreiber der ein hierarchisches Dateisystem mit mehreren Ebenen erzeugt und die zu implementierenden Besucher ausführlich testet!

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!



Systemdokumentation Projekt Filesystem

Version 1.0

S. Offenberger, S. Vogelhuber

Hagenberg, 9. Dezember 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 Systemüberblick	8
2.2 Funktionale Anforderungen	8
2.2.1 Dateien	8
2.2.2 Ordner	9
2.2.3 Verweise	9
2.3 Erzeugung der Elemente	9
2.4 Besucher (Visitor) Anforderungen	10
2.4.1 Visitor: Dump	10
2.4.2 Visitor: FilterFiles	10
3 Systementwurf	11
3.1 Klassendiagramm	11
3.2 Designentscheidungen	12
3.3 Composite Pattern	12
3.3.1 Copy Ctor und Assignment Operator	13
3.4 Factory Pattern	13
3.5 Visitor Pattern	13
3.6 Template Methode Pattern	14
4 Dokumentation der Komponenten (Klassen)	14
5 Testprotokollierung	15
6 Quellcode	28
6.1 Object.hpp	28
6.2 FSOBJECTFACTORY.hpp	29
6.3 FSOBJECTFACTORY.cpp	30
6.4 Filesystem.hpp	31
6.5 Filesystem.cpp	32

6.6	FSObject.hpp	34
6.7	FSObject.cpp	36
6.8	File.hpp	37
6.9	File.cpp	38
6.10	IFolder.hpp	39
6.11	Folder.hpp	40
6.12	Folder.cpp	42
6.13	ILink.hpp	44
6.14	Link.hpp	45
6.15	Link.cpp	46
6.16	IVisitor.hpp	47
6.17	FilterVisitor.hpp	48
6.18	FilterVisitor.cpp	50
6.19	FilterFileVisitor.hpp	52
6.20	FilterFileVisitor.cpp	53
6.21	FilterLinkVisitor.hpp	54
6.22	FilterLinkVisitor.cpp	55
6.23	DumpVisitor.hpp	56
6.24	DumpVisitor.cpp	57
6.25	main.cpp	58
6.26	Test.hpp	82

1 Organisatorisches

1.1 Team

- Simon Offenberger, Matr.-Nr.: S2410306027, E-Mail: Simon.Offenberger@fh-hagenberg.at
- Simon Vogelhuber, Matr.-Nr.: S2410306014, E-Mail: Simon.Vogelhuber@fh-hagenberg.at

1.2 Aufteilung der Verantwortlichkeitsbereiche

- Simon Offenberger
 - Design Klassendiagramm
 - Implementierung und Test der Klassen:
 - * IVisitor,
 - * FilterVisitor,
 - * FilterFileVisitor,
 - * FilterLinkVisitor,
 - * DumpVisitor und
 - * FSObjectFactory
 - Implementierung des Testtreibers
 - Dokumentation
- Simon Vogelhuber
 - Design Klassendiagramm

- Implementierung und Komponententest der Klassen:
 - * FSObject
 - * File,
 - * iFolder,
 - * iLink,
 - * Folder und
 - * Link
- Implementierung des Testtreibers
- Dokumentation

1.3 Aufwand

- Simon Offenberger: geschätzt 10 Ph / tatsächlich 12 Ph
- Simon Vogelhuber: geschätzt 8 Ph / tatsächlich 8 Ph

2 Anforderungsdefinition (Systemspezifikation)

Das zu entwickelnde System dient der Simulation eines einfachen Dateisystems für ein eingebettetes System. Ziel ist es, die Struktur und das Verhalten eines hierarchischen Dateisystems softwaretechnisch abzubilden und durch geeignete Entwurfsmuster (Composite, Factory, Visitor) erweiterbar und wartbar zu gestalten. Die Anforderungen ergeben sich aus der gegebenen Systemspezifikation der Übung.

2.1 Systemüberblick

Das System verwaltet drei Arten von Dateisystemelementen:

- **Dateien**
- **Ordner**
- **Verweise** (Referenzen auf Dateien, Ordner oder weitere Verweise)

Diese Elemente bilden gemeinsam eine hierarchische Struktur, in der Ordner beliebige Kombinationen dieser Elemente enthalten können. Jedes Element besitzt einen Namen, der nachträglich veränderbar ist.

2.2 Funktionale Anforderungen

2.2.1 Dateien

Eine Datei verfügt über folgende unveränderliche Eigenschaften, die bei ihrer Erzeugung festgelegt werden:

- Blockgröße auf dem Speichermedium (Bytes)
- Anzahl reservierter Blöcke

Zusätzlich wird die aktuelle Dateigröße in Bytes verwaltet. Das Schreiben in eine Datei erfolgt über:

- `Write(size_t const bytes)`

Die Datei darf niemals größer werden als der durch die reservierten Blöcke bereitgestellte Speicher.

2.2.2 Ordner

Ein Ordner kann beliebig viele Dateien, Verweise und weitere Ordner enthalten. Er bildet die Grundlage des hierarchischen Dateisystems.

2.2.3 Verweise

Ein Verweis referenziert exakt ein Zielobjekt (Datei, Ordner oder weiteren Verweis). Der Name des Verweises kann verändert werden, zusätzlich muss der Name des Zielobjekts im Rahmen der Filterausgabe ausgegeben werden.

2.3 Erzeugung der Elemente

Für die Erstellung aller Dateisystemelemente ist eine einfache **Fabrik** zu implementieren. Diese kapselt die Instanziierungslogik und stellt sicher, dass die Objekterzeugung einheitlich erfolgt.

2.4 Besucher (Visitor) Anforderungen

2.4.1 Visitor: Dump

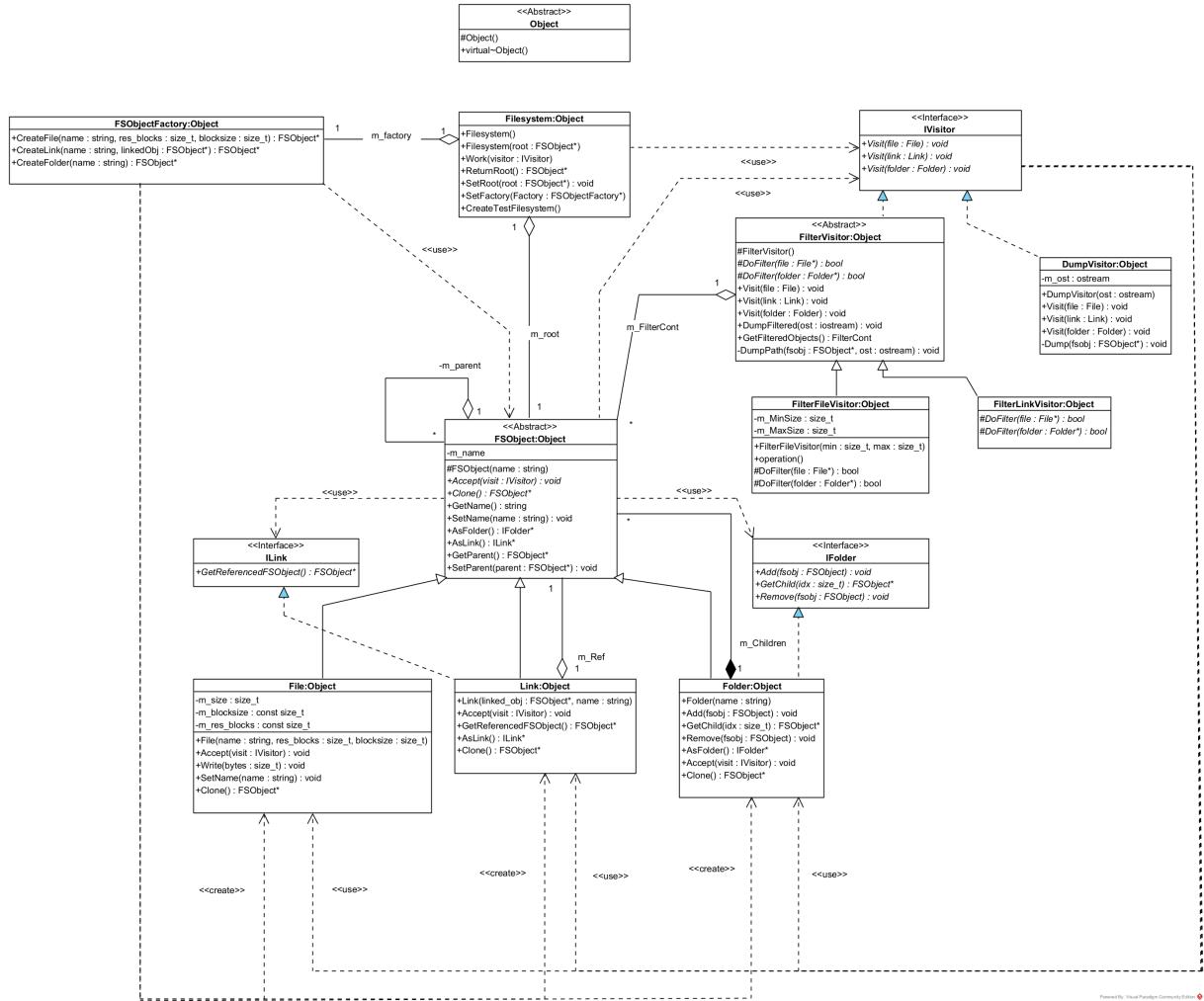
- Gibt die gesamte Dateisystemhierarchie aus.
- Ausgabe sowohl auf der Standardausgabe als auch in einer Datei möglich.
- Muss Dateien, Ordner und Verweise in strukturierter Form darstellen.

2.4.2 Visitor: FilterFiles

- Filtert Dateien anhand eines minimalen und maximalen Größenschwellwerts.
- Ausgabe aller gefilterten Dateien mit ihrem vollständigen Pfad.
- Bei Verweisen muss zusätzlich der Name des referenzierten Zielobjekts ausgegeben werden.

3 Systementwurf

3.1 Klassendiagramm



3.2 Designentscheidungen

Aus der Aufgabenstellung lassen sich folgenden Designpattern ableiten:

- Composite Pattern für die hierarchische Struktur des Dateisystems.
- Factory Pattern für die einheitliche Objekterzeugung der Dateisystemelemente.
- Visitor Pattern für die Implementierung der verschiedenen Besucheroperationen.
- Template Methode Pattern für die gemeinsame Struktur der Filter Visitor.

3.3 Composite Pattern

Dieses Pattern wird verwendet, um die hierarchische Struktur des Dateisystems abzubilden. Die Basisklasse `FSObject` definiert die gemeinsamen Schnittstellen für alle Dateisystemelemente.

Ordner implementieren die Fähigkeit, andere `FSObject`-Instanzen zu enthalten (wie Dateien, Verweise und weitere Ordner), wodurch eine Baumstruktur entsteht.

Bei der gewählten Implementierung wurde besonders darauf geachtet, dass das Liskovsersche Substitutionsprinzip eingehalten wird. Aus diesem Grund wurden die Methoden zur Verwaltung von Kindobjekten nur in der `Folder`-Klasse implementiert. Die Schnittstelle für die Methoden der besonderen Kindklassen wurden in `capabiltiy Interfaces` ausgelagert (`IFolder`, `ILink`).

Dadurch wird verhindert, dass Objekte, die keine Kinder enthalten können (wie Dateien und Verweise), diese Methoden erben und somit das Substitutionsprinzip verletzen.

3.3.1 Copy Ctor und Assignment Operator

Für die Klassen File und Link ist der Default Copy Constructor und Assignment Operator ausreichend. Für die Klasse Folder wurde der Copy Constructor und Assignment Operator überschrieben, um eine tiefe Kopie der enthaltenen Kindobjekte zu gewährleisten. Dadurch wird sichergestellt, dass bei der Kopie eines Ordners alle enthaltenen Objekte ebenfalls kopiert werden, anstatt nur Referenzen auf die Originalobjekte zu übernehmen. Dies verhindert unerwartete Seiteneffekte bei der Manipulation von Ordner und ihren Inhalten. Weiters ist darauf zu achten, dass bei der Implementierung des Copy Constructors und Assignment Operators auch die Parent Beziehung der Kindobjekte korrekt gesetzt wird.

Der Destruktor der Klassen muss nicht überschrieben werden, da ausschließlich mit Smart Pointern gearbeitet wird.

3.4 Factory Pattern

Für die konkrete Implementierung der Objekterzeugung wurde das Pattern Simple Factory verwendet. Die Klasse `FSObjectFactory` kapselt die Logik zur Erstellung von Dateien, Ordner und Verweisen. Dies ermöglicht eine zentrale Verwaltung der Erzeugungslogik und erleichtert zukünftige Erweiterungen. Beim konkreten Design der Factory wurde auf das Interface zwischen Factory und Client verzichtet, da die Factory nur eine einzige Implementierung besitzt und keine weiteren Varianten geplant sind.

Dadurch wurde die Komplexität reduziert, jedoch bleibt die Erfüllung des Dependency Inversion Prinzips aus. Dies ist aber über die Verwendung der Simple Factory hinweg vertretbar.

(Dies wurde mit Prof. Wiesinger diskutiert, und ist hier zulässig.)

3.5 Visitor Pattern

Das Visitor Pattern wird verwendet, um verschiedene Operationen auf den Dateisystemelementen durchzuführen, ohne die Klassenhierarchie der Elemen-

te zu verändern. Die Basisschnittstelle `IVisitor` definiert die Besuchsmethoden für jede Art von Dateisystemelement. Konkrete Besucherklassen wie `DumpVisitor` und `FilterFileVisitor` implementieren diese Methoden, um spezifische Funktionalitäten bereitzustellen.

3.6 Template Methode Pattern

Das Template Methode Pattern wird in den Filter Visitor Klassen verwendet, um die gemeinsame Struktur der Filteroperationen zu definieren.

Die abstrakte Klasse `FilterVisitor` stellt die Template Methode bereit, die den allgemeinen Ablauf der Filterung definiert. Die konkreten Filterklassen wie `FilterFileVisitor` und `FilterLinkVisitor` implementieren die spezifischen Filterkriterien, während die allgemeine Logik in der Basisklasse verbleibt. Somit ist die Erweiterung um weitere Filtertypen einfach möglich, ohne die bestehende Struktur zu verändern.

