

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!



HSD

FH-HAGENBERG

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	FObject.hpp	34
6.7	FObject.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

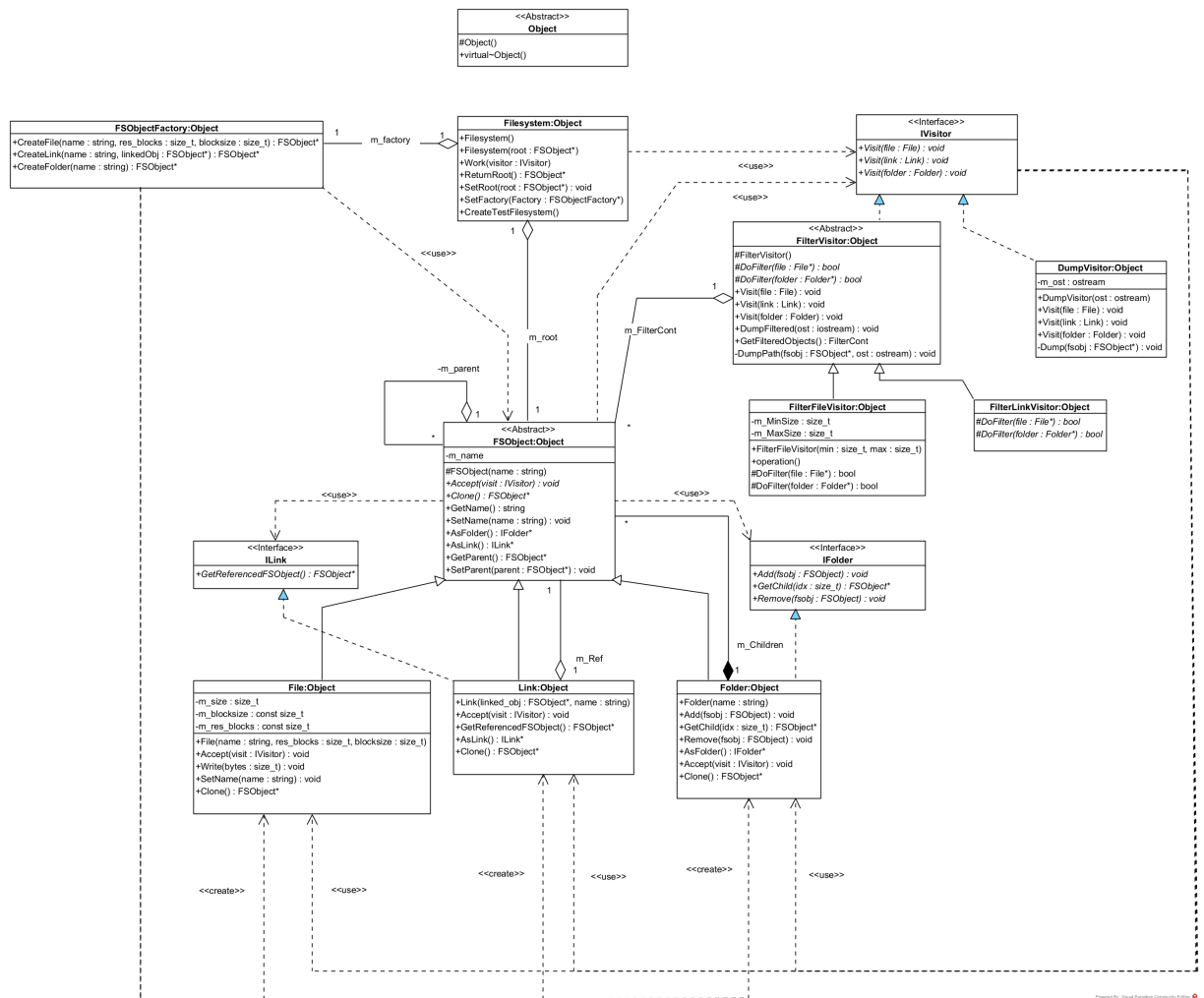
- Gibt die gesamte Dateisystemhierarchie aus.
- Ausgabe sowohl auf der Standardausgabe als auch in einer Datei möglich.
- Muss Dateien, Ordner und Verweise in strukturierter Form darstellen.

2.4.2 Visitor: FilterFiles

- Filtert Dateien anhand eines minimalen und maximalen Größenschwells.
- 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 Dateisystem-elemente.
- 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 capability 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 Ordnern 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, Ordnern und Verweisen. Dies ermöglicht eine zentrale Verwaltung der Erzeugungslogik und erleichtert zukünftige Erweiterungen. Beim konkreten Desing 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 *****
3             TESTCASE START
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 ==
    ↪ Result: ERROR: bad output stream)
22
23
24 *****
25
26
27 *****
28             TESTCASE START
29 *****
30
31 Test Exception nullptr in Visit File
32 [Test OK] Result: (Expected: ERROR Nullptr == Result: ERROR
    ↪ Nullptr)
33
34 Test Exception nullptr in Visit Folder
35 [Test OK] Result: (Expected: ERROR Nullptr == Result: ERROR
    ↪ Nullptr)
36
37 Test Exception nullptr in Visit Link
38 [Test OK] Result: (Expected: ERROR Nullptr == Result: ERROR
    ↪ 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 *****
79             TESTCASE START
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\
    ↳ LinkToFile1 -> file1.txt
90 == Result: \root\sub_folder\sub_sub_folder\LinkToFile1 ->
    ↳ file1.txt
91 )
92
93 Test for Exception in Testcase
94 [Test OK] Result: (Expected: true == Result: true)
95
96 Test for Exception in Dump with bad Ostream
97 [Test OK] Result: (Expected: ERROR: bad output stream ==
    ↳ Result: ERROR: bad output stream)
98
99
100 *****
101
102
103 *****
104             TESTCASE START
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 ==
    | ↪ Result: ERROR: bad output stream)
128 |
129 | Test for Exception in Filter File Visiter CTOR
130 | [Test OK] Result: (Expected: Invalid size range: minimum size
    | ↪ must be less than maximum size == Result: Invalid size
    | ↪ range: minimum size must be less than maximum size)
131 |
132 |
133 | *****
134 |
135 |
136 | *****
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 Exception in Testcase
150 | [Test OK] Result: (Expected: true == Result: true)
151 |
152 | Test Exception nullptr CTOR Link
153 | [Test OK] Result: (Expected: ERROR Nullptr == Result: ERROR
    | ↪ Nullptr)
154 |
155 |
156 | *****
157 |
```