4 Dokumentation der Komponenten (Klassen)

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

5 Testprotokollierung

```
1
2 **** TESTCASE START ****
3
4 ****
5
6 DumpVisitor Test
7 [Test OK] Result: (Expected: |---[root/]
8 |   |---[sub_folder/]
9 |   |   |---[sub_sub_folder/]
10 |   |   |   |---[file1.txt]
11 == Result: |---[root/]
12 |   |---[sub_folder/]
13 |   |   |---[sub_sub_folder/]
14 |   |   |   |---[file1.txt]
15 )
16
17 Test Exception in TestCase
18 [Test OK] Result: (Expected: true == Result: true)
19
20 Test Exception Bad Ostream in DumpVisitor
21 [Test OK] Result: (Expected: ERROR: bad output stream ==
22     ↪ Result: ERROR: bad output stream)
23
24 **** TESTCASE START ****
25
26
27 ****
28 **** TESTCASE START ****
29 ****
30
31 Test Exception nullptr in Visit File
32 [Test OK] Result: (Expected: ERROR Nullptr == Result: ERROR
33     ↪ Nullptr)
34
35 Test Exception nullptr in Visit Folder
36 [Test OK] Result: (Expected: ERROR Nullptr == Result: ERROR
37     ↪ Nullptr)
38
39 Test Exception nullptr in Visit Link
40 [Test OK] Result: (Expected: ERROR Nullptr == Result: ERROR
41     ↪ Nullptr)
```

```
39
40
41 ****
42
43
44 ****
45         TESTCASE START
46 ****
47
48 Test Exception nullptr in Visit File
49 [Test OK] Result: (Expected: ERROR Nullptr == Result: ERROR
  ↪ Nullptr)
50
51 Test Exception nullptr in Visit Folder
52 [Test OK] Result: (Expected: ERROR Nullptr == Result: ERROR
  ↪ Nullptr)
53
54 Test Exception nullptr in Visit Link
55 [Test OK] Result: (Expected: ERROR Nullptr == Result: ERROR
  ↪ Nullptr)
56
57
58 ****
59
60
61 ****
62         TESTCASE START
63 ****
64
65 Test Exception nullptr in Visit File
66 [Test OK] Result: (Expected: ERROR Nullptr == Result: ERROR
  ↪ Nullptr)
67
68 Test Exception nullptr in Visit Folder
69 [Test OK] Result: (Expected: ERROR Nullptr == Result: ERROR
  ↪ Nullptr)
70
71 Test Exception nullptr in Visit Link
72 [Test OK] Result: (Expected: ERROR Nullptr == Result: ERROR
  ↪ Nullptr)
73
74
75 ****
76
```

```
77 ****
78 ***** TESTCASE START ****
79 ****
80 ****
81
82 FilterLinkVisitor Test filtered amount
83 [Test OK] Result: (Expected: 1 == Result: 1)
84
85 FilterLinkVisitor Test filtered obj
86 [Test OK] Result: (Expected: file1.txt == Result: file1.txt)
87
88 Filter Link Visitor Test Dump
89 [Test OK] Result: (Expected: \root\sub_folder\sub_sub_folder\
90   ↪ LinkToFile1 -> file1.txt
91   == Result: \root\sub_folder\sub_sub_folder\LinkToFile1 ->
92   ↪ file1.txt
93 )
94
95 Test for Exception in Testcase
96 [Test OK] Result: (Expected: true == Result: true)
97
98
99
100 ****
101
102
103 **** TESTCASE START ****
104 ****
105 ****
106
107 FilterFileVisitor Test filtered amount
108 [Test OK] Result: (Expected: 2 == Result: 2)
109
110 FilterFileVisitor Test for filtered file
111 [Test OK] Result: (Expected: file3.txt == Result: file3.txt)
112
113 FilterFileVisitor Test for filtered file
114 [Test OK] Result: (Expected: file1.txt == Result: file1.txt)
115
116 Filter File Visitor Test Dump
117 [Test OK] Result: (Expected: \root\file3.txt
```

```
118 | \root\sub_folder\sub_sub_folder\file1.txt
119 | == Result: \root\file3.txt
120 | \root\sub_folder\sub_sub_folder\file1.txt
121 |
122 |
123 | Test for Exception in Testcase
124 | [Test OK] Result: (Expected: true == Result: true)
125 |
126 | Test for Exception in Dump with bad Ostream
127 | [Test OK] Result: (Expected: ERROR: bad output stream ==
128 |   ↪ Result: ERROR: bad output stream)
129 |
130 | Test for Exception in Filter File Visiter CTOR
131 | [Test OK] Result: (Expected: Invalid size range: minimum size
132 |   ↪ must be less than maximum size == Result: Invalid size
133 |   ↪ range: minimum size must be less than maximum size)
134 |
135 |
136 *****TESTCASE START*****
137 *****TESTCASE START*****
138 |
139 |
140 | Test if file was constructed
141 | [Test OK] Result: (Expected: true == Result: true)
142 |
143 | Test if Link was constructed
144 | [Test OK] Result: (Expected: true == Result: true)
145 |
146 | Test if Folder was constructed
147 | [Test OK] Result: (Expected: true == Result: true)
148 |
149 | Test for Execption in Tesstcase
150 | [Test OK] Result: (Expected: true == Result: true)
151 |
152 | Test Exception nullptr CTOR Link
153 | [Test OK] Result: (Expected: ERROR Nullptr == Result: ERROR
154 |   ↪ Nullptr)
155 |
156 *****
```

```
158
159 **** TESTCASE START ****
160
161 ****
162
163 Test normal CTOR Link
164 [Test OK] Result: (Expected: MyFolder == Result: MyFolder)
165
166 Test normal CTOR Link
167 [Test OK] Result: (Expected: LinkToMyFolder == Result:
168   ↪ LinkToMyFolder)
169
170 Test normal CTOR Link - error buffer
171 [Test OK] Result: (Expected: true == Result: true)
172
173 Test Copy CTOR of Link
174 [Test OK] Result: (Expected: 0000026FB3EB1148 == Result:
175   ↪ 0000026FB3EB1148)
176
177 Test for shallow Copy of Link
178 [Test OK] Result: (Expected: 0000026FB3EB1148 == Result:
179   ↪ 0000026FB3EB1148)
180
181 Test normal COPY CTOR Link - error buffer
182 [Test OK] Result: (Expected: true == Result: true)
183
184 Test Assign Op of Link
185 [Test OK] Result: (Expected: 0000026FB3EB1148 == Result:
186   ↪ 0000026FB3EB1148)
187
188 Test Assign Op for Parent of Link
189 [Test OK] Result: (Expected: Modified == Result: Modified)
190
191 Test Assing Op Link - error buffer
192 [Test OK] Result: (Expected: true == Result: true)
193
194 Test Self Assing Op Link - error buffer
195 [Test OK] Result: (Expected: true == Result: true)
196 Test Exception nullptr CTOR Link
```

```
197 [Test OK] Result: (Expected: ERROR Nullptr == Result: ERROR
    ↪ Nullptr)
198 Test Exception empty string CTOR Link
199 [Test OK] Result: (Expected: ERROR String Empty == Result:
    ↪ ERROR String Empty)
200
201 Test GetReferencedFSObject
202 [Test OK] Result: (Expected: file1.txt == Result: file1.txt)
203
204 Empty error buffer
205 [Test OK] Result: (Expected: true == Result: true)
206
207 Test chained links
208 [Test OK] Result: (Expected: Link1 == Result: Link1)
209
210 Test chained links - error buffer
211 [Test OK] Result: (Expected: true == Result: true)
212
213
214 Test link before destruction
215 [Test OK] Result: (Expected: true == Result: true)
216
217 Test link after object destruction
218 [Test OK] Result: (Expected: true == Result: true)
219
220 Test weak_ptr expiration - error buffer
221 [Test OK] Result: (Expected: true == Result: true)
222
223 Test AsLink() returns valid pointer
224 [Test OK] Result: (Expected: true == Result: true)
225
226 Test AsLink() reference matches
227 [Test OK] Result: (Expected: file.txt == Result: file.txt)
228
229 Test AsLink() - error buffer
230 [Test OK] Result: (Expected: true == Result: true)
231
232 Test Link SetName
233 [Test OK] Result: (Expected: NewName == Result: NewName)
234
235 Test SetName - error buffer
236 [Test OK] Result: (Expected: true == Result: true)
237
238 Test Link SetName empty string
```

```
239 [Test OK] Result: (Expected: ERROR String Empty == Result:  
240     ↪ ERROR String Empty)  
241 Test Link Accept visitor - not empty  
242 [Test OK] Result: (Expected: false == Result: false)  
243  
244 Test Link Accept - error buffer  
245 [Test OK] Result: (Expected: true == Result: true)  
246  
247  
248 *****  
249  
250  
251 *****  
252         TESTCASE START  
253 *****  
254  
255 Test normal CTOR Folder  
256 [Test OK] Result: (Expected: MyFolder == Result: MyFolder)  
257  
258 Get Child from folder  
259 [Test OK] Result: (Expected: 0000026FB3EC69E0 == Result:  
260     ↪ 0000026FB3EC69E0)  
261  
262 Get next Child from folder  
263 [Test OK] Result: (Expected: 0000026FB3EC75E0 == Result:  
264     ↪ 0000026FB3EC75E0)  
265  
266 Get Child for invalid index  
267 [Test OK] Result: (Expected: 0000000000000000 == Result:  
268     ↪ 0000000000000000)  
269  
270 Test Folder - error buffer  
271 [Test OK] Result: (Expected: true == Result: true)  
272  
273 Test Copy Ctor Folder - Child 0  
274 [Test OK] Result: (Expected: file1.txt == Result: file1.txt)  
275  
276 Test Copy Ctor Folder - Sub Folder File  
277 [Test OK] Result: (Expected: sub_file.txt == Result: sub_file.  
278     ↪ txt)  
279  
280 Test Copy Ctor Folder test for Deep Copy  
281 [Test OK] Result: (Expected: true == Result: true)
```

```
278 Test Copy Ctor Folder test for Deep Copy in Sub Folder File
279 [Test OK] Result: (Expected: true == Result: true)
280
281 Test Parent of Copied Folder
282 [Test OK] Result: (Expected: 0000026FB3EC6F18 == Result:
283   ↪ 0000026FB3EC6F18)
284
285 Test Folder - error buffer
286 [Test OK] Result: (Expected: true == Result: true)
287
288 Test Assign Op Folder - Child 0
289 [Test OK] Result: (Expected: file1.txt == Result: file1.txt)
290
291 Test Assign Op Folder Parent - Child 0
292 [Test OK] Result: (Expected: MyFolder == Result: MyFolder)
293
294 Test Folder - error buffer
295 [Test OK] Result: (Expected: true == Result: true)
296
297 Test Self Assign Folder - Child 0
298 [Test OK] Result: (Expected: 0000026FB3EC6CE0 == Result:
299   ↪ 0000026FB3EC6CE0)
300
301 Test Folder - error buffer
302 [Test OK] Result: (Expected: true == Result: true)
303
304 Test Remove Child from Folder
305 [Test OK] Result: (Expected: 0000026FB3EC6FE0 == Result:
306   ↪ 0000026FB3EC6FE0)
307
308 Test Folder - error buffer
309 [Test OK] Result: (Expected: true == Result: true)
310
311 Test Folder - add nullptr
312 [Test OK] Result: (Expected: ERROR Nullptr == Result: ERROR
313   ↪ Nullptr)
314
315 Test nested folders - root has subl
```

```
316 [Test OK] Result: (Expected: 0000026FB3EBB060 == Result:  
317     ↪ 0000026FB3EBB060)  
318  
319 Test nested folders - sub1 has sub2  
320 [Test OK] Result: (Expected: 0000026FB3EBB130 == Result:  
321     ↪ 0000026FB3EBB130)  
322  
323 Test nested folders - error buffer  
324 [Test OK] Result: (Expected: true == Result: true)  
325  
326 Test parent pointer set on Add  
327 [Test OK] Result: (Expected: parent == Result: parent)  
328  
329 Test parent pointer - error buffer  
330 [Test OK] Result: (Expected: true == Result: true)  
331  
332 Test remove non-existent child  
333 [Test OK] Result: (Expected: 0000026FB3EC6CE0 == Result:  
334     ↪ 0000026FB3EC6CE0)  
335  
336 Test remove non-existent - error buffer  
337 [Test OK] Result: (Expected: true == Result: true)  
338  
339 Test mixed children - file  
340 [Test OK] Result: (Expected: 0000026FB3EC76A0 == Result:  
341     ↪ 0000026FB3EC76A0)  
342  
343 Test mixed children - folder  
344 [Test OK] Result: (Expected: 0000026FB3EBB068 == Result:  
345     ↪ 0000026FB3EBB068)  
346  
347 Test mixed children - link  
348 [Test OK] Result: (Expected: 0000026FB3EC7160 == Result:  
349     ↪ 0000026FB3EC7160)  
350  
351 Test mixed children - error buffer  
352 [Test OK] Result: (Expected: true == Result: true)  
353
```

```
354 Test Accept visits children
355 [Test OK] Result: (Expected: true == Result: true)
356
357 Test Accept visitor - error buffer
358 [Test OK] Result: (Expected: true == Result: true)
359
360 Test Folder SetName
361 [Test OK] Result: (Expected: renamed == Result: renamed)
362
363 Test Folder SetName - error buffer
364 [Test OK] Result: (Expected: true == Result: true)
365
366
367 ****
368
369
370 ****
371         TESTCASE START
372 ****
373
374 Test normal CTOR File
375 [Test OK] Result: (Expected: file1.txt == Result: file1.txt)
376
377 Test normal CTOR File - size
378 [Test OK] Result: (Expected: 0 == Result: 0)
379
380 Test normal - write file size
381 [Test OK] Result: (Expected: 4096 == Result: 4096)
382
383 Test normal - error buffer empty
384 [Test OK] Result: (Expected: true == Result: true)
385
386 Test Copy Ctor
387 [Test OK] Result: (Expected: file1.txt == Result: file1.txt)
388
389 Test Copy Ctor Parent of file
390 [Test OK] Result: (Expected: ParentFolder == Result:
391             ↛ ParentFolder)
392
393 Test normal - error buffer empty
394 [Test OK] Result: (Expected: true == Result: true)
395 Test CTOR Empty string - error buffer empty
```

```
396 [Test OK] Result: (Expected: ERROR String Empty == Result:  
397     ↪ ERROR String Empty)  
398  
399 Test multiple writes  
400 [Test OK] Result: (Expected: 6000 == Result: 6000)  
401  
402 Test multiple writes - error buffer  
403 [Test OK] Result: (Expected: true == Result: true)  
404  
405 Test write to exact capacity  
406 [Test OK] Result: (Expected: 5120 == Result: 5120)  
407  
408 Test exact capacity - error buffer  
409 [Test OK] Result: (Expected: true == Result: true)  
410  
411 Test write exceeds capacity  
412 [Test OK] Result: (Expected: Not enough space to write data ==  
413     ↪ Result: Not enough space to write data)  
414  
415 Test write zero bytes  
416 [Test OK] Result: (Expected: 0 == Result: 0)  
417  
418 Test write zero - error buffer  
419 [Test OK] Result: (Expected: true == Result: true)  
420  
421 Test multiple writes to capacity  
422 [Test OK] Result: (Expected: 3000 == Result: 3000)  
423  
424 Test approach capacity - error buffer  
425 [Test OK] Result: (Expected: true == Result: true)  
426  
427 Test write when full  
428 [Test OK] Result: (Expected: Not enough space to write data ==  
429     ↪ Result: Not enough space to write data)  
430  
431 Test default blocksize  
432 [Test OK] Result: (Expected: 10000 == Result: 10000)  
433  
434 Test default blocksize - error buffer  
435 [Test OK] Result: (Expected: true == Result: true)  
436
```

```
437 Test File Accept - error buffer
438 [Test OK] Result: (Expected: true == Result: true)
439
440 Test File SetName
441 [Test OK] Result: (Expected: new.txt == Result: new.txt)
442
443 Test File SetName - error buffer
444 [Test OK] Result: (Expected: true == Result: true)
445
446 Test File AsFolder returns nullptr
447 [Test OK] Result: (Expected: true == Result: true)
448
449 Test File AsFolder - error buffer
450 [Test OK] Result: (Expected: true == Result: true)
451
452
453 ****
454
455
456 ****
457             TESTCASE START
458 ****
459
460 Dump of Test Filesystem via Dump Visitor:
461
462 |---[root/]
463 |   |---[file1.txt]
464 |   |---[file2.txt]
465 |   |---[file3.txt]
466 |   |---[file4.txt]
467 |   |---[sub_folder/]
468 |   |   |---[file5.txt]
469 |   |   |---[file6.txt]
470 |   |   |---[sub_sub_folder/]
471 |   |   |   |---[file7.txt]
472 |   |   |   |---[LinkToRoot->]
473
474
475 Test normal op Filesystem - error buffer empty
476 [Test OK] Result: (Expected: true == Result: true)
477
478 Test ReturnRoot matches
479 [Test OK] Result: (Expected: |---[root/]
480     == Result: |---[root/]
```

```
481 )
482
483 Test normal op Filesystem - error buffer empty
484 [Test OK] Result: (Expected: true == Result: true)
485
486 Test Exception Set Null Root
487 [Test OK] Result: (Expected: ERROR Nullptr == Result: ERROR
488   ↪ Nullptr)
489
490 Test Exception Set Null Factory
491 [Test OK] Result: (Expected: ERROR Nullptr == Result: ERROR
492   ↪ Nullptr)
493
494 Test Exception no Factory in Create Test FileSystem
495 [Test OK] Result: (Expected: ERROR Nullptr == Result: ERROR
496   ↪ Nullptr)
497
498
499 ****
500
501 TEST OK!!
```

6 Quellcode

6.1 Object.hpp

```
1  /*****//  
2  * \file Object.h  
3  * \brief Root base class for all objects  
4  *  
5  * \author Simon  
6  * \date December 2025  
7  *****/  
8  #ifndef OBJECT_H  
9  #define OBJECT_H  
10  
11 #include <string>  
12  
13 class Object{  
14 protected:  
15     /** \brief Prevent direct instantiation */  
16     Object() = default;  
17 public:  
18     /** \brief Virtual destructor */  
19     virtual ~Object() {}  
20 };  
21  
22 #endif // OBJECT_H
```

6.2 FSObjectFactory.hpp

```

1  ****  

2  * \file FSObjectFactory.hpp  

3  * \brief Simple Factory class to create filesystem objects  

4  *  

5  * \author Simon  

6  * \date November 2025  

7  ****  

8  #ifndef FS_OBJECT_FACTORY_HPP  

9  #define FS_OBJECT_FACTORY_HPP  

10  

11 #include "Object.h"  

12 #include "FSObject.hpp"  

13 #include "Folder.hpp"  

14 #include "File.hpp"  

15 #include "Link.hpp"  

16 #include <memory>  

17  

18  

19 class FSObjectFactory : public Object  

20 {  

21 public:  

22     using Uptr = std::unique_ptr<FSObjectFactory>;  

23  

24     inline static const std::string ERROR_NULLPTR = "ERROR_Nullptr";  

25  

26     /** \brief Create a File FSObject  

27      * \param name Name of the file  

28      * \param res_blocks Reserved blocks  

29      * \param blocksize Block size (default 4096)  

30      * \return Shared pointer to created File FSObject  

31      */  

32     FSObject::Sptr CreateFile(std::string_view name, const size_t res_blocks, const size_t blocksize  

33         = 4096) const;  

34  

35     /** \brief Create a Folder FSObject  

36      * \param name Name of the folder  

37      * \return Shared pointer to created Folder FSObject  

38      */  

39     FSObject::Spref CreateFolder(std::string_view name = "") const;  

40  

41     /** \brief Create a Link FSObject  

42      * \param name Name of the link  

43      * \param linkedObj Shared pointer to linked FSObject  

44      * \return Shared pointer to created Link FSObject  

45      */  

46     FSObject::Spref CreateLink(std::string_view name, FSObject::Spref linkedObj) const;  

47  

48 private:  

49 };  

#endif

```

6.3 FSObjectFactory.cpp

```
1 //*****\n2 * \file  FSObjectFactory.cpp\n3 * \brief Simple Factory class to create filesystem objects\n4 *\n5 * \author Simon\n6 * \date   December 2025\n7 *****\n8\n9 #include "FSObjectFactory.hpp"\n10\n11\n12 FSObject::Sptr FSObjectFactory::CreateFile(std::string_view name, size_t res_blocks, size_t blocksize)\n13 {\n14     return std::make_shared<File>(name, res_blocks, blocksize);\n15 }\n16\n17 FSObject::Sptr FSObjectFactory::CreateFolder(std::string_view name) const\n18 {\n19     return std::make_shared<Folder>(name);\n20 }\n21\n22 FSObject::Sptr FSObjectFactory::CreateLink(std::string_view name, FSObject::Sptr linkedObj) const\n23 {\n24     return std::make_shared<Link>(move(linkedObj), name);\n25 }
```

6.4 Filesystem.hpp

```

1  /***** //file Filesystem.hpp
2  * \file Filesystem.hpp
3  * \brief Filesystem class representing the root of a filesystem
4  *
5  * \author Simon
6  * \date November 2025
7  *****/
8 #ifndef FILE_SYSTEM_HPP
9 #define FILE_SYSTEM_HPP
10
11 #include "FSObject.hpp"
12 #include "IVisitor.hpp"
13 #include "FSObjectFactory.hpp"
14
15 class FileSystem : public Object
16 {
17 public:
18
19     // Public Error Messages
20     inline static const std::string ERROR_NULLPTR = "ERROR_Nullptr";
21
22     FileSystem() = default;
23
24     /** \brief Construct a FileSystem with a root FSObject
25      * \param root Root FSObject shared pointer
26      */
27     FileSystem(FSObject::Sptr root);
28
29     /** \brief Walk the filesystem with a visitor
30      * \param visitor Visitor to apply
31      * \return Reference to visitor
32      */
33     void Work(IVisitor& visitor);
34
35     /** \brief Returns the root FSObject
36      * \return Shared pointer to root
37      */
38     FSObject::Sptr ReturnRoot();
39
40     /** \brief Set the filesystem root
41      * \param root Shared pointer to new root
42      */
43     void SetRoot(FSObject::Sptr root);
44
45     /** \brief Set the filesystem root
46      * \param root Shared pointer to new root
47      */
48     void SetFactory(FSObjectFactory::Uptr Factory);
49
50     /**
51      * \brief Creates a Test Filesystem using the Factory.
52      * \throw std::invalid_argument if Factory is nullptr.
53      */
54     void CreateTestFilesystem();
55
56     // delete Copy and Assign Operator to prevent untested Behaviour
57     void operator=(FileSystem visit) = delete;
58     FileSystem(FileSystem& visit) = delete;
59
60 private:
61
62     FSObject::Sptr m_Root;
63     FSObjectFactory::Uptr m_Factory;
64 };
#endif

```

6.5 Filesystem.cpp

```
1 //*****\n2 * \file Filesystem.cpp\n3 * \brief Filesystem class representing the root of a filesystem\n4 *\n5 * \author Simon\n6 * \date November 2025\n7 *****\n8\n9 #include "Filesystem.hpp"\n10 #include <stdexcept>\n11 #include <algorithm>\n12\n13 constexpr size_t BLOCKSIZE_SMALL = 2048;\n14 constexpr size_t BLOCKSIZE_MEDIUM = 8192;\n15 constexpr size_t BLOCKSIZE_LARGE = 32768;\n16 constexpr size_t BLOCKSIZE_CUSTOM = 12288;\n17\n18 FileSystem::FileSystem(FSObject::Sptr root)\n19 {\n20     if (root == nullptr) throw std::invalid_argument(ERROR_NULLPTR);\n21\n22     m_Root = move(root);\n23 }\n24 void FileSystem::Work(IVisitor& visitor)\n25 {\n26     if (m_Root == nullptr) throw std::invalid_argument(ERROR_NULLPTR);\n27\n28     m_Root->Accept(visitor);\n29 }\n30\n31 FSObject::Sptr FileSystem::ReturnRoot()\n32 {\n33     return move(m_Root);\n34 }\n35\n36 void FileSystem::SetRoot(FSObject::Sptr root)\n37 {\n38     if (root == nullptr) throw std::invalid_argument(ERROR_NULLPTR);\n39\n40     m_Root = move(root);\n41 }\n42\n43 void FileSystem::SetFactory(FSObjectFactory::Uptr Factory)\n44 {\n45     if (Factory == nullptr) throw std::invalid_argument(ERROR_NULLPTR);\n46\n47     m_Factory = move(Factory);\n48 }\n49\n50 void FileSystem::CreateTestFilesystem()\n51 {\n52     if (m_Factory == nullptr) throw std::invalid_argument(ERROR_NULLPTR);\n53\n54     FSObject::Sptr root_folder = m_Factory->CreateFolder("root");\n55     IFolder::Sptr root_folder_ptr = root_folder->AsFolder();\n56     FSObject::Sptr sub_folder = m_Factory->CreateFolder("sub");\n57     IFolder::Sptr sub_folder_ptr = sub_folder->AsFolder();\n58     FSObject::Sptr sub_sub_folder = m_Factory->CreateFolder("sub");\n59     IFolder::Sptr sub_sub_folder_ptr = sub_sub_folder->AsFolder();\n60\n61     sub_folder->SetName("sub_folder");\n62     sub_sub_folder->SetName("sub_sub_folder");\n63\n64     root_folder->SetName("root");\n65     root_folder_ptr->Add(m_Factory->CreateFile("file1.txt", BLOCKSIZE_SMALL));\n66     root_folder_ptr->Add(m_Factory->CreateFile("file2.txt", BLOCKSIZE_SMALL));\n67     root_folder_ptr->Add(m_Factory->CreateFile("file3.txt", BLOCKSIZE_SMALL));\n68     root_folder_ptr->Add(m_Factory->CreateFile("file4.txt", BLOCKSIZE_SMALL));\n69     root_folder_ptr->Add(sub_folder);\n70     sub_folder_ptr->Add(m_Factory->CreateFile("file5.txt", BLOCKSIZE_MEDIUM));\n71     sub_folder_ptr->Add(m_Factory->CreateFile("file6.txt", BLOCKSIZE_LARGE));\n72 }
```

```
73     sub_folder_ptr->Add(sub_sub_folder);
74     sub_sub_folder_ptr->Add(m_Factory->CreateFile("file7.txt", BLOCKSIZE_CUSTOM));
75     sub_sub_folder_ptr->Add(m_Factory->CreateLink("LinkToRoot", root_folder));
76
77     m_Root = move(root_folder);
78 }
```

6.6 FSOObject.hpp

```

1  /*****\file FSOObject.hpp
2  * \brief Base class for filesystem objects
3  *
4  * \author Simon
5  * \date November 2025
6  *****/
7  #ifndef FS_OBJECT_HPP
8  #define FS_OBJECT_HPP
9
10
11 #include "Object.h"
12 #include "IVisitor.hpp"
13 #include "IFolder.hpp"
14 #include "ILink.hpp"
15
16 #include <memory>
17 #include <vector>
18
19 class FSOObject : public Object
20 {
21 public:
22     // Public Error Messages
23     inline static const std::string ERROR_NULLPTR = "ERROR_Nullptr";
24     inline static const std::string ERROR_STRING_EMPTY = "ERROR_String_Empty";
25
26     // Smart pointer types
27     using Sptr = std::shared_ptr<FSOObject>;
28     using Uptr = std::unique_ptr<FSOObject>;
29     using Wptr = std::weak_ptr<FSOObject>;
30
31     /** \brief Accept a visitor (pure virtual)
32      * \param visit Visitor to accept
33      */
34     virtual void Accept(IVisitor& visit) = 0;
35
36     /** \brief Clones it self as a new
37      * \return Shared pointer to the cloned FSOObject
38      */
39     virtual FSOObj_Sptr Clone() const = 0;
40
41     /** \brief Try to "cast" this FSOObject to a folder
42      * \return Shared pointer to IFolder or nullptr
43      */
44     virtual IFolder::Sptr AsFolder();
45
46     /** \brief Try to "cast" this FSOObject to a folder
47      * \return Shared pointer to IFolder or nullptr
48      */
49     virtual std::shared_ptr<const IFolder> AsFolder() const;
50
51     /** \brief Try to cast this FSOObject to a link
52      * \return Shared pointer to ILink or nullptr
53      */
54     virtual std::shared_ptr<const ILink> AsLink() const;
55
56     /** \brief Get the name of the object
57      * \return Name as std::string_view
58      */
59     std::string_view GetName() const;
60
61     /** \brief Set the name of the object
62      * \param name New name
63      */
64     void SetName(std::string_view name);
65
66
67     /** \brief Get parent as weak pointer
68      * \return Weak pointer to parent
69      */
70     FSOObj_Wptr GetParent() const;
71
72     /** \brief Set parent of this FSOObject

```

```
73     * \param parent Shared pointer to parent FSObject
74     */
75     void SetParent(Sptr parent);
76
77 protected:
78     /** \brief Construct an FSObject with optional name
79     * \param name Name of the FSObject
80     */
81     FSObject(std::string_view name = "");
82
83
84 private:
85     std::string m_Name;
86     FSObj_Wptr m_Parent;
87 };
88
89 #endif
```

6.7 FSObject.cpp

```
1 //*****\n2 * \file FSObject.cpp\n3 * \brief Base class for filesystem objects\n4 *\n5 * \author Simon\n6 * \date November 2025\n7 *****\n8 #include "FSObject.hpp"\n9 #include <string>\n10 #include <stdexcept>\n11\n12 IFolder::Sptr FSObject::AsFolder()\n13 {\n14     return nullptr;\n15 }\n16\n17 std::shared_ptr<const IFolder> FSObject::AsFolder() const\n18 {\n19     return nullptr;\n20 }\n21\n22 std::shared_ptr<const ILink> FSObject::AsLink() const\n23 {\n24     return nullptr;\n25 }\n26\n27 std::string_view FSObject::GetName() const\n28 {\n29     return std::string_view(m_Name);\n30 }\n31\n32 void FSObject::SetName(std::string_view name)\n33 {\n34     if (name.empty()) throw std::invalid_argument(ERROR_STRING_EMPTY);\n35     m_Name = name;\n36 }\n37\n38 void FSObject::SetParent(Sptr parent)\n39 {\n40     if (parent == nullptr) throw std::invalid_argument(ERROR_NULLPTR);\n41     m_Parent = move(parent);\n42 }\n43\n44 FSObject::FSObject(std::string_view name)\n45 {\n46     if (name.empty()) throw std::invalid_argument(ERROR_STRING_EMPTY);\n47     m_Name = name;\n48 }\n49\n50 FSOBJ_Wptr FSObject::GetParent() const\n51 {\n52     return m_Parent;\n53 }
```

6.8 File.hpp

```
1 //*****
2 * \file File.hpp
3 * \brief File class representing a file in the filesystem
4 *
5 * \author Simon
6 * \date November 2025
7 ****/
8 #ifndef FILE_HPP
9 #define FILE_HPP
10
11 #include "FSObject.hpp"
12
13 class File : public FSObject, public std::enable_shared_from_this<File>
14 {
15 public:
16     // Public Error Messages
17     inline static const std::string ERROR_NULLPTR = "ERROR_Nullptr";
18     inline static const std::string ERR_OUT_OF_SPACE = "Not_enough_space_to_write_data";
19
20     // Smart pointer types
21     using Uptr = std::unique_ptr<File>;
22     using Sptr = std::shared_ptr<File>;
23     using Wptr = std::shared_ptr<File>;
24
25     /** \brief Construct a file
26      * \param name File name
27      * \param res_blocks Reserved blocks
28      * \param blocksize Block size (default 4096)
29      */
30     File(std::string_view name, const size_t res_blocks, const size_t blocksize = 4096)
31         : m_size(0), m_blocksize(blocksize), FSObject{ name },
32         m_res_blocks(res_blocks)
33     {}
34
35     /** \brief Accept a visitor
36      * \param visit Visitor to accept
37      */
38     virtual void Accept(IVisitor& visit) override;
39
40     /** \brief Write bytes to the file (increases size)
41      * \param bytes Number of bytes to write
42      * Call by Value is intentional because it is faster than by reference for built-in
43      * types
44      */
45     void Write(const size_t bytes);
46
47     /** \brief Get current size of the file
48      * \return Size in bytes
49      */
50     size_t GetSize() const;
51
52     /** \brief Clones it self as a new
53      * \return Shared pointer to the cloned FSObject
54      */
55     virtual FSOBJ_Sptr Clone() const override;
56
57 private:
58     size_t m_size;
59     const size_t m_blocksize;
60     const size_t m_res_blocks;
61 };
#endif
```

6.9 File.cpp

```
1 //*****  
2 * File.cpp  
3 * \brief File class representing a file in the filesystem  
4 *  
5 * \author Simon  
6 * \date November 2025  
7 *****  
8  
9 #include "File.hpp"  
10 #include <stdexcept>  
11 /** \brief Accept a visitor for this file */  
12 void File::Accept(IVisitor& visit)  
13 {  
14     visit.Visit(move(shared_from_this()));  
15 }  
16  
17 /** \brief Write bytes to the file, throws on out of space */  
18 void File::Write(const size_t bytes)  
19 {  
20     if ((bytes + m_size) > m_blocksize * m_res_blocks)  
21         throw std::runtime_error(ERR_OUT_OF_SPACE);  
22  
23     m_size += bytes;  
24 }  
25  
26 /** \brief Return current size */  
27 size_t File::GetSize() const  
28 {  
29     return m_size;  
30 }  
31  
32 FSOBJ_Sptr File::Clone() const  
33 {  
34     return std::make_shared<File>(File(*this));  
35 }
```

6.10 IFolder.hpp

```

1  /***** *****/  

2  * \file IFolder.hpp  

3  * \brief Interface for folder-like FSObjects  

4  *  

5  * \author Simon  

6  * \date November 2025  

7  *****/  

8  #ifndef IFOLDER_HPP  

9  #define IFOLDER_HPP  

10 #include <memory>  

11  

12 // fwd declaration  

13 class FSObject;  

14  

15 // Type aliases  

16 using FSObj_Sptr = std::shared_ptr<FSObject>;  

17 using FSObj_Wptr = std::weak_ptr<FSObject>;  

18  

19 class IFolder  

20 {  

21 public:  

22  

23     using Sptr = std::shared_ptr<IFolder>;  

24  

25     /** \brief Add a child FSObject to the folder  

26      * \param fsobj Shared pointer to the FSObject to add  

27      */  

28     virtual void Add(FSObj_Sptr fsobj) =0;  

29  

30     /** \brief Get a child by index  

31      * \param idx Index of the child  

32      * \return Shared pointer to the child or nullptr if out of range  

33      */  

34     virtual FSObj_Sptr GetChild(size_t idx) const =0;  

35  

36     /** \brief Remove a child FSObject from the folder  

37      * \param fsobj Shared pointer to the FSObject to remove  

38      */  

39     virtual void Remove(FSObj_Sptr fsobj) =0;  

40  

41     /** \brief Virtual destructor */  

42     virtual ~IFolder() = default;  

43  

44 private:  

45 };  

46  

47 #endiff

```

6.11 Folder.hpp

```

1  /*****\file Folder.hpp
2  * \brief Folder class representing a folder in the filesystem
3  *
4  * \author Simon
5  * \date November 2025
6  *****/
7  #ifndef FOLDER_HPP
8  #define FOLDER_HPP
9
10
11 #include "IFolder.hpp"
12 #include "IVisitor.hpp"
13 #include "FSObject.hpp"
14
15 #include <memory>
16 #include <vector>
17
18 class Folder : public IFolder, public FSObject, public std::enable_shared_from_this<Folder>
19 {
20 public:
21
22     // Smart pointer types
23     using Uptr = std::unique_ptr<Folder>;
24     using Sptr = std::shared_ptr<Folder>;
25     using Wptr = std::weak_ptr<Folder>;
26     using Cont = std::vector<FSObj_Sptr>;
27
28     /** \brief Construct a folder with a name
29      * \param name Name of the folder
30      */
31     Folder(std::string_view name) : FSObject(name) {}
32
33     /** \brief Add a child FSObject to this folder
34      * \param fsobj Shared pointer to the child
35      */
36     virtual void Add(FSObj_Sptr fsobj);
37
38     /** \brief Get child by index
39      * \param idx Index (by value is faster than by reference)
40      * \return Shared pointer to child or nullptr
41      */
42     virtual FSObj_Sptr GetChild(const size_t idx) const override;
43
44     /** \brief Remove a child from the folder
45      * \param fsobj Child to remove
46      */
47     virtual void Remove(FSObj_Sptr fsobj);
48
49     /** \brief Cast this FSObject to a folder interface
50      * \return Shared pointer to IFolder
51      */
52     virtual std::shared_ptr<const IFolder> AsFolder() const override;
53
54     /** \brief Cast this FSObject to a folder interface
55      * \return Shared pointer to IFolder
56      */
57     virtual IFolder::Sptr AsFolder() override;
58
59     /** \brief Accept a visitor and propagate to children
60      * \param visit Visitor to accept
61      */
62     virtual void Accept(IVisitor& visit) override;
63
64     /** \brief Clones it self as a new
65      * \return Shared pointer to the cloned FSObject
66      */
67     virtual FSObj_Sptr Clone() const override;
68
69     /** \brief Assignment operator for Folder
70      * This makes a deep copy of the folder and its children.
71      * \param fold Folder to copy from
72      */