```
158
159 *****
160 TESTCASE START
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:
    ↪ LinkToMyFolder)
168
169 Test normal CTOR Link - error buffer
170 [Test OK] Result: (Expected: true == Result: true)
171
172 Test Copy CTOR of Link
173 [Test OK] Result: (Expected: 0000026FB3EB1148 == Result:
    ↪ 0000026FB3EB1148)
174
175 Test for shallow Copy of Link
176 [Test OK] Result: (Expected: 0000026FB3EB1148 == Result:
    ↪ 0000026FB3EB1148)
177
178 Test for parent of Copied Link
179 [Test OK] Result: (Expected: Modified == Result: Modified)
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:
    ↪ 0000026FB3EB1148)
186
187 Test Assign Op for Parent of Link
188 [Test OK] Result: (Expected: Modified == Result: Modified)
189
190 Test Assing Op Link - error buffer
191 [Test OK] Result: (Expected: true == Result: true)
192
193 Test Self Assing Op Link - error buffer
194 [Test OK] Result: (Expected: true == Result: true)
195
196 Test Exception nullptr CTOR Link
```

```
197 [Test OK] Result: (Expected: ERROR Nullptr == Result: ERROR
    ↪ Nullptr)
198
199 Test Exception empty string CTOR Link
200 [Test OK] Result: (Expected: ERROR String Empty == Result:
    ↪ ERROR String Empty)
201
202 Test GetReferencedFSObject
203 [Test OK] Result: (Expected: file1.txt == Result: file1.txt)
204
205 Empty error buffer
206 [Test OK] Result: (Expected: true == Result: true)
207
208 Test chained links
209 [Test OK] Result: (Expected: Link1 == Result: Link1)
210
211 Test chained links - error buffer
212 [Test OK] Result: (Expected: true == Result: true)
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:
    ↪ ERROR String Empty)
240
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:
    ↪ 0000026FB3EC69E0)
260
261 Get next Child from folder
262 [Test OK] Result: (Expected: 0000026FB3EC75E0 == Result:
    ↪ 0000026FB3EC75E0)
263
264 Get Child for invalid index
265 [Test OK] Result: (Expected: 0000000000000000 == Result:
    ↪ 0000000000000000)
266
267 Test Folder - error buffer
268 [Test OK] Result: (Expected: true == Result: true)
269
270 Test Copy Ctor Folder - Child 0
271 [Test OK] Result: (Expected: file1.txt == Result: file1.txt)
272
273 Test Copy Ctor Folder - Sub Folder File
274 [Test OK] Result: (Expected: sub_file.txt == Result: sub_file.
    ↪ txt)
275
276 Test Copy Ctor Folder test for Deep Copy
277 [Test OK] Result: (Expected: true == Result: true)
```

```
278
279 Test Copy Ctor Folder test for Deep Copy in Sub Folder File
280 [Test OK] Result: (Expected: true == Result: true)
281
282 Test Parent of Copied Folder
283 [Test OK] Result: (Expected: 0000026FB3EC6F18 == Result:
    ↪ 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:
    ↪ 0000026FB3EC6CE0)
299
300 Test Folder - error buffer
301 [Test OK] Result: (Expected: true == Result: true)
302
303 Test Remove Child from Folder
304 [Test OK] Result: (Expected: 0000026FB3EC6FE0 == Result:
    ↪ 0000026FB3EC6FE0)
305
306 Test Folder - error buffer
307 [Test OK] Result: (Expected: true == Result: true)
308
309 Test Folder - add nullptr
310 [Test OK] Result: (Expected: ERROR Nullptr == Result: ERROR
    ↪ Nullptr)
311
312 Test Folder - CTOR with empty string
313 [Test OK] Result: (Expected: ERROR String Empty == Result:
    ↪ ERROR String Empty)
314
315 Test nested folders - root has sub1
```

```
316 [Test OK] Result: (Expected: 0000026FB3EBB060 == Result:
    ↪ 0000026FB3EBB060)
317
318 Test nested folders - sub1 has sub2
319 [Test OK] Result: (Expected: 0000026FB3EBB130 == Result:
    ↪ 0000026FB3EBB130)
320
321 Test nested folders - error buffer
322 [Test OK] Result: (Expected: true == Result: true)
323
324 Test parent pointer set on Add
325 [Test OK] Result: (Expected: parent == Result: parent)
326
327 Test parent pointer - error buffer
328 [Test OK] Result: (Expected: true == Result: true)
329
330 Test remove non-existent child
331 [Test OK] Result: (Expected: 0000026FB3EC6CE0 == Result:
    ↪ 0000026FB3EC6CE0)
332
333 Test remove non-existent - error buffer
334 [Test OK] Result: (Expected: true == Result: true)
335
336 Test mixed children - file
337 [Test OK] Result: (Expected: 0000026FB3EC76A0 == Result:
    ↪ 0000026FB3EC76A0)
338
339 Test mixed children - folder
340 [Test OK] Result: (Expected: 0000026FB3EBB068 == Result:
    ↪ 0000026FB3EBB068)
341
342 Test mixed children - link
343 [Test OK] Result: (Expected: 0000026FB3EC7160 == Result:
    ↪ 0000026FB3EC7160)
344
345 Test mixed children - error buffer
346 [Test OK] Result: (Expected: true == Result: true)
347
348 Test AsFolder() returns valid pointer
349 [Test OK] Result: (Expected: true == Result: true)
350
351 Test AsFolder() - 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:
    ↪ ParentFolder)
391
392 Test normal - error buffer empty
393 [Test OK] Result: (Expected: true == Result: true)
394
395 Test CTOR Empty string - error buffer empty
```