```

```
73     void operator=(const Folder& fold);
74
75 protected:
76     /**
77      * \brief Copy Constructor of a Folder .
78      * This makes a deep copy of the folder and its children.
79      * This is protected to prevent direct usage, use Clone() instead
80      * \param fold
81      */
82     Folder(const Folder& fold);
83
84     // DTOR is defaulted because no special action is needed!
85
86 private:
87     Folder::Cont m_Children;
88 };
89
90 #endif
```

6.12 Folder.cpp

```

1  /*************************************************************************/
2  * \file Folder.cpp
3  * \brief Folder class representing a folder in the filesystem
4  *
5  * \author Simon
6  * \date November 2025
7  /*************************************************************************/
8  #include "Folder.hpp"
9  #include <stdexcept>
10 #include <algorithm>
11
12
13 Folder::Folder(const Folder& fold) : FSObject(fold)
14 {
15     m_Children.reserve(fold.m_Children.size());
16     for (const auto & child : fold.m_Children)
17     {
18         // clone each child; do not call Add() because it needs shared_from_this()
19         // and we are still in the constructor so shared_from_this() is not available yet.
20         m_Children.emplace_back(child->Clone());
21     }
22 }
23
24
25 /** \brief Add child to folder, sets parent pointer on child */
26 void Folder::Add(FSObj_Sptr fsobj)
27 {
28     if (fsobj == nullptr) throw std::invalid_argument(FSObject::ERROR_NULLPTR);
29
30     fsobj->SetParent(std::move(shared_from_this()));
31
32     m_Children.emplace_back(move(fsobj));
33 }
34
35 /** \brief Get child by index */
36 FSObj_Sptr Folder::GetChild(const size_t idx) const
37 {
38     if(idx < m_Children.size())
39     {
40         return m_Children.at(idx);
41     }
42
43     return nullptr;
44 }
45
46 /** \brief Remove a child from container */
47 void Folder::Remove(FSObj_Sptr fsobj)
48 {
49     m_Children.erase(
50         std::remove(m_Children.begin(), m_Children.end(), fsobj), m_Children.end()
51     );
52 }
53
54 /** \brief Return this as IFolder shared pointer */
55 std::shared_ptr<const IFolder> Folder::AsFolder() const
56 {
57     return shared_from_this();
58 }
59
60 IFolder::Sptr Folder::AsFolder()
61 {
62     return shared_from_this();
63 }
64
65 /** \brief Accept a visitor and forward to children */
66 void Folder::Accept(IVisitor& visit)
67 {
68     visit.Visit(move(shared_from_this()));
69
70     for(auto& child : m_Children)
71     {
72         child->Accept(visit);
73     }
74 }
```

```
73     }
74 }
75
76 FSOBJ_Sptr Folder::Clone() const
77 {
78     // Create a shared_ptr-owned copy so we can set parent pointers correctly
79     // Use explicit new here so protected copy ctor is accessible in this class context
80     auto newFolder = std::shared_ptr<Folder>(new Folder(*this));
81
82     // Set parent of each cloned child to the new folder
83     for (auto & child : newFolder->m_Children)
84     {
85         if (child)
86         {
87             child->SetParent(newFolder);
88         }
89     }
90
91     return newFolder;
92 }
93
94
95 void Folder::operator=(const Folder& fold)
96 {
97     // prevent self-assignment
98     if (this != &fold)
99     {
100         // call base class assignment
101         FSOBJ::operator=(fold);
102
103         // clear current children
104         m_Children.clear();
105
106         // deep copy of children
107         m_Children.reserve(fold.m_Children.size());
108
109         for (const auto& child : fold.m_Children)
110         {
111             Add(child->Clone());
112         }
113     }
114 }
```

6.13 ILink.hpp

6.14 Link.hpp

```

1  /***** ****
2  * \file Link.hpp
3  * \brief A link to another FSObject
4  *
5  * \author Simon
6  * \date November 2025
7  *****/
8 #ifndef LINK_HPP
9 #define LINK_HPP
10
11 #include "FSObject.hpp"
12 #include "IVisitor.hpp"
13
14 class Link : public FSObject, public ILink, public std::enable_shared_from_this<Link>
15 {
16 public:
17
18     // Public Error Messages
19     using Sptr = std::shared_ptr<Link>;
20     using Uptr = std::unique_ptr<Link>;
21     using Wptr = std::weak_ptr<Link>;
22
23     /** \brief Constructor taking a shared pointer to the linked FSObject
24      * \param linked_obj Shared pointer to the referenced FSObject
25      * \param name Optional name for the link
26      */
27     explicit Link(FSObj_Sptr linked_obj, std::string_view name = "");
28
29     /** \brief Cast this object to link interface
30      * \return Shared pointer to ILink
31      */
32     virtual std::shared_ptr<const ILink> AsLink() const override;
33
34     /** \brief Get the referenced FSObject
35      * \return Shared pointer to the referenced FSObject or nullptr if expired
36      */
37     virtual FSObj_Sptr GetReferencedFSObject() const override;
38
39     /** \brief Accept a visitor
40      * \param visit Visitor to accept
41      */
42     virtual void Accept(IVisitor& visit) override;
43
44     /** \brief Clones it self as a new
45      * \return Shared pointer to the cloned FSObject
46      */
47     virtual FSObj_Sptr Clone() const override;
48
49 private:
50     /** \brief Weak pointer to the linked FSObject
51      */
52     FSObj_Wptr m_Ref;
53 };
54
55 #endif

```

6.15 Link.cpp

6.16 IVisitor.hpp

```
1  /*****  
2  * \file IVisitor.hpp  
3  * \brief Interface for visitor pattern in filesystem objects  
4  *  
5  * \author Simon  
6  * \date November 2025  
7  *****/  
8  #ifndef IVISITOR_HPP  
9  #define IVISITOR_HPP  
10  
11 // Forward declarations to avoid circular dependencies  
12 class Folder;  
13 class File;  
14 class Link;  
15  
16 #include <memory>  
17  
18 class IVisitor  
19 {  
20 public:  
21  
22     /** \brief Visit a folder  
23      * \param folder Shared pointer to the folder to visit  
24      */  
25     virtual void Visit(const std::shared_ptr<const Folder> folder)=0;  
26  
27     /** \brief Visit a file  
28      * \param file Shared pointer to the file to visit  
29      */  
30     virtual void Visit(const std::shared_ptr<const File> file)=0;  
31  
32     /** \brief Visit a link  
33      * \param link Shared pointer to the link to visit  
34      */  
35     virtual void Visit(const std::shared_ptr<const Link> link)=0;  
36  
37     /** \brief Virtual destructor for visitor implementations */  
38     virtual ~IVisitor() = default;  
39  
40 private:  
41 };  
42  
43 #endif
```

6.17 FilterVisitor.hpp

```

1  /*****\file FilterVisitor.hpp
2  * \brief Visitor that filters filesystem objects based on criteria defines in derived classes
3  *
4  * \author Simon
5  * \date November 2025
6  *****/
7  #ifndef FILTER_VISITOR_HPP
8  #define FILTER_VISITOR_HPP
9
10
11 #include "IVisitor.hpp"
12 #include "FSObject.hpp"
13
14 #include <vector>
15 #include <ostream>
16
17 class FilterVisitor : public Object, public IVisitor
18 {
19 public:
20
21     // Public Error Messages
22     inline static const std::string ERROR_NULLPTR = "ERROR_Nullptr";
23     inline static const std::string ERROR_BAD_OSTREAM = "ERROR:_bad_output_stream";
24
25     // container Alias for filtered objects (weak pointers)
26     using TContFSobj = std::vector<std::weak_ptr<const FSObject>>;
27
28     /** \brief Visit a folder (default no-op)
29      * \param folder Folder to visit
30      */
31     virtual void Visit(const std::shared_ptr<const Folder> folder) override;
32
33     /** \brief Visit a file and apply filter
34      * \param file File to visit
35      */
36     virtual void Visit(const std::shared_ptr<const File> file) override;
37
38     /** \brief Visit a link and apply filter
39      * \param link Link to visit
40      */
41     virtual void Visit(const std::shared_ptr<const Link> link) override;
42
43     /** \brief Dump filtered objects to stream
44      * \param ost Output stream
45      */
46     void DumpFiltered(std::ostream& ost) const;
47
48     /** \brief Get the container of filtered objects (weak pointers)
49      * \return Const reference to container
50      */
51     const TContFSobj & GetFilteredObjects() const;
52
53     // delete Copy and Assign Operator to prevent untested Behaviour
54     void operator=(FilterVisitor visit) = delete;
55     FilterVisitor(FilterVisitor& visit) = delete;
56
57 protected:
58
59     /** \brief Check if a file matches the filter
60      * \param file File to check
61      * \return true if accepted
62      */
63     virtual bool DoFilter(const std::shared_ptr<const File>& file) const = 0;
64
65     /** \brief Check if a link matches the filter
66      * \param link Link to check
67      * \return true if accepted
68      */
69     virtual bool DoFilter(const std::shared_ptr<const Link>& link) const = 0;
70
71     FilterVisitor() = default;
72

```

```
73 |     private:
74 | 
75 |     /** \brief Dump a single FSOBJECT path to the output stream
76 |      * \param fsobj Weak pointer to object
77 |      * \param ost Output stream
78 |      */
79 |     void DumpPath(const std::weak_ptr<const FSOBJECT> & fsobj, std::ostream& ost) const;
80 | 
81 |     TContFSObj m_FilterCont;
82 | };
83 | 
84 | #endif
```

6.18 FilterVisitor.cpp

```

1  /**************************************************************************/ 
2  * \file FilterVisitor.cpp
3  * \brief Visitor that filters filesystem objects based on criteria defines in derived classes
4  *
5  * \author Simon
6  * \date November 2025
7  /**************************************************************************/
8 #include "FilterVisitor.hpp"
9 #include "Folder.hpp"
10 #include "File.hpp"
11 #include "Link.hpp"
12
13 #include <vector>
14 #include <iostream>
15 #include <cassert>
16 #include <stdexcept>
17
18 void FilterVisitor::DumpPath(const std::weak_ptr<const FSObject> & fsobj, std::ostream& ost) const
19 {
20     // end recursion on expired weak pointer
21     if (fsobj.expired()) return;
22
23     const auto obj = fsobj.lock();
24     if (!obj) return; // defensive: lock could fail
25
26     // first dump parent path
27     DumpPath(obj->GetParent(), ost);
28
29     if (!ost.good()) throw std::invalid_argument(FilterVisitor::ERROR_BAD_OSTREAM);
30
31     ost << "\\";
32
33     const std::shared_ptr<const ILink> link_ptr = obj->AsLink();
34
35     if (link_ptr) {
36         const FSObject::Sptr linked_obj = link_ptr->GetReferencedFSObject();
37         if (linked_obj) {
38             ost << "_->" << linked_obj->GetName();
39         }
40         else {
41             ost << "_->" << "linked Object Expired!";
42         }
43     }
44 }
45
46 /**
47 * \brief Default visit for folder (no-op)
48 void FilterVisitor::Visit(const std::shared_ptr<const Folder> folder)
49 {
50     if (folder == nullptr) throw std::invalid_argument(ERROR_NULLPTR);
51 }
52
53 /**
54 * \brief Visit a file and if it matches add to filtered container
55 void FilterVisitor::Visit(const std::shared_ptr<const File> file)
56 {
57     if (file == nullptr) throw std::invalid_argument(ERROR_NULLPTR);
58
59     // if file matches filter add to container
60     if (DoFilter(file))
61     {
62         m_FilterCont.emplace_back(file);
63     }
64 }
65 /**
66 * \brief Visit a link and if it matches add to filtered container
67 void FilterVisitor::Visit(const std::shared_ptr<const Link> link)
68 {
69     if (link == nullptr) throw std::invalid_argument(ERROR_NULLPTR);
70
71     // if link matches filter add to container
72     if (DoFilter(link))
73     {

```

```
73     m_FilterCont.emplace_back(link);
74 }
75 }
76 /**
77  * \brief Dump all filtered objects to given ostream */
78 void FilterVisitor::DumpFiltered(std::ostream& ost) const
79 {
80     if (!ost.good()) throw std::invalid_argument(FilterVisitor::ERROR_BAD_OSTREAM);
81
82     for (const auto & obj : m_FilterCont) {
83         DumpPath(obj, ost);
84         ost << '\n';
85     }
86 }
87
88 /**
89  * \brief Return the filtered objects container */
90 const FilterVisitor::TContFSobj& FilterVisitor::GetFilteredObjects() const
91 {
92     return m_FilterCont;
93 }
```

6.19 FilterFileVisitor.hpp

```

1  ****  

2  * \file FilterFileVisitor.hpp  

3  * \brief Visitor that filters files by size range  

4  *  

5  * \author Simon  

6  * \date November 2025  

7  ****
8 #ifndef FILTER_FILE_VISITOR_HPP
9 #define FILTER_FILE_VISITOR_HPP
10
11 #include "FilterVisitor.hpp"
12
13 class FilterFileVisitor : public FilterVisitor
14 {
15 public:
16     // Public Error Messages
17     inline static const std::string ERROR_INVALID_SIZE_RANGE = "Invalid_size_range:_minimum_size_"
18         "must_be_less_than_maximum_size";
19
20     /** \brief Construct file filter with size range [min,max]
21      * \param min Minimum size (inclusive) call by value for built-in type -> is faster than by
22      *             reference
23      * \param max Maximum size (inclusive) call by value for built-in type -> is faster than by
24      *             reference
25      */
26     FilterFileVisitor(const size_t min, const size_t max);
27
28     // delete Copy and Assign Operator to prevent untested Behaviour
29     void operator=(FilterFileVisitor visit) = delete;
30     FilterFileVisitor(FilterFileVisitor& visit) = delete;
31
32 protected:
33
34     /** \brief Do filter check for files
35      * \param file File to check
36      * \return true if file size is within range
37      */
38     virtual bool DoFilter(const std::shared_ptr<const File>& file) const override;
39
40     /** \brief Links are not accepted by this filter
41      * \param link Link to check
42      * \return false always
43      */
44     virtual bool DoFilter(const std::shared_ptr<const Link>& link) const override;
45
46 private:
47     // cannot be const because there are checks in the constructor
48     size_t m_MinSize;
49     size_t m_MaxSize;
50 };
51
52 #endif

```

6.20 FilterFileVisitor.cpp

```
1 //*****\n2 * \file FilterFileVisitor.cpp\n3 * \brief Visitor that filters files by size range\n4 *\n5 * \author Simon\n6 * \date November 2025\n7 *****\n8 #include "FilterFileVisitor.hpp"\n9 #include "Folder.hpp"\n10 #include "File.hpp"\n11 #include "Link.hpp"\n12\n13 /** \brief Construct filter with size bounds */\n14 FilterFileVisitor::FilterFileVisitor(const size_t min, const size_t max)\n15 {\n16     if (min >= max) throw std::invalid_argument(ERROR_INVALID_SIZE_RANGE);\n17\n18     m_MinSize = min;\n19     m_MaxSize = max;\n20 }\n21\n22 /** \brief Accept files whose size is within range */\n23 bool FilterFileVisitor::DoFilter(const std::shared_ptr<const File>& file) const\n24 {\n25     if (file == nullptr) throw std::invalid_argument(ERROR_NULLPTR);\n26\n27     return file->GetSize() >= m_MinSize && file->GetSize() <= m_MaxSize;\n28 }\n29\n30 /** \brief Links are not accepted by file filter */\n31 bool FilterFileVisitor::DoFilter(const std::shared_ptr<const Link>& link) const\n32 {\n33     if (link == nullptr) throw std::invalid_argument(ERROR_NULLPTR);\n34\n35     return false;\n36 }
```

6.21 FilterLinkVisitor.hpp

```
1  /*****  
2  * \file  FilterLinkVisitor.hpp  
3  * \brief  Visitor that filters links in the filesystem  
4  *  
5  * \author Simon  
6  * \date   December 2025  
7  *****/  
8 #ifndef FILTER_LINK_VISITOR_HPP  
9 #define FILTER_LINK_VISITOR_HPP  
10  
11 #include "FilterVisitor.hpp"  
12  
13 class FilterLinkVisitor : public FilterVisitor  
14 {  
15 public:  
16  
17     FilterLinkVisitor() = default;  
18  
19     // delete Copy and Assign Operator to prevent untested Behaviour  
20     void operator=(FilterLinkVisitor visit) = delete;  
21     FilterLinkVisitor(FilterLinkVisitor& visit) = delete;  
22  
23 protected:  
24  
25     /** \brief Links are accepted by this filter  
26      * \param file File to check  
27      * \return false always  
28      */  
29     virtual bool DoFilter(const std::shared_ptr<const File>& file) const override;  
30  
31     /** \brief Links are accepted by this filter  
32      * \param link Link to check  
33      * \return true if link is present  
34      */  
35     virtual bool DoFilter(const std::shared_ptr<const Link>& link) const override;  
36  
37 private:  
38 };  
39  
40 #endif
```