```
396 [Test OK] Result: (Expected: ERROR String Empty == Result:
    ↪ ERROR String Empty)
397
398 Test multiple writes
399 [Test OK] Result: (Expected: 6000 == Result: 6000)
400
401 Test multiple writes - error buffer
402 [Test OK] Result: (Expected: true == Result: true)
403
404 Test write to exact capacity
405 [Test OK] Result: (Expected: 5120 == Result: 5120)
406
407 Test exact capacity - error buffer
408 [Test OK] Result: (Expected: true == Result: true)
409
410 Test write exceeds capacity
411 [Test OK] Result: (Expected: Not enough space to write data ==
    ↪ Result: Not enough space to write data)
412
413 Test write zero bytes
414 [Test OK] Result: (Expected: 0 == Result: 0)
415
416 Test write zero - error buffer
417 [Test OK] Result: (Expected: true == Result: true)
418
419 Test multiple writes to capacity
420 [Test OK] Result: (Expected: 3000 == Result: 3000)
421
422 Test approach capacity - error buffer
423 [Test OK] Result: (Expected: true == Result: true)
424
425 Test write when full
426 [Test OK] Result: (Expected: Not enough space to write data ==
    ↪ Result: Not enough space to write data)
427
428 Test default blocksize
429 [Test OK] Result: (Expected: 10000 == Result: 10000)
430
431 Test default blocksize - error buffer
432 [Test OK] Result: (Expected: true == Result: true)
433
434 Test File Accept visitor
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
    ↪ Nullptr)
488
489 Test Exception Set Null Factory
490 [Test OK] Result: (Expected: ERROR Nullptr == Result: ERROR
    ↪ Nullptr)
491
492 Test Exception no Factory in Create Test FileSystem
493 [Test OK] Result: (Expected: ERROR Nullptr == Result: ERROR
    ↪ Nullptr)
494
495 Test Exception Work with no root set
496 [Test OK] Result: (Expected: ERROR Nullptr == Result: ERROR
    ↪ 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::Sptr 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::Sptr CreateLink(std::string_view name, FSObject::Sptr linkedObj) const;  
47 private:  
48 };  
49 #endif
```

6.3 FSObjectFactory.cpp

```
1  /*****  
2  * \file   FSObjectFactory.cpp  
3  * \brief  Simple Factory class to create filesystem objects  
4  *  
5  * \author Simon  
6  * \date   December 2025  
7  *****/  
8  
9  #include "FSObjectFactory.hpp"  
10  
11  
12  FSObject::Sptr FSObjectFactory::CreateFile(std::string_view name, size_t res_blocks, size_t blocksz)  
13      const  
14  {  
15      return std::make_shared<File>(name, res_blocks, blocksz);  
16  }  
17  FSObject::Sptr FSObjectFactory::CreateFolder(std::string_view name) const  
18  {  
19      return std::make_shared<Folder>(name);  
20  }  
21  
22  FSObject::Sptr FSObjectFactory::CreateLink(std::string_view name, FSObject::Sptr linkedObj) const  
23  {  
24      return std::make_shared<Link>(move(linkedObj), name);  
25  }
```

6.4 Filesystem.hpp

```
1  /*****
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 };
65 #endif
```

6.5 Filesystem.cpp

```
1  /*****  
2  * \file Filesystem.cpp  
3  * \brief Filesystem class representing the root of a filesystem  
4  *  
5  * \author Simon  
6  * \date November 2025  
7  *****/  
8  
9  #include "Filesystem.hpp"  
10 #include <stdexcept>  
11 #include <algorithm>  
12  
13 constexpr size_t BLOCKSIZE_SMALL = 2048;  
14 constexpr size_t BLOCKSIZE_MEDIUM = 8192;  
15 constexpr size_t BLOCKSIZE_LARGE = 32768;  
16 constexpr size_t BLOCKSIZE_CUSTOM = 12288;  
17  
18  
19 FileSystem::FileSystem(FSObject::Sptr root)  
20 {  
21     if (root == nullptr) throw std::invalid_argument(ERROR_NULLPTR);  
22  
23     m_Root = move(root);  
24 }  
25 void FileSystem::Work(IVisitor& visitor)  
26 {  
27     if (m_Root == nullptr) throw std::invalid_argument(ERROR_NULLPTR);  
28  
29     m_Root->Accept(visitor);  
30 }  
31  
32 FSObject::Sptr FileSystem::ReturnRoot()  
33 {  
34     return move(m_Root);  
35 }  
36  
37 void FileSystem::SetRoot(FSObject::Sptr root)  
38 {  
39     if (root == nullptr) throw std::invalid_argument(ERROR_NULLPTR);  
40  
41     m_Root = move(root);  
42 }  
43  
44 void FileSystem::SetFactory(FSObjectFactory::Uptr Factory)  
45 {  
46     if (Factory == nullptr) throw std::invalid_argument(ERROR_NULLPTR);  
47  
48     m_Factory = move(Factory);  
49 }  
50  
51 void FileSystem::CreateTestFilesystem()  
52 {  
53     if (m_Factory == nullptr) throw std::invalid_argument(ERROR_NULLPTR);  
54  
55     FSObject::Sptr root_folder = m_Factory->CreateFolder("root");  
56     IFolder::Sptr root_folder_ptr = root_folder->AsFolder();  
57     FSObject::Sptr sub_folder = m_Factory->CreateFolder("sub");  
58     IFolder::Sptr sub_folder_ptr = sub_folder->AsFolder();  
59     FSObject::Sptr sub_sub_folder = m_Factory->CreateFolder("sub");  
60     IFolder::Sptr sub_sub_folder_ptr = sub_sub_folder->AsFolder();  
61  
62     sub_folder->SetName("sub_folder");  
63     sub_sub_folder->SetName("sub_sub_folder");  
64  
65     root_folder->SetName("root");  
66     root_folder_ptr->Add(m_Factory->CreateFile("file1.txt", BLOCKSIZE_SMALL));  
67     root_folder_ptr->Add(m_Factory->CreateFile("file2.txt", BLOCKSIZE_SMALL));  
68     root_folder_ptr->Add(m_Factory->CreateFile("file3.txt", BLOCKSIZE_SMALL));  
69     root_folder_ptr->Add(m_Factory->CreateFile("file4.txt", BLOCKSIZE_SMALL));  
70     root_folder_ptr->Add(sub_folder);  
71     sub_folder_ptr->Add(m_Factory->CreateFile("file5.txt", BLOCKSIZE_MEDIUM));  
72     sub_sub_folder_ptr->Add(m_Factory->CreateFile("file6.txt", BLOCKSIZE_LARGE));
```



```
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 FSObject.hpp

```

1  /*****
2  * \file FSObject.hpp
3  * \brief Base class for filesystem objects
4  *
5  * \author Simon
6  * \date November 2025
7  *****/
8  #ifndef FS_OBJECT_HPP
9  #define FS_OBJECT_HPP
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 FSObject : 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<FSObject>;
28     using Uptr = std::unique_ptr<FSObject>;
29     using Wptr = std::weak_ptr<FSObject>;
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 FSObject
38     */
39     virtual FSObj_Sptr Clone() const = 0;
40
41     /** \brief Try to "cast" this FSObject to a folder
42     * \return Shared pointer to IFolder or nullptr
43     */
44     virtual IFolder::Sptr AsFolder();
45
46     /** \brief Try to "cast" this FSObject 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 FSObject 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     /** \brief Get parent as weak pointer
67     * \return Weak pointer to parent
68     */
69     FSObj_Wptr GetParent() const;
70
71     /** \brief Set parent of this FSObject

```

```
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  /*****
2  * \file FSObject.cpp
3  * \brief Base class for filesystem objects
4  *
5  * \author Simon
6  * \date   November 2025
7  *****/
8  #include "FSObject.hpp"
9  #include <string>
10 #include <stdexcept>
11
12 IFolder::Sptr FSObject::AsFolder()
13 {
14     return nullptr;
15 }
16
17 std::shared_ptr<const IFolder> FSObject::AsFolder() const
18 {
19     return nullptr;
20 }
21
22 std::shared_ptr<const ILink> FSObject::AsLink() const
23 {
24     return nullptr;
25 }
26
27 std::string_view FSObject::GetName() const
28 {
29     return std::string_view(m_Name);
30 }
31
32 void FSObject::SetName(std::string_view name)
33 {
34     if (name.empty()) throw std::invalid_argument(ERROR_STRING_EMPTY);
35     m_Name = name;
36 }
37
38 void FSObject::SetParent(Sptr parent)
39 {
40     if (parent == nullptr) throw std::invalid_argument(ERROR_NULLPTR);
41     m_Parent = move(parent);
42 }
43
44 FSObject::FSObject(std::string_view name)
45 {
46     if (name.empty()) throw std::invalid_argument(ERROR_STRING_EMPTY);
47     m_Name = name;
48 }
49
50 FSObj_Wptr FSObject::GetParent() const
51 {
52     return m_Parent;
53 }
```

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::weak_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 private:  
57     size_t m_size;  
58     const size_t m_blocksize;  
59     const size_t m_res_blocks;  
60 };  
61 #endif
```

6.9 File.cpp

```
1  /**
2   * \file 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::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 #endif
```

6.11 Folder.hpp

```

1  /*****
2  * \file Folder.hpp
3  * \brief Folder class representing a folder in the filesystem
4  *
5  * \author Simon
6  * \date   November 2025
7  *****/
8  #ifndef FOLDER_HPP
9  #define FOLDER_HPP
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 }
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         FSOBJECT::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

```
1  /*****
2  * \file ILink.hpp
3  * \brief Interface for folder-like FSObjects
4  *
5  * \author Simon
6  * \date   November 2025
7  *****/
8  #ifndef ILink_HPP
9  #define ILink_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 ILink
20 {
21 public:
22
23     using Sptr = std::shared_ptr<ILink>;
24
25     /** \brief Get the referenced FSObject
26      * \return Shared pointer to the referenced FSObject or nullptr if expired
27      */
28     virtual FSObj_Sptr GetReferncedFSObject() const =0;
29
30     /** \brief Virtual destructor */
31     virtual ~ILink() = default;
32
33 private:
34 };
35
36 #endif
```

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 GetReferncedFSObject() 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

```
1  /*****  
2  * \file Link.cpp  
3  * \brief A link to another FSObject  
4  *  
5  * \author Simon  
6  * \date   November 2025  
7  *****/  
8  #include "Link.hpp"  
9  #include <stdexcept>  
10  
11 /** \brief Construct a link to another FSObject */  
12 Link::Link(FSObj_Sptr linked_obj, std::string_view name) : FSObject(name)  
13 {  
14     if (linked_obj == nullptr) throw std::invalid_argument(Link::ERROR_NULLPTR);  
15     if (name.empty())         throw std::invalid_argument(Link::ERROR_STRING_EMPTY);  
16  
17     m_Ref = move(linked_obj);  
18 }  
19  
20 /** \brief Cast to ILink */  
21 std::shared_ptr<const ILink> Link::AsLink() const  
22 {  
23     return move(shared_from_this());  
24 }  
25  
26 /** \brief Get referenced FSObject (shared_ptr) or nullptr */  
27 FSObj_Sptr Link::GetReferncedFSObject() const  
28 {  
29     return m_Ref.lock();  
30 }  
31  
32 /** \brief Accept a visitor */  
33 void Link::Accept(IVisitor& visit)  
34 {  
35     visit.Visit(move(shared_from_this()));  
36 }  
37  
38 FSObj_Sptr Link::Clone() const  
39 {  
40     // Call Copy Constructor of Link  
41     return make_shared<Link>(Link::Link(*this));  
42 }
```

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  /*****
2  * \file FilterVisitor.hpp
3  * \brief Visitor that filters filesystem objects based on criteria defines in derived classes
4  *
5  * \author Simon
6  * \date   November 2025
7  *****/
8  #ifndef FILTER_VISITOR_HPP
9  #define FILTER_VISITOR_HPP
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
19 void FilterVisitor::DumpPath(const std::weak_ptr<const FSObject> & fsobj, std::ostream& ost) const
20 {
21     // end recursion on expired weak pointer
22     if (fsobj.expired()) return;
23
24     const auto obj = fsobj.lock();
25     if (!obj) return; // defensive: lock could fail
26
27     // first dump parent path
28     DumpPath(obj->GetParent(), ost);
29
30     if (!ost.good()) throw std::invalid_argument(FilterVisitor::ERROR_BAD_OSTREAM);
31
32     ost << "\\\" << obj->GetName();
33
34     const std::shared_ptr<const ILink> link_ptr = obj->AsLink();
35
36     if (link_ptr) {
37         const FSObject::Sptr linked_obj = link_ptr->GetReferncedFSObject();
38         if (linked_obj) {
39             ost << "└─>" << linked_obj->GetName();
40         }
41         else {
42             ost << "└─>" << "linked_Object_Expired!";
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 /** \brief Visit a file and if it matches add to filtered container */
54 void FilterVisitor::Visit(const std::shared_ptr<const File> file)
55 {
56     if (file == nullptr) throw std::invalid_argument(ERROR_NULLPTR);
57
58     // if file matches filter add to container
59     if (DoFilter(file))
60     {
61         m_FilterCont.emplace_back(file);
62     }
63 }
64
65 /** \brief Visit a link and if it matches add to filtered container */
66 void FilterVisitor::Visit(const std::shared_ptr<const Link> link)
67 {
68     if (link == nullptr) throw std::invalid_argument(ERROR_NULLPTR);
69
70     // if link matches filter add to container
71     if (DoFilter(link))
72     {

```

```
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 /** \brief Return the filtered objects container */
89 const FilterVisitor::TContFSobj& FilterVisitor::GetFilteredObjects() const
90 {
91     return m_FilterCont;
92 }
```

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     /** \brief Do filter check for files  
34     * \param file File to check  
35     * \return true if file size is within range  
36     */  
37     virtual bool DoFilter(const std::shared_ptr<const File>& file) const override;  
38  
39     /** \brief Links are not accepted by this filter  
40     * \param link Link to check  
41     * \return false always  
42     */  
43     virtual bool DoFilter(const std::shared_ptr<const Link>& link) const override;  
44  
45 private:  
46     // cannot be const because there are checks in the constructor  
47     size_t m_MinSize;  
48     size_t m_MaxSize;  
49 };  
50 #endif
```