6.22 FilterLinkVisitor.cpp

```
1 //*****\n2 * \file FilterLinkVisitor.cpp\n3 * \brief Visitor that filters links in the filesystem\n4 *\n5 * \author Simon\n6 * \date December 2025\n7 *****\n8 #include "FilterLinkVisitor.hpp"\n9 #include <cassert>\n10 #include <stdexcept>\n11\n12 /** \brief Files are not accepted by link filter */\n13 bool FilterLinkVisitor::DoFilter(const std::shared_ptr<const File>& file) const\n14 {\n15     if(file == nullptr) throw std::invalid_argument(ERROR_NULLPTR);\n16     return false;\n17 }\n18\n19 /** \brief Links are accepted by link filter */\n20 bool FilterLinkVisitor::DoFilter(const std::shared_ptr<const Link>& link) const\n21 {\n22     if(link == nullptr) throw std::invalid_argument(ERROR_NULLPTR);\n23     return true;\n24 }
```

6.23 DumpVisitor.hpp

```

1  ****  

2  * \file DumpVisitor.hpp  

3  * \brief Visitor that dumps filesystem object paths to an output stream  

4  *  

5  * \author Simon  

6  * \date November 2025  

7  ****  

8  #ifndef DUMP_VISITOR_HPP  

9  #define DUMP_VISITOR_HPP  

10  

11 #include <iostream>  

12 #include "IVisitor.hpp"  

13 #include "FSObject.hpp"  

14  

15 class DumpVisitor : public Object, public IVisitor  

16 {  

17 public:  

18  

19     // Public Error Messages  

20     inline static const std::string ERROR_NULLPTR = "ERROR_Nullptr";  

21     inline static const std::string ERROR_BAD_OSTREAM = "ERROR:_bad_output_stream";  

22  

23     /** \brief Construct a dumper that writes to given ostream  

24      * \param ost Output stream reference  

25      */  

26     DumpVisitor(std::ostream& ost) : m_ost{ ost } {}  

27  

28     /** \brief Visit folder  

29      * \param folder Folder to visit  

30      */  

31     virtual void Visit(const std::shared_ptr<const Folder> folder) override;  

32  

33     /** \brief Visit file  

34      * \param file File to visit  

35      */  

36     virtual void Visit(const std::shared_ptr<const File> file) override;  

37  

38     /** \brief Visit link  

39      * \param Link Link to visit  

40      */  

41     virtual void Visit(const std::shared_ptr<const Link> Link) override;  

42  

43     // delete Copy and Assign Operator to prevent untested Behaviour  

44     void operator=(DumpVisitor visit) = delete;  

45     DumpVisitor(DumpVisitor& visit) = delete;  

46  

47 private:  

48     /** \brief Dump a single FSObject path to the output stream  

49      * \param fsobj Shared pointer to object  

50      */  

51     void Dump(const std::shared_ptr<const FSObject> fsobj);  

52  

53     // Output stream reference  

54     std::ostream & m_ost;  

55 };  

56  

57 #endif

```

6.24 DumpVisitor.cpp

```

1  /*************************************************************************/
2  * \file DumpVisitor.cpp
3  * \brief Visitor that dumps filesystem object paths to an output stream
4  *
5  * \author Simon
6  * \date November 2025
7  ****
8  #include "DumpVisitor.hpp"
9  #include "Folder.hpp"
10 #include "File.hpp"
11 #include "Link.hpp"
12
13 #include <vector>
14 #include <algorithm>
15 #include <cassert>
16
17
18 /** \brief Visit folder and dump its path */
19 void DumpVisitor::Visit(const std::shared_ptr<const Folder> folder)
20 {
21     if (m_ost.fail()) throw std::invalid_argument(ERROR_BAD_OSTREAM);
22     if (folder == nullptr) throw std::invalid_argument(ERROR_NULLPTR);
23
24     Dump(folder);
25 }
26
27
28 /** \brief Visit file and dump its path */
29 void DumpVisitor::Visit(const std::shared_ptr<const File> file)
30 {
31     if (m_ost.fail()) throw std::invalid_argument(ERROR_BAD_OSTREAM);
32     if (file == nullptr) throw std::invalid_argument(ERROR_NULLPTR);
33
34     Dump(file);
35 }
36
37 /** \brief Visit link and dump its path */
38 void DumpVisitor::Visit(const std::shared_ptr<const Link> Link)
39 {
40     if (m_ost.fail()) throw std::invalid_argument(ERROR_BAD_OSTREAM);
41     if (Link == nullptr) throw std::invalid_argument(ERROR_NULLPTR);
42
43     Dump(Link);
44 }
45
46 /** \brief Dump full path for a FSObject to the internal ostream */
47 void DumpVisitor::Dump(const std::shared_ptr<const FSObject> fsobj)
48 {
49     assert(m_ost.good());
50     assert(fsobj != nullptr);
51
52     // Get parent pointer
53     FSObject::Sptr parent = fsobj->GetParent().lock();
54
55     // Print an indentation token for each ancestor
56     while (parent != nullptr) {
57         m_ost << "|__";
58         parent = parent->GetParent().lock();
59     }
60
61     m_ost << "|---[" << fsobj->GetName();
62
63     if (fsobj->AsFolder()) {
64         m_ost << "]\\n";
65     }
66     else if (fsobj->AsLink()) {
67         m_ost << "->]\\n";
68     }
69     else {
70         m_ost << "]\\n";
71     }
72 }
```