6.20 FilterFileVisitor.cpp

```
1  /**
2   * \file FilterFileVisitor.cpp
3   * \brief Visitor that filters files by size range
4   *
5   * \author Simon
6   * \date   November 2025
7   */
8  #include "FilterFileVisitor.hpp"
9  #include "Folder.hpp"
10 #include "File.hpp"
11 #include "Link.hpp"
12
13 /** \brief Construct filter with size bounds */
14 FilterFileVisitor::FilterFileVisitor(const size_t min, const size_t max)
15 {
16     if (min >= max) throw std::invalid_argument(ERROR_INVALID_SIZE_RANGE);
17
18     m_MinSize = min;
19     m_MaxSize = max;
20 }
21
22 /** \brief Accept files whose size is within range */
23 bool FilterFileVisitor::DoFilter(const std::shared_ptr<const File>& file) const
24 {
25     if (file == nullptr) throw std::invalid_argument(ERROR_NULLPTR);
26
27     return file->GetSize() >= m_MinSize && file->GetSize() <= m_MaxSize;
28 }
29
30 /** \brief Links are not accepted by file filter */
31 bool FilterFileVisitor::DoFilter(const std::shared_ptr<const Link>& link) const
32 {
33     if (link == nullptr) throw std::invalid_argument(ERROR_NULLPTR);
34
35     return false;
36 }
```

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  /*****  
2  * \file   FilterLinkVisitor.cpp  
3  * \brief  Visitor that filters links in the filesystem  
4  *  
5  * \author Simon  
6  * \date   December 2025  
7  *****/  
8  #include "FilterLinkVisitor.hpp"  
9  #include <cassert>  
10 #include <stdexcept>  
11  
12 /** \brief Files are not accepted by link filter */  
13 bool FilterLinkVisitor::DoFilter(const std::shared_ptr<const File>& file) const  
14 {  
15     if(file == nullptr) throw std::invalid_argument(ERROR_NULLPTR);  
16     return false;  
17 }  
18  
19 /** \brief Links are accepted by link filter */  
20 bool FilterLinkVisitor::DoFilter(const std::shared_ptr<const Link>& link) const  
21 {  
22     if(link == nullptr) throw std::invalid_argument(ERROR_NULLPTR);  
23     return true;  
24 }  
25
```

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 fobj Shared pointer to object
50     */
51     void Dump(const std::shared_ptr<const FSObject> fobj);
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
19 /** \brief Visit folder and dump its path */
20 void DumpVisitor::Visit(const std::shared_ptr<const Folder> folder)
21 {
22     if (m_ost.fail()) throw std::invalid_argument(ERROR_BAD_OSTREAM);
23     if (folder == nullptr) throw std::invalid_argument(ERROR_NULLPTR);
24
25     Dump(folder);
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  /*****  
2  * \file   main.cpp  
3  * \brief  Testdriver for the filesystem  
4  *  
5  * \author Simon  
6  * \date   December 2025  
7  *****/  
8  
9  #include <iostream>  
10 #include <string>  
11 #include <memory>  
12 #include "FSObject.hpp"  
13 #include "IFolder.hpp"  
14 #include "ILink.hpp"  
15 #include "FSObjectFactory.hpp"  
16 #include "DumpVisitor.hpp"  
17 #include "FilterFileVisitor.hpp"  
18 #include "FilterLinkVisitor.hpp"  
19 #include "Filesystem.hpp"  
20 #include <cassert>  
21 #include <sstream>  
22 #include "Test.hpp"  
23 #include "fstream"  
24 #include "vld.h"  
25  
26 using namespace std;  
27  
28 #define WriteOutputFile ON  
29  
30 static bool TestDumpVisitor(ostream& ost);  
31 static bool TestFilterLinkVisitor(ostream& ost);  
32 static bool TestFilterFileVisitor(ostream& ost);  
33 static bool TestVisitor(ostream& ost, IVisitor & visitor);  
34 static bool TestFactory(ostream& ost);  
35 static bool TestLink(ostream& ost);  
36 static bool TestFolder(ostream& ost);  
37 static bool TestFile(ostream& ost);  
38 static bool TestFileSystem(ostream& ost);  
39  
40 int main()  
41 {  
42  
43     ofstream output{ "Testoutput.txt" };  
44     if (!output.is_open()) {  
45         cerr << "Konnte_Testoutput.txt_nicht_oeffnen" << TestCaseFail;  
46         return 1;  
47     }  
48  
49     try {  
50         DumpVisitor visitor(std::cout);  
51  
52         FilterLinkVisitor filter_link_visitor;  
53  
54         FilterFileVisitor filter_file_visitor(4096, 16384);  
55  
56         Filesystem homework;  
57  
58         homework.SetFactory(std::make_unique<FSObjectFactory>());  
59         homework.CreateTestFilesystem();  
60  
61         homework.Work(visitor);  
62  
63         std::cout << "-----" << std::endl;  
64         homework.Work(filter_link_visitor);  
65  
66         filter_link_visitor.DumpFiltered(std::cout);  
67  
68         std::cout << "-----" << std::endl;  
69  
70         homework.Work(filter_file_visitor);  
71  
72         filter_file_visitor.DumpFiltered(std::cout);
```