6.25 main.cpp

```
1 //*****\n2 * \file main.cpp\n3 * \brief Testdriver for the filesystem\n4 *\n5 * \author Simon\n6 * \date December 2025\n7 *****\n8\n9 #include <iostream>\n10 #include <string>\n11 #include <memory>\n12 #include "FSObject.hpp"\n13 #include "IFolder.hpp"\n14 #include "ILink.hpp"\n15 #include "FSObjectFactory.hpp"\n16 #include "DumpVisitor.hpp"\n17 #include "FilterFileVisitor.hpp"\n18 #include "FilterLinkVisitor.hpp"\n19 #include "Filesystem.hpp"\n20 #include <cassert>\n21 #include <sstream>\n22 #include "Test.hpp"\n23 #include "fstream"\n24 #include "vld.h"\n25\n26 using namespace std;\n27\n28 #define WriteOutputFile ON\n29\n30 static bool TestDumpVisitor(ostream& ost);\n31 static bool TestFilterLinkVisitor(ostream& ost);\n32 static bool TestFilterFileVisitor(ostream& ost);\n33 static bool TestVisitor(ostream& ost, IVisitor & visit);\n34 static bool TestFactory(ostream& ost);\n35 static bool TestLink(ostream& ost);\n36 static bool TestFolder(ostream& ost);\n37 static bool TestFile(ostream& ost);\n38 static bool TestFileSystem(ostream& ost);\n39\n40 int main()\n41 {\n42\n43     ofstream output{ "Testoutput.txt" };\n44     if (!output.is_open()) {\n45         cerr << "Konnte Testoutput.txt nicht öffnen" << TestCaseFail;\n46         return 1;\n47     }\n48\n49     try {\n50         DumpVisitor visitor(std::cout);\n51\n52         FilterLinkVisitor filter_link_visitor;\n53\n54         FilterFileVisitor filter_file_visitor(4096, 16384);\n55\n56         FileSystem homework;\n57\n58         homework.SetFactory(std::make_unique<FSObjectFactory>());\n59         homework.CreateTestFilesystem();\n60\n61         homework.Work(visitor);\n62\n63         std::cout << "-----" << std::endl;\n64         homework.Work(filter_link_visitor);\n65\n66         filter_link_visitor.DumpFiltered(std::cout);\n67\n68         std::cout << "-----" << std::endl;\n69\n70         homework.Work(filter_file_visitor);\n71\n72         filter_file_visitor.DumpFiltered(std::cout);
```

```
73
74     bool TestOK = true;
75
76     DumpVisitor dumper{ cout };
77     FilterLinkVisitor filter_link;
78     FilterFileVisitor filter_file(0, 1024);
79
80     TestOK = TestOK && TestDumpVisitor(cout);
81     TestOK = TestOK && TestVisitor(cout, dumper);
82     TestOK = TestOK && TestVisitor(cout, filter_link);
83     TestOK = TestOK && TestVisitor(cout, filter_file);
84     TestOK = TestOK && TestFilterLinkVisitor(cout);
85     TestOK = TestOK && TestFilterFileVisitor(cout);
86     TestOK = TestOK && TestFactory(cout);
87     TestOK = TestOK && TestLink(cout);
88     TestOK = TestOK && TestFolder(cout);
89     TestOK = TestOK && TestFile(cout);
90     TestOK = TestOK && TestFileSystem(cout);
91
92     if (WriteOutputFile) {
93
94         TestOK = TestOK && TestDumpVisitor(output);
95         TestOK = TestOK && TestVisitor(output, dumper);
96         TestOK = TestOK && TestVisitor(output, filter_link);
97         TestOK = TestOK && TestVisitor(output, filter_file);
98         TestOK = TestOK && TestFilterLinkVisitor(output);
99         TestOK = TestOK && TestFilterFileVisitor(output);
100        TestOK = TestOK && TestFactory(output);
101        TestOK = TestOK && TestLink(output);
102        TestOK = TestOK && TestFolder(output);
103        TestOK = TestOK && TestFile(output);
104        TestOK = TestOK && TestFileSystem(output);
105
106        if (TestOK) {
107            output << TestCaseOK;
108        }
109        else {
110            output << TestCaseFail;
111        }
112
113        output.close();
114    }
115
116    if (TestOK) {
117        cout << TestCaseOK;
118    }
119    else {
120        cout << TestCaseFail;
121    }
122 }
123 catch (const string& err) {
124     cerr << err << TestCaseFail;
125 }
126 catch (bad_alloc const& error) {
127     cerr << error.what() << TestCaseFail;
128 }
129 catch (const exception& err) {
130     cerr << err.what() << TestCaseFail;
131 }
132 catch (...) {
133     cerr << "UnhandeltedException" << TestCaseFail;
134 }
135
136 if (output.is_open()) output.close();
137
138     return 0;
139 };
140
141 bool TestDumpVisitor(ostream & ost)
142 {
143     assert(ost.good());
144     ost << TestStart;
145
146     bool TestOK = true;
```

```
148     string error_msg;
149
150     try {
151         FSOObjectFactory factory;
152         FSOObject::Sptr root_folder = factory.CreateFolder("root");
153         FSOObject::Sptr sub_folder = factory.CreateFolder("sub_folder");
154         FSOObject::Sptr sub_sub_folder = factory.CreateFolder("sub_sub_folder");
155         sub_sub_folder->AsFolder()->Add(File::Sptr{make_shared<File>("file1.txt", 2048)}));
156         sub_folder->AsFolder()->Add(sub_sub_folder);
157         root_folder->AsFolder()->Add(sub_folder);
158
159         stringstream result;
160         stringstream expected;
161
162         DumpVisitor dumper(result);
163
164         root_folder->Accept(dumper);
165
166         expected << "|---[root/]\n"
167             << "|__|---[sub_folder/]\n"
168             << "|__|__|---[sub_sub_folder/]\n"
169             << "|__|__|__|[file1.txt]\n";
170
171         TestOK = TestOK && check_dump(ost, "DumpVisitor_Test", expected.str(), result.str());
172
173     }
174     catch (const string& err) {
175         error_msg = err;
176     }
177     catch (bad_alloc const& error) {
178         error_msg = error.what();
179     }
180     catch (const exception& err) {
181         error_msg = err.what();
182     }
183     catch (...) {
184         error_msg = "Unhandelt_Exception";
185     }
186
187     TestOK = TestOK && check_dump(ost, "Test_Exception_in_TestCase", true, error_msg.empty());
188     error_msg.clear();
189
190     try {
191
192         FSOObjectFactory factory;
193         FSOObject::Sptr root_folder = factory.CreateFolder("root");
194
195         stringstream result;
196
197         result.setstate(ios::badbit);
198
199         DumpVisitor dumper(result);
200
201         root_folder->Accept(dumper); // <= sould throw Exception bad Ostream
202
203     }
204     catch (const string& err) {
205         error_msg = err;
206     }
207     catch (bad_alloc const& error) {
208         error_msg = error.what();
209     }
210     catch (const exception& err) {
211         error_msg = err.what();
212     }
213     catch (...) {
214         error_msg = "Unhandelt_Exception";
215     }
216
217     TestOK = TestOK && check_dump(ost, "Test_Exception_Bad_Ostream_in_DumpVisitor", DumpVisitor:::
218         ERROR_BAD_OSTREAM, error_msg);
219     error_msg.clear();
220
221     ost << TestEnd;
```

```
222     return TestOK;
223 }
224
225 bool TestFilterLinkVisitor(ostream& ost)
226 {
227     assert(ost.good());
228
229     ost << TestStart;
230
231     bool TestOK = true;
232     string error_msg;
233
234
235     try {
236         FSObjectFactory factory;
237         FSObject::Sptr root_folder = factory.CreateFolder("root");
238         FSObject::Sptr sub_folder = factory.CreateFolder("sub_folder");
239         FSObject::Sptr sub_sub_folder = factory.CreateFolder("sub_sub_folder");
240         File::Sptr file = make_shared<File>("file1.txt", 2048);
241         Link::Sptr link = make_shared<Link>(file, "LinkToFile1");
242         sub_sub_folder->AsFolder()->Add(file );
243         sub_sub_folder->AsFolder()->Add(link);
244         sub_folder->AsFolder()->Add(sub_sub_folder);
245         root_folder->AsFolder()->Add(sub_folder);
246
247         FilterLinkVisitor link_filter;
248
249         root_folder->Accept(link_filter);
250
251         TestOK = TestOK && check_dump(ost, "FilterLinkVisitor_Test_filtered_amount",
252                                         static_cast<size_t>(1), link_filter.GetFilteredObjects().size());
253         TestOK = TestOK && check_dump(ost, "FilterLinkVisitor_Test_filtered_obj", link->
254                                         GetReferencedFSObject()->GetName(), link_filter.GetFilteredObjects().cbegin()->lock
255                                         ()->AsLink()->GetReferencedFSObject()->GetName());
256
257         stringstream result;
258         stringstream expected;
259
260         link_filter.DumpFiltered(result);
261
262         expected << "\\root\\sub_folder\\sub_sub_folder\\LinkToFile1->_file1.txt" << std::endl
263         ;
264
265         TestOK = TestOK && check_dump(ost, "Filter_Link_Visitor_Test_Dump", expected.str(),
266                                         result.str());
267
268     } catch (const string& err) {
269         error_msg = err;
270     } catch (bad_alloc const& error) {
271         error_msg = error.what();
272     } catch (const exception& err) {
273         error_msg = err.what();
274     } catch (...) {
275         error_msg = "UnhandeltedException";
276     }
277
278     TestOK = TestOK && check_dump(ost, "Test_for_Exception_in_TestCase", true, error_msg.empty());
279     error_msg.clear();
280
281     try {
282         FilterLinkVisitor link_filter();
283
284         stringstream result;
285         result.setstate(ios::badbit);
286
287         link_filter.DumpFiltered(result);
288     } catch (const string& err) {
289         error_msg = err;
290     }
291 }
```

```
292     catch (bad_alloc const& error) {
293         error_msg = error.what();
294     }
295     catch (const exception& err) {
296         error_msg = err.what();
297     }
298     catch (...) {
299         error_msg = "UnhandeltedException";
300     }
301
302     TestOK = TestOK && check_dump(ost, "Test_for_Exception_in_Dump_with_Bad_Ostream", error_msg,
303                                     FilterLinkVisitor::ERROR_BAD_OSTREAM);
304     error_msg.clear();
305
306     ost << TestEnd;
307
308     return TestOK;
309 }
310
311 bool TestFilterFileVisitor(ostream& ost)
312 {
313     assert(ost.good());
314
315     ost << TestStart;
316
317     bool TestOK = true;
318     string error_msg;
319
320     try {
321         FSObjectFactory factory;
322         FSObject::Sptr root_folder = factory.CreateFolder("root");
323         FSObject::Sptr sub_folder = factory.CreateFolder("sub_folder");
324         FSObject::Sptr sub_sub_folder = factory.CreateFolder("sub_sub_folder");
325         File::Sptr file = make_shared<File>("file1.txt", 10);
326         File::Sptr file1 = make_shared<File>("file2.txt", 10);
327         File::Sptr file2 = make_shared<File>("file3.txt", 10);
328         File::Sptr file3 = make_shared<File>("file4.txt", 10);
329         Link::Sptr link = make_shared<Link>(file, "LinkToFile1");
330
331         file->Write(8192);
332         file1->Write(4096);
333         file2->Write(6000);
334         file3->Write(1000);
335
336         sub_sub_folder->AsFolder()->Add(file);
337         root_folder->AsFolder()->Add(file2);
338         sub_sub_folder->AsFolder()->Add(link);
339         sub_folder->AsFolder()->Add(sub_sub_folder);
340         sub_folder->AsFolder()->Add(file3);
341         root_folder->AsFolder()->Add(sub_folder);
342         root_folder->AsFolder()->Add(file1);
343
344         FilterFileVisitor file_filter(5000, 9000);
345
346         root_folder->Accept(file_filter);
347
348         TestOK = TestOK && check_dump(ost, "FilterFileVisitor_Test_filtered_amount", static_cast<size_t>(2), file_filter.GetFilteredObjects().size());
349         TestOK = TestOK && check_dump(ost, "FilterFileVisitor_Test_for_filtered_file", file2->GetName()
350                                         , file_filter.GetFilteredObjects().cbegin()->lock()->GetName());
351         TestOK = TestOK && check_dump(ost, "FilterFileVisitor_Test_for_filtered_file", file->GetName(),
352                                         file_filter.GetFilteredObjects().crbegin()->lock()->GetName());
353
354         stringstream result;
355         stringstream expected;
356
357         file_filter.DumpFiltered(result);
358         expected << "\\\root\\\\file3.txt" << std::endl
359                         << "\\\root\\\\sub_folder\\\\sub_sub_folder\\\\file1.txt" << std::endl;
360
361         TestOK = TestOK && check_dump(ost, "Filter_File_Visitor_Test_Dump_", expected.str(), result.str()
362                                         ());
363 }
```

```
362     }
363     catch (const string& err) {
364         error_msg = err;
365     }
366     catch (bad_alloc const& error) {
367         error_msg = error.what();
368     }
369     catch (const exception& err) {
370         error_msg = err.what();
371     }
372     catch (...) {
373         error_msg = "Unhandelt_Exception";
374     }
375 }
376 TestOK = TestOK && check_dump(ost, "Test_for_Exception_in_Testcase", true, error_msg.empty());
377 error_msg.clear();
378
379 try {
380     FilterFileVisitor file_filter(1,2);
381
382     stringstream result;
383     result.setstate(ios::badbit);
384
385     file_filter.DumpFiltered(result);
386 }
387 catch (const string& err) {
388     error_msg = err;
389 }
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, "Test_for_Exception_in_Dump_with_bad_Ostream", error_msg,
401     FilterLinkVisitor::ERROR_BAD_OSTREAM);
402 error_msg.clear();
403
404 try {
405     FilterFileVisitor file_filter( 2,1 ); // <= should throw invalid size range
406 }
407 catch (const string& err) {
408     error_msg = err;
409 }
410 catch (bad_alloc const& error) {
411     error_msg = error.what();
412 }
413 catch (const exception& err) {
414     error_msg = err.what();
415 }
416 catch (...) {
417     error_msg = "Unhandelt_Exception";
418 }
419
420 TestOK = TestOK && check_dump(ost, "Test_for_Exception_in_Filter_File_VisiterCTOR", error_msg,
421     FilterFileVisitor::ERROR_INVALID_SIZE_RANGE);
422 error_msg.clear();
423
424
425 ost << TestEnd;
426
427 return TestOK;
428 }
429
430 bool TestVisitor(ostream& ost, IVisitor& visit)
431 {
432     assert(ost.good());
433
434 }
```

```
435     ost << TestStart;
436
437     bool TestOK = true;
438     string error_msg;
439
440
441     try {
442
443         stringstream result;
444
445         File::Sptr file = nullptr;
446
447         visit.Visit(file); // <= sould throw Exception Nullptr
448
449     }
450     catch (const string& err) {
451         error_msg = err;
452     }
453     catch (bad_alloc const& error) {
454         error_msg = error.what();
455     }
456     catch (const exception& err) {
457         error_msg = err.what();
458     }
459     catch (...) {
460         error_msg = "Unhandelt_Exception";
461     }
462
463     TestOK = TestOK && check_dump(ost, "Test_Exception_Nullptr_in_Visit_File", DumpVisitor:::
464                                     ERROR_NULLPTR, error_msg);
465     error_msg.clear();
466
467     try {
468
469         stringstream result;
470
471         Folder::Sptr folder = nullptr;
472
473         visit.Visit(folder); // <= sould throw Exception Nullptr
474
475     }
476     catch (const string& err) {
477         error_msg = err;
478     }
479     catch (bad_alloc const& error) {
480         error_msg = error.what();
481     }
482     catch (const exception& err) {
483         error_msg = err.what();
484     }
485     catch (...) {
486         error_msg = "Unhandelt_Exception";
487     }
488
489     TestOK = TestOK && check_dump(ost, "Test_Exception_Nullptr_in_Visit_Folder", DumpVisitor:::
490                                     ERROR_NULLPTR, error_msg);
491     error_msg.clear();
492
493     try {
494
495         stringstream result;
496
497         Link::Sptr lnk = nullptr;
498
499         visit.Visit(lnk); // <= sould throw Exception Nullptr
500
501     }
502     catch (const string& err) {
503         error_msg = err;
504     }
505     catch (bad_alloc const& error) {
506         error_msg = error.what();
507     }
508     catch (const exception& err) {
509         error_msg = err.what();
```

```
508     }
509     catch (...) {
510         error_msg = "Unhandelt_Exception";
511     }
512 
513     TestOK = TestOK && check_dump(ost, "Test_Exception_nullptr_in_Visit_Link", DumpVisitor::
514         ERROR_NULLPTR, error_msg);
515     error_msg.clear();
516 
517     ost << TestEnd;
518 
519     return TestOK;
520 }
521 bool TestFactory(ostream& ost)
522 {
523     assert(ost.good());
524 
525     ost << TestStart;
526 
527     bool TestOK = true;
528     string error_msg;
529 
530     try {
531         FSObjectFactory fact;
532         FSObj_Sptr file = fact.CreateFile("file1.txt",10);
533         FSObj_Sptr folder = fact.CreateFolder("root");
534         FSObj_Sptr lnk = fact.CreateLink("link_to_file",file);
535 
536         TestOK = TestOK && check_dump(ost, "Test_if_file_was_constructed", true, file != nullptr);
537         TestOK = TestOK && check_dump(ost, "Test_if_Link_was_constructed", true, lnk->AsLink() != nullptr);
538         TestOK = TestOK && check_dump(ost, "Test_if_Folder_was_constructed", true, folder->AsFolder() != nullptr);
539 
540     }
541     catch (const string& err) {
542         error_msg = err;
543     }
544     catch (bad_alloc const& error) {
545         error_msg = error.what();
546     }
547     catch (const exception& err) {
548         error_msg = err.what();
549     }
550     catch (...) {
551         error_msg = "Unhandelt_Exception";
552     }
553 
554     TestOK = TestOK && check_dump(ost, "Test_for_Execution_in_Tesstcase", true, error_msg.empty());
555     error_msg.clear();
556 
557     try {
558         FSObjectFactory fact;
559         File::Sptr file= nullptr;
560         FSObj_Sptr Lnk = fact.CreateLink("Link_to_File", file);
561 
562     }
563     catch (const string& err) {
564         error_msg = err;
565     }
566     catch (bad_alloc const& error) {
567         error_msg = error.what();
568     }
569     catch (const exception& err) {
570         error_msg = err.what();
571     }
572     catch (...) {
573         error_msg = "Unhandelt_Exception";
574     }
575 
576     TestOK = TestOK && check_dump(ost, "Test_Exception_nullptrCTOR_Link", Link::ERROR_NULLPTR,
577         error_msg);
```

```
579     error_msg.clear();
580 
581     ost << TestEnd;
582 
583     return TestOK;
584 }
585 
586 bool TestLink(ostream& ost)
587 {
588     assert(ost.good());
589 
590     ost << TestStart;
591 
592     bool TestOK = true;
593     string error_msg;
594 
595     // test normal operation
596     try
597     {
598         std::string_view folder_name = "MyFolder";
599         std::string_view link_name = "LinkToMyFolder";
600         Folder::SPtr folder = make_shared<Folder>(folder_name);
601         Link::SPtr link = make_shared<Link>(folder, link_name);
602 
603         TestOK = TestOK && check_dump(ost, "Test_normal_CTOR_Link", folder_name, link->
604             GetReferencedFSObject()->GetName());
605         TestOK = TestOK && check_dump(ost, "Test_normal_CTOR_Link", link_name, link->GetName());
606 
607     }
608     catch (const string& err)
609     {
610         error_msg = err;
611     }
612     catch (bad_alloc const& error)
613     {
614         error_msg = error.what();
615     }
616     catch (const exception& err)
617     {
618         error_msg = err.what();
619     }
620     catch (...)
621     {
622         error_msg = "Unhandelt_Exception";
623     }
624 
625     TestOK = TestOK && check_dump(ost, "Test_normal_CTOR_Link_error_buffer", true, error_msg.empty());
626     ;
627     error_msg.clear();
628 
629     // test Copy CTOR of Link
630     try
631     {
632         std::string_view folder_name = "MyFolder";
633         std::string_view link_name = "LinkToMyFolder";
634         Folder::SPtr folder = make_shared<Folder>(folder_name);
635         Link::SPtr link = make_shared<Link>(folder, link_name);
636         Folder::SPtr parent_folder = make_shared<Folder>("AnotherFolder");
637         parent_folder->Add(link); // set parent of link
638 
639         // Call Copy CTOR
640         Link::SPtr link_copy = make_shared<Link>(*link);
641 
642         TestOK = TestOK && check_dump(ost, "Test_Copy_CTOR_of_Link", link->GetReferencedFSObject(),
643             link_copy->GetReferencedFSObject());
644 
645         // modify copied link referenced FSObject name
646         link_copy->GetReferencedFSObject()->SetName("NewFolderName");
647 
648         TestOK = TestOK && check_dump(ost, "Test_for_shallow_Copy_of_Link", link->GetReferencedFSObject()
649             (), link_copy->GetReferencedFSObject());
650 
651         parent_folder->SetName("Modified");
652 
653         TestOK = TestOK && check_dump(ost, "Test_for_parent_of_Copied_Link", parent_folder->GetName(),
654             link_copy->GetParent().lock()->GetName());
655     }
656     catch (const string& err) {
```

```
649     error_msg = err;
650 }
651 catch (bad_alloc const& error) {
652     error_msg = error.what();
653 }
654 catch (const exception& err) {
655     error_msg = err.what();
656 }
657 catch (...) {
658     error_msg = "UnhandeltedException";
659 }
660
661 TestOK = TestOK && check_dump(ost, "Test_normal_COPYCTOR_Link_error_buffer", true, error_msg.empty());
662 error_msg.clear();
663
664 // test Assign Op of Link
665 try
666 {
667     std::string_view folder_name = "MyFolder";
668     std::string_view link_name = "LinkToMyFolder";
669     Folder::Sptr folder = make_shared<Folder>(folder_name);
670     Link::Sptr link = make_shared<Link>(folder, link_name);
671     Folder::Sptr another_folder = make_shared<Folder>("AnotherFolder");
672     another_folder->Add(link); // set parent of link
673
674     Link::Sptr link_ass = make_shared<Link>(folder, "Ass_Link");
675
676     *link_ass = *link;
677
678     TestOK = TestOK && check_dump(ost, "Test_Assign_Op_of_Link", link->GetReferencedFSObject(),
679                                   link_ass->GetReferencedFSObject());
680
681     another_folder->SetName("Modified");
682
683     TestOK = TestOK && check_dump(ost, "Test_Assign_Op_for_Parent_of_Link", another_folder->GetName(),
684                                   link_ass->GetParent().lock()->GetName());
685
686     catch (const string& err) {
687         error_msg = err;
688     }
689     catch (bad_alloc const& error) {
690         error_msg = error.what();
691     }
692     catch (const exception& err) {
693         error_msg = err.what();
694     }
695     catch (...) {
696         error_msg = "UnhandeltedException";
697     }
698
699     TestOK = TestOK && check_dump(ost, "Test_Assing_Op_Link_error_buffer", true, error_msg.empty());
700     error_msg.clear();
701
702 // test Self Assign of Link
703 try
704 {
705     std::string_view folder_name = "MyFolder";
706     std::string_view link_name = "LinkToMyFolder";
707     Folder::Sptr folder = make_shared<Folder>(folder_name);
708     Link::Sptr link = make_shared<Link>(folder, link_name);
709
710     *link = *link; // <= could throw
711     catch (const string& err) {
712         error_msg = err;
713     }
714     catch (bad_alloc const& error) {
715         error_msg = error.what();
716     }
717     catch (const exception& err) {
718         error_msg = err.what();
719     }
720     catch (...) {
```

```
721     error_msg = "Unhandelt_Exception";
722 }
723
724 TestOK = TestOK && check_dump(ost, "Test_Self_Assing_Op_Link_<error_buffer", true, error_msg.empty
725     ());
726 error_msg.clear();
727
728 // test link to nullptr
729 try
730 {
731     Link::Sptr link = make_shared<Link>(nullptr, "LinkToNothing");
732 }
733 catch (const string& err) {
734     error_msg = err;
735 }
736 catch (bad_alloc const& error) {
737     error_msg = error.what();
738 }
739 catch (const exception& err) {
740     error_msg = err.what();
741 }
742 catch (...) {
743     error_msg = "Unhandelt_Exception";
744 }
745
746 TestOK = TestOK && check_dump(ost, "Test_Exception nullptr_CTOR_Link", Link::ERROR_NULLPTR,
747     error_msg);
748 error_msg.clear();
749
750 // test Link with empty string
751 try
752 {
753     File::Sptr file = make_shared<File>("file1.txt", 2048);
754     Link::Sptr link = make_shared<Link>(file, "");
755 }
756 catch (const string& err) {
757     error_msg = err;
758 }
759 catch (bad_alloc const& error) {
760     error_msg = error.what();
761 }
762 catch (const exception& err) {
763     error_msg = err.what();
764 }
765 catch (...) {
766     error_msg = "Unhandelt_Exception";
767 }
768
769 TestOK = TestOK && check_dump(ost, "Test_Exception_empty_string_CTOR_Link", Link::
770     ERROR_STRING_EMPTY, error_msg);
771 error_msg.clear();
772
773 // test Link GetReferencedFSObject
774 try
775 {
776     File::Sptr file = make_shared<File>("file1.txt", 2048);
777     Link::Sptr link = make_shared<Link>(file, file->GetName());
778
779     FSOBJ_Sptr ref = link->GetReferencedFSObject(); // <= should be File not Folder
780
781     TestOK = TestOK && check_dump(ost, "Test_GetReferencedFSObject", file->GetName(), ref->GetName
782         ());
783 }
784 catch (const string& err) {
785     error_msg = err;
786 }
787 catch (bad_alloc const& error) {
788     error_msg = error.what();
789 }
790 catch (...) {
791     error_msg = "Unhandelt_Exception",
```

```
792     }
793 
794     TestOK = TestOK && check_dump(ost, "Empty_error_buffer", true, error_msg.empty());
795     error_msg.clear();
796 
797     // Link to a Link (chained links)
798     try
799     {
800         File::Sptr file = make_shared<File>("original.txt", 2048);
801         Link::Sptr link1 = make_shared<Link>(file, "Link1");
802         Link::Sptr link2 = make_shared<Link>(link1, "Link2");
803 
804         TestOK = TestOK && check_dump(ost, "Test_chained_links",
805             link1->GetName(), link2->GetReferencedFSObject()->GetName());
806     }
807     catch (const exception& err) {
808         error_msg = err.what();
809     }
810     TestOK = TestOK && check_dump(ost, "Test_chained_links_error_buffer", true, error_msg.empty());
811     error_msg.clear();
812 
813     //Link when referenced object is destroyed (weak_ptr expiration)
814     try
815     {
816         Link::Sptr link;
817         {
818             File::Sptr file = make_shared<File>("temp.txt", 2048);
819             link = make_shared<Link>(file, "LinkToTemp");
820             TestOK = TestOK && check_dump(ost, "Test_link_before_destruction",
821                 true, link->GetReferencedFSObject() != nullptr);
822         } // file goes out of scope here
823 
824         FSOBJ_Sptr expired_ref = link->GetReferencedFSObject();
825         TestOK = TestOK && check_dump(ost, "Test_link_after_object_destruction",
826             true, expired_ref == nullptr);
827     }
828     catch (const exception& err) {
829         error_msg = err.what();
830     }
831     TestOK = TestOK && check_dump(ost, "Test_weak_ptr_expiration_error_buffer", true, error_msg.empty()
832         ());
833     error_msg.clear();
834 
835     //AsLink() method returns valid pointer
836     try
837     {
838         File::Sptr file = make_shared<File>("file.txt", 2048);
839         Link::Sptr link = make_shared<Link>(file, "TestLink");
840 
841         std::shared_ptr<const ILink> ilink = link->AsLink();
842         TestOK = TestOK && check_dump(ost, "Test_AsLink_returns_valid_pointer",
843             true, ilink != nullptr);
844         TestOK = TestOK && check_dump(ost, "Test_AsLink_reference_matches",
845             file->GetName(), ilink->GetReferencedFSObject()->GetName());
846     }
847     catch (const exception& err) {
848         error_msg = err.what();
849     }
850     TestOK = TestOK && check_dump(ost, "Test_AsLink_error_buffer", true, error_msg.empty());
851     error_msg.clear();
852 
853     //Link SetName functionality
854     try
855     {
856         File::Sptr file = make_shared<File>("file.txt", 2048);
857         Link::Sptr link = make_shared<Link>(file, "OriginalName");
858 
859         link->SetName("NewName");
860         TestOK = TestOK && check_dump(ost, "Test_Link_SetName",
861             string_view("NewName"), link->GetName());
862     }
863     catch (const exception& err) {
864         error_msg = err.what();
865     }
866     TestOK = TestOK && check_dump(ost, "Test_SetName_error_buffer", true, error_msg.empty());
```

```
866     error_msg.clear();
867
868     //Link SetName with empty string (should throw)
869     try
870     {
871         File::SPtr file = make_shared<File>("file.txt", 2048);
872         Link::SPtr link = make_shared<Link>(file, "OriginalName");
873         link->SetName("");
874     }
875     catch (const exception& err) {
876         error_msg = err.what();
877     }
878     TestOK = TestOK && check_dump(ost, "Test_Link_SetName_empty_string",
879                                     FSObject::ERROR_STRING_EMPTY, error_msg);
880     error_msg.clear();
881
882     // Link Accept visitor
883     try
884     {
885         File::SPtr file = make_shared<File>("file.txt", 2048);
886         Link::SPtr link = make_shared<Link>(file, "TestLink");
887         stringstream result;
888         DumpVisitor visitor(result);
889
890         link->Accept(visitor);
891         TestOK = TestOK && check_dump(ost, "Test_Link_Accept_visitor-not_empty",
892                                         false, result.str().empty());
893     }
894     catch (const exception& err) {