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

```
148     string error_msg;
149
150     try {
151         FSObjectFactory factory;
152         FSObject::Sptr root_folder = factory.CreateFolder("root");
153         FSObject::Sptr sub_folder = factory.CreateFolder("sub_folder");
154         FSObject::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     catch (const string& err) {
174         error_msg = err;
175     }
176     catch (bad_alloc const& error) {
177         error_msg = error.what();
178     }
179     catch (const exception& err) {
180         error_msg = err.what();
181     }
182     catch (...) {
183         error_msg = "Unhandelt_Exception";
184     }
185
186     TestOK = TestOK && check_dump(ost, "Test_Exception_in_TestCase", true, error_msg.empty());
187     error_msg.clear();
188
189     try {
190
191         FSObjectFactory factory;
192         FSObject::Sptr root_folder = factory.CreateFolder("root");
193
194         stringstream result;
195
196         result.setstate(ios::badbit);
197
198         DumpVisitor dumper(result);
199
200         root_folder->Accept(dumper); // <= sould throw Exception bad Ostream
201
202     }
203     catch (const string& err) {
204         error_msg = err;
205     }
206     catch (bad_alloc const& error) {
207         error_msg = error.what();
208     }
209     catch (const exception& err) {
210         error_msg = err.what();
211     }
212     catch (...) {
213         error_msg = "Unhandelt_Exception";
214     }
215
216     TestOK = TestOK && check_dump(ost, "Test_Exception_Bad_Ostream_in_DumpVisitor", DumpVisitor::
217         ERROR_BAD_OSTREAM, error_msg);
218     error_msg.clear();
219
220     ost << TestEnd;
221
```

```
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     try {
235         FSObjectFactory factory;
236         FSObject::Sptr root_folder = factory.CreateFolder("root");
237         FSObject::Sptr sub_folder = factory.CreateFolder("sub_folder");
238         FSObject::Sptr sub_sub_folder = factory.CreateFolder("sub_sub_folder");
239         File::Sptr file = make_shared<File>("file1.txt", 2048);
240         Link::Sptr link = make_shared<Link>(file, "LinkToFile1");
241         sub_sub_folder->AsFolder()->Add(file);
242         sub_sub_folder->AsFolder()->Add(link);
243         sub_folder->AsFolder()->Add(sub_sub_folder);
244         root_folder->AsFolder()->Add(sub_folder);
245
246         FilterLinkVisitor link_filter;
247
248         root_folder->Accept(link_filter);
249
250         TestOK = TestOK && check_dump(ost, "FilterLinkVisitor_Test_filtered_amount",
251                                     static_cast<size_t>(1), link_filter.GetFilteredObjects().size());
252         TestOK = TestOK && check_dump(ost, "FilterLinkVisitor_Test_filtered_obj", link->
253                                     GetReferncedFSObject()->GetName(), link_filter.GetFilteredObjects().cbegin()->lock
254                                     ()->AsLink()->GetReferncedFSObject()->GetName());
255
256         stringstream result;
257         stringstream expected;
258
259         link_filter.DumpFiltered(result);
260
261         expected << "\\root\\sub_folder\\sub_sub_folder\\LinkToFile1_>_file1.txt" << std::endl;
262
263         TestOK = TestOK && check_dump(ost, "Filter_Link_Visitor_Test_Dump", expected.str(),
264                                     result.str());
265
266     }
267     catch (const string& err) {
268         error_msg = err;
269     }
270     catch (bad_alloc const& error) {
271         error_msg = error.what();
272     }
273     catch (const exception& err) {
274         error_msg = err.what();
275     }
276     catch (...) {
277         error_msg = "Unhandelt_Exception";
278     }
279
280     TestOK = TestOK && check_dump(ost, "Test_for_Exception_in_Testcase", true, error_msg.empty());
281     error_msg.clear();
282
283     try {
284         FilterLinkVisitor link_filter{};
285
286         stringstream result;
287         result.setstate(ios::badbit);
288
289         link_filter.DumpFiltered(result);
290
291     }
292     catch (const string& err) {
293         error_msg = err;
294     }
295 }
```

```
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 = "Unhandelt_Exception";
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>
349             >(2), file_filter.GetFilteredObjects().size());
350         TestOK = TestOK && check_dump(ost, "FilterFileVisitor_Test_for_filtered_file", file2->GetName(),
351             file_filter.GetFilteredObjects().cbegin()->lock()->GetName());
352         TestOK = TestOK && check_dump(ost, "FilterFileVisitor_Test_for_filtered_file", file->GetName(),
353             file_filter.GetFilteredObjects().crbegin()->lock()->GetName());
354
355         stringstream result;
356         stringstream expected;
357
358         file_filter.DumpFiltered(result);
359
360         expected << "\\root\\file3.txt" << std::endl
361             << "\\root\\sub_folder\\sub_sub_folder\\file1.txt" << std::endl;
362
363         TestOK = TestOK && check_dump(ost, "Filter_File_Visitor_Test_Dump", expected.str(), result.str
364             ());
```

```
362
363     }
364     catch (const string& err) {
365         error_msg = err;
366     }
367     catch (bad_alloc const& error) {
368         error_msg = error.what();
369     }
370     catch (const exception& err) {
371         error_msg = err.what();
372     }
373     catch (...) {
374         error_msg = "Unhandelt_Exception";
375     }
376
377     TestOK = TestOK && check_dump(ost, "Test_for_Exception_in_Testcase", true, error_msg.empty());
378     error_msg.clear();
379
380     try {
381
382         FilterFileVisitor file_filter(1,2);
383
384         stringstream result;
385         result.setstate(ios::badbit);
386
387         file_filter.DumpFiltered(result);
388     }
389     catch (const string& err) {
390         error_msg = err;
391     }
392     catch (bad_alloc const& error) {
393         error_msg = error.what();
394     }
395     catch (const exception& err) {
396         error_msg = err.what();
397     }
398     catch (...) {
399         error_msg = "Unhandelt_Exception";
400     }
401
402     TestOK = TestOK && check_dump(ost, "Test_for_Exception_in_Dump_with_bad_Ostream", error_msg,
403                                 FilterLinkVisitor::ERROR_BAD_OSTREAM);
404     error_msg.clear();
405
406     try {
407
408         FilterFileVisitor file_filter( 2,1 ); // <= should throw invalid size range
409     }
410     catch (const string& err) {
411         error_msg = err;
412     }
413     catch (bad_alloc const& error) {
414         error_msg = error.what();
415     }
416     catch (const exception& err) {
417         error_msg = err.what();
418     }
419     catch (...) {
420         error_msg = "Unhandelt_Exception";
421     }
422
423     TestOK = TestOK && check_dump(ost, "Test_for_Exception_in_Filter_File_Visitor_CTOR", error_msg,