895         error_msg = err.what();
896     }
897     TestOK = TestOK && check_dump(ost, "Test_Link_Accept_error_buffer", true,
898                                   error_msg.empty());
899     error_msg.clear();
900
901     ost << TestEnd;
902     return TestOK;
903 }
904 bool TestFolder(ostream& ost)
905 {
906     assert(ost.good());
907
908     ost << TestStart;
909
910     bool TestOK = true;
911     string error_msg;
912
913     // test folder as intended
914     try
915     {
916         string_view folder_name = "MyFolder";
917         Folder::SPtr folder = make_shared<Folder>(folder_name);
918         TestOK = TestOK && check_dump(ost, "Test_normalCTOR_Folder", folder_name, folder->GetName());
919
920         File::SPtr file1 = make_shared<File>("file1.txt", 2048);
921         File::SPtr file2 = make_shared<File>("file2.txt", 4096);
922
923         folder->Add(file1);
924         folder->Add(file2);
925
926         FSObject::SPtr err_file = folder->GetChild(2); // <= should be nullptr
927         FSObject::SPtr shared_null = nullptr;
928
929         TestOK = TestOK && check_dump(ost, "Get_Child_from_folder", static_pointer_cast<FSObject>(file1),
930                                       folder->GetChild(0));
931         TestOK = TestOK && check_dump(ost, "Get_next_Child_from_folder", static_pointer_cast<FSObject>(
932                                         file2), folder->GetChild(1));
933         TestOK = TestOK && check_dump(ost, "Get_Child_for_invalid_index", err_file, shared_null);
934     }
935     catch (const string& err) {
936         error_msg = err;
937     }
938     catch (bad_alloc const& error) {
939         error_msg = error.what();
940     }
941     catch (const exception& err) {
```

```
939     error_msg = err.what();
940 }
941 catch (...) {
942     error_msg = "UnhandeltedException";
943 }
944
945 TestOK = TestOK && check_dump(ost, "Test_Folder->error_buffer", error_msg.empty(), true);
946 error_msg.clear();
947
948 // test Copy Ctor of Folder
949 try
{
950     string_view folder_name = "MyFolder";
951     Folder::Sptr folder = make_shared<Folder>( folder_name );
952     File::Sptr file1 = make_shared<File>("file1.txt", 2048);
953     File::Sptr file2 = make_shared<File>("file2.txt", 4096);
954     Folder::Sptr sub_folder = make_shared<Folder>("SubFolder");
955     File::Sptr sub_file = make_shared<File>("sub_file.txt", 1024);
956
957     folder->Add(file1);
958     folder->Add(file2);
959     folder->Add(sub_folder);
960     sub_folder->Add(sub_file);
961
962     // Call Copy Ctor
963     FSObject::Sptr folder_copy = folder->Clone();
964
965     TestOK = TestOK && check_dump(ost, "Test_Copy_Ctor_Folder->Child_0", file1->GetName(),
966                                   folder_copy->AsFolder()->GetChild(0)->GetName());
967     TestOK = TestOK && check_dump(ost, "Test_Copy_Ctor_Folder->Sub_File", sub_file
968                                   ->GetName(), folder_copy->AsFolder()->GetChild(2)->AsFolder()->GetChild(0)->
969                                   GetName());
970
971     file1->SetName("modified_file1.txt");
972     sub_file->SetName("modified_sub.txt");
973
974     TestOK = TestOK && check_dump(ost, "Test_Copy_Ctor_Folder_test_for_Deep_Copy", true,
975                                   file1->GetName() != folder_copy->AsFolder()->GetChild(0)->GetName());
976     TestOK = TestOK && check_dump(ost, "Test_Copy_Ctor_Folder_test_for_Deep_Copy_in_Sub_
977                                   Folder_File", true, sub_file->GetName() != folder_copy->AsFolder()->GetChild(2)->
978                                   AsFolder()->GetChild(0)->GetName());
979
980     TestOK = TestOK && check_dump(ost, "Test_Parent_of_Copied_Folder", static_pointer_cast<
981                                   FSObject>(folder_copy), folder_copy->AsFolder()->GetChild(0)->GetParent().lock());
982
983 }
984 catch (const string& err) {
985     error_msg = err;
986 }
987 catch (bad_alloc const& error) {
988     error_msg = error.what();
989 }
990 catch (const exception& err) {
991     error_msg = err.what();
992 }
993 catch (...) {
994     error_msg = "UnhandeltedException";
995 }
996
997 TestOK = TestOK && check_dump(ost, "Test_Folder->error_buffer", error_msg.empty(), true);
998 error_msg.clear();
999
1000 // test Assign Operator of Folder
1001 try
{
1002     string_view folder_name = "MyFolder";
1003     Folder::Sptr folder = make_shared<Folder>( folder_name );
1004     File::Sptr file1 = make_shared<File>("file1.txt", 2048);
1005     File::Sptr file2 = make_shared<File>("file2.txt", 4096);
1006
1007     folder->Add(file1);
1008     folder->Add(file2);
1009
1010     Folder::Sptr folder_ass = make_shared<Folder>( "Ass_folder" );
1011     *folder_ass = *folder;
```

```
1007
1008     TestOK = TestOK && check_dump(ost, "Test_Assign_Op_Folder->Child_0", file1->GetName(),
1009                                     folder_ass->GetChild(0)->GetName());
1010
1011     folder->SetName("Modified_Name");
1012
1013     TestOK = TestOK && check_dump(ost, "Test_Assign_Op_Folder->Parent->Child_0", folder_ass->
1014                                     GetName(), folder_ass->GetChild(0)->GetParent().lock()->GetName());
1015
1016     }
1017
1018     catch (const string& err) {
1019         error_msg = err;
1020     }
1021     catch (bad_alloc const& error) {
1022         error_msg = error.what();
1023     }
1024     catch (const exception& err) {
1025         error_msg = err.what();
1026     }
1027     catch (...) {
1028         error_msg = "Unhandelt_Exception";
1029     }
1030
1031     TestOK = TestOK && check_dump(ost, "Test_Folder->error_buffer", error_msg.empty(), true);
1032     error_msg.clear();
1033
1034     // test Assign Operator of Folder Self Assign
1035     try
1036     {
1037         string_view folder_name = "MyFolder";
1038         Folder::Sptr folder = make_shared<Folder>(folder_name);
1039         File::Sptr file1 = make_shared<File>("file1.txt", 2048);
1040         File::Sptr file2 = make_shared<File>("file2.txt", 4096);
1041
1042         folder->Add(file1);
1043         folder->Add(file2);
1044         *folder = *folder;
1045
1046         TestOK = TestOK && check_dump(ost, "Test_SelfAssign_Folder->Child_0",
1047                                         static_pointer_cast<FSObject>(file1), folder->GetChild(0));
1048
1049     }
1050     catch (const string& err) {
1051         error_msg = err;
1052     }
1053     catch (bad_alloc const& error) {
1054         error_msg = error.what();
1055     }
1056     catch (const exception& err) {
1057         error_msg = err.what();
1058     }
1059     catch (...) {
1060         error_msg = "Unhandelt_Exception";
1061     }
1062
1063     TestOK = TestOK && check_dump(ost, "Test_Folder->error_buffer", error_msg.empty(), true);
1064     error_msg.clear();
1065
1066     // test remove child
1067     try
1068     {
1069         Folder::Sptr folder = make_shared<Folder>("MyFolder");
1070         File::Sptr file1 = make_shared<File>("file1.txt", 2048);
1071         File::Sptr file2 = make_shared<File>("file2.txt", 4096);
1072         folder->Add(file1);
1073         folder->Add(file2);
1074         folder->Remove(file1);
1075         TestOK = TestOK && check_dump(ost, "Test_Remove_Child_from_Folder", static_pointer_cast<
1076                                         FSObject>(file2), folder->GetChild(0));
1077
1078     }
1079     catch (const string& err) {
1080         error_msg = err;
1081     }
1082     catch (bad_alloc const& error) {
1083         error_msg = error.what();
```

```
1078     }
1079     catch (const exception& err) {
1080         error_msg = err.what();
1081     }
1082     catch (...) {
1083         error_msg = "UnhandeltedException";
1084     }
1085
1086     TestOK = TestOK && check_dump(ost, "Test_Folder->error_buffer", error_msg.empty(), true);
1087     error_msg.clear();
1088
1089     // test add nullptr
1090     try
1091     {
1092         Folder::Sptr folder = make_shared<Folder>("MyFolder");
1093         FSOObject::Sptr null_ptr = nullptr;
1094         folder->Add(null_ptr); // <= should throw Exception
1095     }
1096     catch (const string& err) {
1097         error_msg = err;
1098     }
1099     catch (bad_alloc const& error) {
1100         error_msg = error.what();
1101     }
1102     catch (const exception& err) {
1103         error_msg = err.what();
1104     }
1105     catch (...) {
1106         error_msg = "UnhandeltedException";
1107     }
1108
1109     TestOK = TestOK && check_dump(ost, "Test_Folder->add nullptr", Folder::ERROR_NULLPTR, error_msg);
1110     error_msg.clear();
1111
1112     // test Folder with empty string
1113     try
1114     {
1115         Folder::Sptr folder = make_shared<Folder>("");
1116     }
1117     catch (const string& err) {
1118         error_msg = err;
1119     }
1120     catch (bad_alloc const& error) {
1121         error_msg = error.what();
1122     }
1123     catch (const exception& err) {
1124         error_msg = err.what();
1125     }
1126     catch (...) {
1127         error_msg = "UnhandeltedException";
1128     }
1129
1130     TestOK = TestOK && check_dump(ost, "Test_Folder->CTOR_with_empty_string", FSOObject::
1131         ERROR_STRING_EMPTY, error_msg);
1132     error_msg.clear();
1133
1134     //Nested folder structure
1135     try
1136     {
1137         Folder::Sptr root = make_shared<Folder>("root");
1138         Folder::Sptr sub1 = make_shared<Folder>("sub1");
1139         Folder::Sptr sub2 = make_shared<Folder>("sub2");
1140
1141         root->Add(sub1);
1142         sub1->Add(sub2);
1143
1144         TestOK = TestOK && check_dump(ost, "Test_nested_folders->root_has_sub1",
1145             sub1, static_pointer_cast<Folder>(root->GetChild(0)));
1146         TestOK = TestOK && check_dump(ost, "Test_nested_folders->sub1_has_sub2",
1147             sub2, static_pointer_cast<Folder>(sub1->GetChild(0)));
1148     }
1149     catch (const exception& err) {
1150         error_msg = err.what();
1151     }
1152     TestOK = TestOK && check_dump(ost, "Test_nested_folders->error_buffer", true, error_msg.empty());
```

```
1152     error_msg.clear();
1153
1154     //Parent pointer is set correctly when adding child
1155     try
1156     {
1157         Folder::Sptr parent = make_shared<Folder>("parent");
1158         File::Sptr child = make_shared<File>("child.txt", 2048);
1159
1160         parent->Add(child);
1161         FSOBJ_WPTR parent_wptr = child->GetParent();
1162         FSOBJ_SPTR parent_sptra = parent_wptr.lock();
1163
1164         TestOK = TestOK && check_dump(ost, "Test_parent_pointer_set_on_Add",
1165             parent->GetName(), parent_sptra->GetName());
1166     }
1167     catch (const exception& err) {
1168         error_msg = err.what();
1169     }
1170     TestOK = TestOK && check_dump(ost, "Test_parent_pointer_error_buffer", true, error_msg.empty());
1171     error_msg.clear();
1172
1173     //Remove non-existent child (should not crash)
1174     try
1175     {
1176         Folder::Sptr folder = make_shared<Folder>("folder");
1177         File::Sptr file1 = make_shared<File>("file1.txt", 2048);
1178         File::Sptr file2 = make_shared<File>("file2.txt", 2048);
1179
1180         folder->Add(file1);
1181         folder->Remove(file2); // file2 was never added
1182
1183         TestOK = TestOK && check_dump(ost, "Test_remove_non-existent_child",
1184             static_pointer_cast<FSObject>(file1), folder->GetChild(0));
1185     }
1186     catch (const exception& err) {
1187         error_msg = err.what();
1188     }
1189     TestOK = TestOK && check_dump(ost, "Test_remove_non-existent_error_buffer", true, error_msg.empty());
1190     error_msg.clear();
1191
1192     //Multiple children of different types
1193     try
1194     {
1195         Folder::Sptr folder = make_shared<Folder>("mixed");
1196         File::Sptr file = make_shared<File>("file.txt", 2048);
1197         Folder::Sptr subfolder = make_shared<Folder>("subfolder");
1198         Link::Sptr link = make_shared<Link>(file, "link");
1199
1200         folder->Add(file);
1201         folder->Add(subfolder);
1202         folder->Add(link);
1203
1204         TestOK = TestOK && check_dump(ost, "Test_mixed_children_file",
1205             static_pointer_cast<FSObject>(file), folder->GetChild(0));
1206         TestOK = TestOK && check_dump(ost, "Test_mixed_children_folder",
1207             static_pointer_cast<FSObject>(subfolder), folder->GetChild(1));
1208         TestOK = TestOK && check_dump(ost, "Test_mixed_children_link",
1209             static_pointer_cast<FSObject>(link), folder->GetChild(2));
1210     }
1211     catch (const exception& err) {
1212         error_msg = err.what();
1213     }
1214     TestOK = TestOK && check_dump(ost, "Test_mixed_children_error_buffer", true, error_msg.empty());
1215     error_msg.clear();
1216
1217     //AsFolder() returns valid pointer
1218     try
1219     {
1220         Folder::Sptr folder = make_shared<Folder>("test");
1221         IFolder::Sptr ifolder = folder->AsFolder();
1222
1223         TestOK = TestOK && check_dump(ost, "Test_AsFolder_returns_valid_pointer",
1224             true, ifolder != nullptr);
1225     }
```

```
1226     catch (const exception& err) {
1227         error_msg = err.what();
1228     }
1229     TestOK = TestOK && check_dump(ost, "Test_AsFolder()_>_error_buffer", true, error_msg.empty());
1230     error_msg.clear();
1231
1232     //Accept visitor with children
1233     try
1234     {
1235         Folder::Sptr folder = make_shared<Folder>("root");
1236         File::Sptr file = make_shared<File>("file.txt", 2048);
1237         folder->Add(file);
1238
1239         stringstream result;
1240         DumpVisitor visitor(result);
1241         folder->Accept(visitor);
1242
1243         // Should visit both folder and file
1244         TestOK = TestOK && check_dump(ost, "Test_Accept_visits_children",
1245             true, result.str().find("root") != string::npos &&
1246             result.str().find("file.txt") != string::npos);
1247     }
1248     catch (const exception& err)
1249     {
1250         error_msg = err.what();
1251     }
1252     TestOK = TestOK && check_dump(ost, "Test_Accept_visitor_>_error_buffer", true, error_msg.empty());
1253     error_msg.clear();
1254
1255     //SetName on folder
1256     try
1257     {
1258         Folder::Sptr folder = make_shared<Folder>("original");
1259         folder->SetName("renamed");
1260
1261         TestOK = TestOK && check_dump(ost, "Test_Folder_SetName",
1262             string_view("renamed"), folder->GetName());
1263     }
1264     catch (const exception& err)
1265     {
1266         error_msg = err.what();
1267     }
1268     TestOK = TestOK && check_dump(ost, "Test_Folder_SetName_>_error_buffer", true, error_msg.empty());
1269     error_msg.clear();
1270
1271     ost << TestEnd;
1272     return TestOK;
1273 }
1274 bool TestFile(ostream& ost)
1275 {
1276     assert(ost.good());
1277
1278     ost << TestStart;
1279
1280     bool TestOK = true;
1281     string error_msg;
1282
1283     // File as intended
1284     try
1285     {
1286         string_view file_name = "file1.txt";
1287         size_t block_size = 2048;
1288         size_t res_blocks = 20;
1289         File::Sptr file = make_shared<File>(file_name, res_blocks, block_size);
1290
1291         TestOK = TestOK && check_dump(ost, "Test_normal_CTOR_File", file_name, file->GetName());
1292         TestOK = TestOK && check_dump(
1293             ost, "Test_normal_CTOR_File_>_size",
1294             static_cast<size_t>(0), file->GetSize());
1295
1296         // Write to file
1297         size_t write_size = 4096;
1298         file->Write(write_size);
1299         TestOK = TestOK && check_dump(ost, "Test_normal_>write_file_size", write_size, file->GetSize()
1300             );
1301     }
1302     catch (const string& err) {
```

```
1300     error_msg = err;
1301 }
1302 catch (bad_alloc const& error) {
1303     error_msg = error.what();
1304 }
1305 catch (const exception& err) {
1306     error_msg = err.what();
1307 }
1308 catch (...) {
1309     error_msg = "UnhandeltedException";
1310 }
1311
1312 TestOK = TestOK && check_dump(ost, "Test_normal_error_buffer_empty", error_msg.empty(), true);
1313 error_msg.clear();
1314
1315 // File Copy Ctor
1316 try
1317 {
1318     string_view file_name = "file1.txt";
1319     size_t block_size = 2048;
1320     size_t res_blocks = 20;
1321     File::Sptr file = make_shared<File>( file_name, res_blocks, block_size );
1322     Folder::Sptr parent_folder = make_shared<Folder>("ParentFolder");
1323     parent_folder->Add(file);
1324
1325     File file_copy = *file; // Copy ctor
1326
1327     // Write to file
1328     size_t write_size = 4096;
1329
1330     file->Write(write_size);
1331
1332     TestOK = TestOK && check_dump(ost, "Test_Copy_Ctor", file->GetName(), file_copy.GetName());
1333
1334     TestOK = TestOK && check_dump(ost, "Test_Copy_Ctor_Parent_of_file", file->GetParent().lock()->
1335                                     GetName(), file_copy.GetParent().lock()->GetName());
1336 }
1337 catch (const string& err) {
1338     error_msg = err;
1339 }
1340 catch (bad_alloc const& error) {
1341     error_msg = error.what();
1342 }
1343 catch (const exception& err) {
1344     error_msg = err.what();
1345 }
1346 catch (...) {
1347     error_msg = "UnhandeltedException";
1348 }
1349
1350 TestOK = TestOK && check_dump(ost, "Test_normal_error_buffer_empty", error_msg.empty(), true);
1351 error_msg.clear();
1352
1353     // Assign Operator is deletet because of const members!
1354
1355 // File with empty string
1356 try
1357 {
1358     File::Sptr file = make_shared<File>("", 20, 2048);
1359 }
1360 catch (const string& err) {
1361     error_msg = err;
1362 }
1363 catch (bad_alloc const& error) {
1364     error_msg = error.what();
1365 }
1366 catch (const exception& err) {
1367     error_msg = err.what();
1368 }
1369 catch (...) {
1370     error_msg = "UnhandeltedException";
1371 }
1372
1373 TestOK = TestOK && check_dump(ost, "TestCTOREmpty_string_error_buffer_empty", error_msg, File::
1374     ERROR_STRING_EMPTY);
```

```
1373     error_msg.clear();
1374
1375     // Write multiple times
1376     try
1377     {
1378         File::Sptr file = make_shared<File>("multi.txt", 10, 2048);
1379
1380         file->Write(1000);
1381         file->Write(2000);
1382         file->Write(3000);
1383
1384         TestOK = TestOK && check_dump(ost, "Test_multiple_writes",
1385             static_cast<size_t>(6000), file->GetSize());
1386     }
1387     catch (const exception& err)
1388     {
1389         error_msg = err.what();
1390     }
1391     TestOK = TestOK && check_dump(ost, "Test_multiple_writes_error_buffer", true, error_msg.empty());
1392     error_msg.clear();
1393
1394     // Write exactly to capacity
1395     try
1396     {
1397         size_t blocks = 5;
1398         size_t blocksize = 1024;
1399         File::Sptr file = make_shared<File>("exact.txt", blocks, blocksize);
1400
1401         file->Write(blocks * blocksize); // Write exactly to capacity
1402
1403         TestOK = TestOK && check_dump(ost, "Test_write_to_exact_capacity",
1404             blocks * blocksize, file->GetSize());
1405     }
1406     catch (const exception& err)
1407     {
1408         error_msg = err.what();
1409     }
1410     TestOK = TestOK && check_dump(ost, "Test_exact_capacity_error_buffer", true, error_msg.empty());
1411     error_msg.clear();
1412
1413     // Write exceeds capacity (should throw)
1414     try
1415     {
1416         File::Sptr file = make_shared<File>("overflow.txt", 2, 1024);
1417         file->Write(3000); // Exceeds 2 * 1024 = 2048
1418     }
1419     catch (const exception& err)
1420     {
1421         error_msg = err.what();
1422     }
1423     TestOK = TestOK && check_dump(ost, "Test_write_exceeds_capacity",
1424         File::ERR_OUT_OF_SPACE, error_msg);
1425     error_msg.clear();
1426
1427     // Write zero bytes
1428     try
1429     {
1430         File::Sptr file = make_shared<File>("zero.txt", 10, 2048);
1431         file->Write(0);
1432
1433         TestOK = TestOK && check_dump(ost, "Test_write_zero_bytes",
1434             static_cast<size_t>(0), file->GetSize());
1435     }
1436     catch (const exception& err)
1437     {
1438         error_msg = err.what();
1439     }
1440     TestOK = TestOK && check_dump(ost, "Test_write_zero_error_buffer", true, error_msg.empty());
1441     error_msg.clear();
1442
1443     // Multiple writes approaching capacity
1444     try
1445     {
1446         File::Sptr file = make_shared<File>("approach.txt", 3, 1000);
1447         file->Write(1000);
1448         file->Write(1000);
1449         file->Write(1000); // Total = 3000, capacity = 3000
1450
1451         TestOK = TestOK && check_dump(ost, "Test_multiple_writes_to_capacity",
```

```
1448         static_cast<size_t>(3000), file->GetSize());
1449     }
1450     catch (const exception& err) {
1451         error_msg = err.what();
1452     }
1453     TestOK = TestOK && check_dump(ost, "Test_approach_capacity_-_error_buffer", true, error_msg.empty())
1454     );
1455     error_msg.clear();
1456 
1457     // Write after reaching capacity (should throw)
1458     try
1459     {
1460         File::Sptr file = make_shared<File>("full.txt", 2, 1024);
1461         file->Write(2048); // Fill to capacity
1462         file->Write(1);    // Should throw
1463     }
1464     catch (const exception& err) {
1465         error_msg = err.what();
1466     }
1467     TestOK = TestOK && check_dump(ost, "Test_write_when_full", File::ERR_OUT_OF_SPACE, error_msg);
1468     error_msg.clear();
1469 
1470     // File with default blocksize (4096)
1471     try
1472     {
1473         File::Sptr file = make_shared<File>("default.txt", 5); // Default blocksize = 4096
1474         file->Write(10000);
1475 
1476         TestOK = TestOK && check_dump(ost, "Test_default_blocksize", static_cast<size_t>(10000), file->
1477             GetSize());
1478     }
1479     catch (const exception& err) {
1480         error_msg = err.what();
1481     }
1482     TestOK = TestOK && check_dump(ost, "Test_default_blocksize_-_error_buffer", true, error_msg.empty())
1483     );
1484     error_msg.clear();
1485 
1486     // Accept visitor
1487     try
1488     {
1489         File::Sptr file = make_shared<File>("visitor.txt", 10, 2048);
1490         stringstream result;
1491         DumpVisitor visitor(result);
1492 
1493         file->Accept(visitor);
1494 
1495         TestOK = TestOK && check_dump(ost, "Test_File_Accept_visitor", true, result.str().find("visitor
1496             .txt") != string::npos);
1497     }
1498     catch (const exception& err) {
1499         error_msg = err.what();
1500     }
1501     TestOK = TestOK && check_dump(ost, "Test_File_Accept_-_error_buffer", true, error_msg.empty());
1502     error_msg.clear();
1503 
1504     // SetName on file
1505     try
1506     {
1507         File::Sptr file = make_shared<File>("old.txt", 10, 2048);
1508         file->SetName("new.txt");
1509 
1510         TestOK = TestOK && check_dump(ost, "Test_File_SetName",
1511             string_view("new.txt"), file->GetName());
1512     }
1513     catch (const exception& err) {
1514         error_msg = err.what();
1515     }
1516     TestOK = TestOK && check_dump(ost, "Test_File_SetName_-_error_buffer", true, error_msg.empty());
1517     error_msg.clear();
1518 
1519     // File AsFolder should return nullptr
1520     try
1521     {
1522         File::Sptr file = make_shared<File>("notfolder.txt", 10, 2048);
```

```
1519     IFolder::Sptr folder_ptr = file->AsFolder();
1520
1521     TestOK = TestOK && check_dump(ost, "Test_File_AsFolder_returns_nullptr", true, folder_ptr ==
1522         nullptr);
1523
1524     catch (const exception& err) {
1525         error_msg = err.what();
1526     }
1527     TestOK = TestOK && check_dump(ost, "Test_File_AsFolder_<error>buffer", true, error_msg.empty());
1528     error_msg.clear();
1529
1530     ost << TestEnd;
1531     return TestOK;
1532 }
1533
1534 bool TestFileSystem(ostream& ost)
1535 {
1536     assert(ost.good());
1537
1538     ost << TestStart;
1539
1540     bool TestOK = true;
1541     string error_msg;
1542
1543     try
1544     {
1545         FileSystem fsys;
1546
1547         fsys.SetFactory(make_unique<FSObjectFactory>());
1548
1549         // build a Test filesystem using the set Factory
1550         fsys.CreateTestFilesystem();
1551
1552         DumpVisitor dumper(ost);
1553
1554         ost << "Dump_of_Test_Filesystem_via_Dump_Visitor:\n\n";
1555
1556         fsys.Work(dumper);
1557
1558         ost << "\n\n";
1559     catch (const string& err) {
1560         error_msg = err;
1561     }
1562     catch (bad_alloc const& error) {
1563         error_msg = error.what();
1564     }
1565     catch (const exception& err) {
1566         error_msg = err.what();
1567     }
1568     catch (...) {
1569         error_msg = "Unhandelt_Exception";
1570     }
1571
1572     TestOK = TestOK && check_dump(ost, "Test_normal_op_Filesystem_<error>empty", error_msg.
1573         empty(), true);
1574     error_msg.clear();
1575
1576     try
1577     {
1578         FileSystem fsys;
1579
1580         FSObjectFactory factory;
1581
1582         FSObject::Sptr root = factory.CreateFolder("root");
1583
1584         fsys.SetRoot(move(root));
1585
1586         stringstream result;
1587         stringstream expected;
1588
1589         DumpVisitor dumper(result);
1590
1591 }
```

```
1592         fsys.Work(dumper);
1593
1594     root = move(fsys.ReturnRoot());
1595
1596     DumpVisitor expected_dumper(expected);
1597
1598     root->Accept(expected_dumper);
1599
1600     TestOK = TestOK && check_dump(ost, "Test_ReturnRoot_matches",
1601                                     expected.str(), result.str());
1602
1603 }
1604 catch (const string& err) {
1605     error_msg = err;
1606 }
1607 catch (bad_alloc const& error) {
1608     error_msg = error.what();
1609 }
1610 catch (const exception& err) {
1611     error_msg = err.what();
1612 }
1613 catch (...) {
1614     error_msg = "Unhandelt_Exception";
1615 }
1616
1617 TestOK = TestOK && check_dump(ost, "Test_normal_op_Filesystem_error_buffer_empty",
1618                               empty(), true);
1619 error_msg.clear();
1620
1621 try
1622 {
1623     FileSystem fsys;
1624     FSObject::Sptr root = nullptr;
1625
1626     fsys.SetRoot(move(root)); // <= should throw
1627 }
1628 catch (const string& err) {
1629     error_msg = err;
1630 }
1631 catch (bad_alloc const& error) {
1632     error_msg = error.what();
1633 }
1634 catch (const exception& err) {
1635     error_msg = err.what();
1636 }
1637 catch (...) {
1638     error_msg = "Unhandelt_Exception";
1639 }
1640
1641 TestOK = TestOK && check_dump(ost, "Test_Exception_Set_Null_Root", FileSystem::ERROR_NULLPTR,
1642                               error_msg);
1643 error_msg.clear();
1644
1645 try
1646 {
1647     FileSystem fsys;
1648     FSObjectFactory::Uptr factory = nullptr;
1649
1650     fsys.SetFactory(move(factory)); // <= should throw
1651 }
1652 catch (const string& err) {
1653     error_msg = err;
1654 }
1655 catch (bad_alloc const& error) {
1656     error_msg = error.what();
1657 }
1658 catch (const exception& err) {
1659     error_msg = err.what();
1660 }
1661 catch (...) {
1662     error_msg = "Unhandelt_Exception";
1663 }
```

```
1664     TestOK = TestOK && check_dump(ost, "Test_Exception_Set_Null_Factory", FileSystem::ERROR_NULLPTR ,
1665             error_msg);
1666     error_msg.clear();
1667     try
1668     {
1669         FileSystem fsys;
1670         fsys.CreateTestFilesystem(); // <= should throw because no factory set
1671     }
1672     catch (const string& err) {
1673         error_msg = err;
1674     }
1675     catch (bad_alloc const& error) {
1676         error_msg = error.what();
1677     }
1678     catch (const exception& err) {
1679         error_msg = err.what();
1680     }
1681     catch (...) {
1682         error_msg = "Unhandelt_Exception";
1683     }
1684
1685     TestOK = TestOK && check_dump(ost, "Test_Exception_no_Factory_in_Create_Test_FileSystem",
1686             FileSystem::ERROR_NULLPTR ,error_msg);
1687     error_msg.clear();
1688     try
1689     {
1690         FileSystem fsys;
1691         DumpVisitor dumper(ost);
1692         fsys.Work(dumper); // <= should throw because root is null
1693     }
1694     catch (const string& err) {
1695         error_msg = err;
1696     }
1697     catch (bad_alloc const& error) {
1698         error_msg = error.what();
1699     }
1700     catch (const exception& err) {
1701         error_msg = err.what();
1702     }
1703     catch (...) {
1704         error_msg = "Unhandelt_Exception";
1705     }
1706
1707     TestOK = TestOK && check_dump(ost, "Test_Exception_Work_with_no_root_set", FileSystem::
1708             ERROR_NULLPTR ,error_msg);
1709     error_msg.clear();
1710
1711     ost << TestEnd;
1712
1713     return TestOK;
1714 }
1715 }
```