424                                 FilterFileVisitor::ERROR_INVALID_SIZE_RANGE);
425     error_msg.clear();
426
427     ost << TestEnd;
428     return TestOK;
429 }
430
431 bool TestVisitor(ostream& ost, IVisitor& visit)
432 {
433     assert(ost.good());
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     catch (const string& err) {
450         error_msg = err;
451     }
452     catch (bad_alloc const& error) {
453         error_msg = error.what();
454     }
455     catch (const exception& err) {
456         error_msg = err.what();
457     }
458     catch (...) {
459         error_msg = "Unhandelt_Exception";
460     }
461
462     TestOK = TestOK && check_dump(ost, "Test_Exception_nullptr_in_Visit_File", DumpVisitor::
463         ERROR_NULLPTR, error_msg);
464     error_msg.clear();
465
466     try {
467
468         stringstream result;
469
470         Folder::Sptr folder = nullptr;
471
472         visit.Visit(folder); // <= sould throw Exception Nullptr
473     }
474     catch (const string& err) {
475         error_msg = err;
476     }
477     catch (bad_alloc const& error) {
478         error_msg = error.what();
479     }
480     catch (const exception& err) {
481         error_msg = err.what();
482     }
483     catch (...) {
484         error_msg = "Unhandelt_Exception";
485     }
486
487     TestOK = TestOK && check_dump(ost, "Test_Exception_nullptr_in_Visit_Folder", DumpVisitor::
488         ERROR_NULLPTR, error_msg);
489     error_msg.clear();
490
491     try {
492
493         stringstream result;
494
495         Link::Sptr lnk = nullptr;
496
497         visit.Visit(lnk); // <= sould throw Exception Nullptr
498     }
499     catch (const string& err) {
500         error_msg = err;
501     }
502     catch (bad_alloc const& error) {
503         error_msg = error.what();
504     }
505     catch (const exception& err) {
506         error_msg = err.what();
507     }
```



```
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     return TestOK;
519 }
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() !=
538             nullptr);
539         TestOK = TestOK && check_dump(ost, "Test_if_Folder_was_constructed", true, folder->AsFolder()
540             != nullptr);
541     }
542     catch (const string& err) {
543         error_msg = err;
544     }
545     catch (bad_alloc const& error) {
546         error_msg = error.what();
547     }
548     catch (const exception& err) {
549         error_msg = err.what();
550     }
551     catch (...) {
552         error_msg = "Unhandelt_Exception";
553     }
554
555     TestOK = TestOK && check_dump(ost, "Test_for_Exception_in_Tesstcase", true, error_msg.empty());
556     error_msg.clear();
557
558     try {
559         FSObjectFactory fact;
560         File::Sptr file= nullptr;
561         FSObj_Sptr Lnk = fact.CreateLink("Link_to_File", file);
562
563     }
564     catch (const string& err) {
565         error_msg = err;
566     }
567     catch (bad_alloc const& error) {
568         error_msg = error.what();
569     }
570     catch (const exception& err) {
571         error_msg = err.what();
572     }
573     catch (...) {
574         error_msg = "Unhandelt_Exception";
575     }
576
577     TestOK = TestOK && check_dump(ost, "Test_Exception_nullptr_CTOR_Link", Link::ERROR_NULLPTR,
578         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->
            GetReferncedFSObject()->GetName());
604         TestOK = TestOK && check_dump(ost, "Test_normal_CTOR_Link", link_name, link->GetName());
605
606     }
607     catch (const string& err) {
608         error_msg = err;
609     }
610     catch (bad_alloc const& error) {
611         error_msg = error.what();
612     }
613     catch (const exception& err) {
614         error_msg = err.what();
615     }
616     catch (...) {
617         error_msg = "Unhandelt_Exception";
618     }
619
620     TestOK = TestOK && check_dump(ost, "Test_normal_CTOR_Link_-_error_buffer", true, error_msg.empty())
        ;
621     error_msg.clear();
622
623     // test Copy CTOR of Link
624     try
625     {
626         std::string_view folder_name = "MyFolder";
627         std::string_view link_name = "LinkToMyFolder";
628         Folder::Sptr folder = make_shared<Folder>(folder_name);
629         Link::Sptr link = make_shared<Link>(folder, link_name);
630         Folder::Sptr parent_folder = make_shared<Folder>("AnotherFolder");
631         parent_folder->Add(link); // set parent of link
632
633         // Call Copy CTOR
634         Link::Sptr link_copy = make_shared<Link>(*link);
635
636         TestOK = TestOK && check_dump(ost, "Test_Copy_CTOR_of_Link", link->GetReferncedFSObject(),
            link_copy->GetReferncedFSObject());
637
638         // modify copied link referenced FSObject name
639         link_copy->GetReferncedFSObject()->SetName("NewFolderName");
640
641         TestOK = TestOK && check_dump(ost, "Test_for_shallow_Copy_of_Link", link->GetReferncedFSObject()
            (), link_copy->GetReferncedFSObject());
642
643         parent_folder->SetName("Modified");
644
645         TestOK = TestOK && check_dump(ost, "Test_for_parent_of_Copied_Link", parent_folder->GetName(),
            link_copy->GetParent().lock()->GetName());
646
647     }
648     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 = "Unhandelt_Exception";
659 }
660
661 TestOK = TestOK && check_dump(ost, "Test_normal_COPY_CTOR_Link_-_error_buffer", true, error_msg.
662     empty());
663 error_msg.clear();
664
665 // test Assign Op of Link
666 try
667 {
668     std::string_view folder_name = "MyFolder";
669     std::string_view link_name = "LinkToMyFolder";
670     Folder::Sptr folder = make_shared<Folder>(folder_name);
671     Link::Sptr link = make_shared<Link>(folder, link_name);
672     Folder::Sptr another_folder = make_shared<Folder>("AnotherFolder");
673     another_folder->Add(link); // set parent of link
674
675     Link::Sptr link_ass = make_shared<Link>(folder, "Ass_Link");
676
677     *link_ass = *link;
678
679     TestOK = TestOK && check_dump(ost, "Test_Assign_Op_of_Link", link->GetReferncedFSObject(),
680         link_ass->GetReferncedFSObject());
681
682     another_folder->SetName("Modified");
683
684     TestOK = TestOK && check_dump(ost, "Test_Assign_Op_for_Parent_of_Link", another_folder->GetName
685         (), link_ass->GetParent().lock()->GetName());
686
687 }
688 catch (const string& err) {
689     error_msg = err;
690 }
691 catch (bad_alloc const& error) {
692     error_msg = error.what();
693 }
694 catch (const exception& err) {
695     error_msg = err.what();
696 }
697 catch (...) {
698     error_msg = "Unhandelt_Exception";
699 }
700
701 TestOK = TestOK && check_dump(ost, "Test_Assign_Op_Link_-_error_buffer", true, error_msg.empty());
702 error_msg.clear();
703
704 // test Self Assign of Link
705 try
706 {