6.26 Test.hpp

```

1 /*****\file Test.hpp
2 * \brief File that provides a Test Function with a formated output
3 *
4 * \author Simon
5 * \date April 2025
6 *****/
7 #ifndef TEST_HPP
8 #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 inline const char* colorRed() { return "\x1B[31m"; }
24 inline const char* colorGreen() { return "\x1B[32m"; }
25 inline const char* colorWhite() { return "\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 << "*****\n";
50         ost << "TESTCASE_START\n";
51         ost << "*****\n";
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 << "*****\n";
61         ost << std::endl;
62     }
63     return ost;
64 }
65
66 inline std::ostream& TestCaseOK(std::ostream& ost) {
67
68 #if COLOR_OUTPUT
69     if (ost.good()) {
70         ost << colorGreen() << "TEST_OK!!" << colorWhite() << std::endl;
71     }
72 #else

```

```

73     if (ost.good()) {
74         ost << "TEST_OK!!" << std::endl;
75     }
76 #endif // COLOR_OUTPUT
77
78     return ost;
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()
109             << "Result:" << (Expected:<: " << std::boolalpha << expected << " !=" << " "
110             : " << result << ")" << std::noboolalpha << std::endl << std::endl;
111         }
112         else {
113             ostr << testcase << std::endl << colorRed() << "[Test_FAILED]" << colorWhite()
114             << "Result:" << (Expected:<: " << std::boolalpha << expected << " !=" << " "
115             : " << result << ")" << std::noboolalpha << std::endl << std::endl;
116         }
117     }
118     else {
119         if (expected == result) {
120             ostr << testcase << std::endl << "[Test_OK]" << "Result:" << (Expected:<: " << std::
121             boolalpha << expected << " !=" << "Result:" << result << ")" << std::
122             noboolalpha << std::endl << std::endl;
123         }
124         else {
125             ostr << testcase << std::endl << "[Test_FAILED]" << "Result:" << (Expected:<: " << std::
126             boolalpha << expected << " !=" << "Result:" << result << ")" << std::
127             noboolalpha << std::endl << std::endl;
128         }
129     }
130     return expected == result;
131 }
132
133 template <typename T1, typename T2>
134 std::ostream& operator<< (std::ostream& ost, const std::pair<T1,T2> & p) {
135     if (!ost.good()) throw std::runtime_error("Error_bad_Ostream!");
136     ost << "(" << p.first << "," << p.second << ")";
137     return ost;
138 }
139

```

```
140 template <typename T>
141 std::ostream& operator<< (std::ostream& ost, const std::vector<T> & cont) {
142     if (!ost.good()) throw std::runtime_error("Error_bad_Ostream!");
143     std::copy(cont.cbegin(), cont.cend(), std::ostream_iterator<T>(ost, " "));
144     return ost;
145 }
146
147 template <typename T>
148 std::ostream& operator<< (std::ostream& ost, const std::list<T> & cont) {
149     if (!ost.good()) throw std::runtime_error("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::deque<T> & cont) {
156     if (!ost.good()) throw std::runtime_error("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::forward_list<T> & cont) {
163     if (!ost.good()) throw std::runtime_error("Error_bad_Ostream!");
164     std::copy(cont.cbegin(), cont.cend(), std::ostream_iterator<T>(ost, " "));
165     return ost;
166 }
167
168
169 #endif // !TEST_HPP
```