707     std::string_view folder_name = "MyFolder";
708     std::string_view link_name = "LinkToMyFolder";
709     Folder::Sptr folder = make_shared<Folder>(folder_name);
710     Link::Sptr link = make_shared<Link>(folder, link_name);
711
712     *link = *link; // <= could throw
713
714 }
715 catch (const string& err) {
716     error_msg = err;
717 }
718 catch (bad_alloc const& error) {
719     error_msg = error.what();
720 }
721 catch (const exception& err) {
722     error_msg = err.what();
723 }
724 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 error_msg.clear();
726
727 // test link to nullptr
728 try
729 {
730     Link::Sptr link = make_shared<Link>(nullptr, "LinkToNothing");
731 }
732 catch (const string& err) {
733     error_msg = err;
734 }
735 catch (bad_alloc const& error) {
736     error_msg = error.what();
737 }
738 catch (const exception& err) {
739     error_msg = err.what();
740 }
741 catch (...) {
742     error_msg = "Unhandelt_Exception";
743 }
744
745 TestOK = TestOK && check_dump(ost, "Test_Exception_nullable_CTOR_Link", Link::ERROR_NULLPTR,
    error_msg);
746 error_msg.clear();
747
748 // test Link with empty string
749 try
750 {
751     File::Sptr file = make_shared<File>("file1.txt", 2048);
752     Link::Sptr link = make_shared<Link>(file, "");
753 }
754 catch (const string& err) {
755     error_msg = err;
756 }
757 catch (bad_alloc const& error) {
758     error_msg = error.what();
759 }
760 catch (const exception& err) {
761     error_msg = err.what();
762 }
763 catch (...) {
764     error_msg = "Unhandelt_Exception";
765 }
766
767 TestOK = TestOK && check_dump(ost, "Test_Exception_empty_string_CTOR_Link", Link::
    ERROR_STRING_EMPTY, error_msg);
768 error_msg.clear();
769
770 // test Link GetReferencedFSObject
771 try
772 {
773     File::Sptr file = make_shared<File>("file1.txt", 2048);
774     Link::Sptr link = make_shared<Link>(file, file->GetName());
775
776     FSObj_Sptr ref = link->GetReferncedFSObject(); // <= should be File not Folder
777
778     TestOK = TestOK && check_dump(ost, "Test_GetReferencedFSObject", file->GetName(), ref->GetName
        ());
779 }
780 catch (const string& err) {
781     error_msg = err;
782 }
783 catch (bad_alloc const& error) {
784     error_msg = error.what();
785 }
786 catch (const exception& err) {
787     error_msg = err.what();
788 }
789 catch (...) {
790     error_msg = "Unhandelt_Exception";
791 }
```

```

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->GetReferncedFSObject()->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->GetReferncedFSObject() != nullptr);
822         } // file goes out of scope here
823
824         FSObj_Sptr expired_ref = link->GetReferncedFSObject();
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     error_msg.clear();
833
834     //AsLink() method returns valid pointer
835     try
836     {
837         File::Sptr file = make_shared<File>("file.txt", 2048);
838         Link::Sptr link = make_shared<Link>(file, "TestLink");
839
840         std::shared_ptr<const ILink> ilink = link->AsLink();
841         TestOK = TestOK && check_dump(ost, "Test_AsLink()_returns_valid_pointer",
842             true, ilink != nullptr);
843         TestOK = TestOK && check_dump(ost, "Test_AsLink()_reference_matches",
844             file->GetName(), ilink->GetReferncedFSObject()->GetName());
845     }
846     catch (const exception& err) {
847         error_msg = err.what();
848     }
849     TestOK = TestOK && check_dump(ost, "Test_AsLink()_-_error_buffer", true, error_msg.empty());
850     error_msg.clear();
851
852     //Link SetName functionality
853     try
854     {
855         File::Sptr file = make_shared<File>("file.txt", 2048);
856         Link::Sptr link = make_shared<Link>(file, "OriginalName");
857
858         link->SetName("NewName");
859         TestOK = TestOK && check_dump(ost, "Test_Link_SetName",
860             string_view("NewName"), link->GetName());
861     }
862     catch (const exception& err) {
863         error_msg = err.what();
864     }
865     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(""); // should throw
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, error_msg.empty());
898 error_msg.clear();
899
900 ost << TestEnd;
901 return TestOK;
902 }
903 bool TestFolder(ostream& ost)
904 {
905     assert(ost.good());
906
907     ost << TestStart;
908
909     bool TestOK = true;
910     string error_msg;
911
912     // test folder as intended
913     try
914     {
915         string_view folder_name = "MyFolder";
916         Folder::Sptr folder = make_shared<Folder>(folder_name);
917         TestOK = TestOK && check_dump(ost, "Test_normal_CTOR_Folder", folder_name, folder->GetName());
918
919         File::Sptr file1 = make_shared<File>("file1.txt", 2048);
920         File::Sptr file2 = make_shared<File>("file2.txt", 4096);
921
922         folder->Add(file1);
923         folder->Add(file2);
924
925         FSOBJECT::Sptr err_file = folder->GetChild(2); // <= should be nullptr
926         FSOBJECT::Sptr shared_null = nullptr;
927
928         TestOK = TestOK && check_dump(ost, "Get_Child_from_folder", static_pointer_cast<FSObject>(file1),
929             folder->GetChild(0));
930         TestOK = TestOK && check_dump(ost, "Get_next_Child_from_folder", static_pointer_cast<FSObject>(
931             file2), folder->GetChild(1));
932         TestOK = TestOK && check_dump(ost, "Get_Child_for_invalid_index", err_file, shared_null);
933     }
934     catch (const string& err) {
935         error_msg = err;
936     }
937     catch (bad_alloc const& error) {
938         error_msg = error.what();
939     }
940     catch (const exception& err) {
```

```

939         error_msg = err.what();
940     }
941     catch (...) {
942         error_msg = "Unhandelt_Exception";
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     {
951         string_view folder_name = "MyFolder";
952         Folder::Sptr folder = make_shared<Folder>( folder_name );
953         File::Sptr file1 = make_shared<File>("file1.txt", 2048);
954         File::Sptr file2 = make_shared<File>("file2.txt", 4096);
955         Folder::Sptr sub_folder = make_shared<Folder>("SubFolder");
956         File::Sptr sub_file = make_shared<File>("sub_file.txt", 1024);
957
958         folder->Add(file1);
959         folder->Add(file2);
960         folder->Add(sub_folder);
961         sub_folder->Add(sub_file);
962
963         // Call Copy Ctor
964         FSObject::Sptr folder_copy = folder->Clone();
965
966         TestOK = TestOK && check_dump(ost, "Test_Copy_Ctor_Folder_-_Child_0", file1->GetName(),
967                                     folder_copy->AsFolder()->GetChild(0)->GetName());
968         TestOK = TestOK && check_dump(ost, "Test_Copy_Ctor_Folder_-_Sub_Folder_File", sub_file
969                                     ->GetName(), folder_copy->AsFolder()->GetChild(2)->AsFolder()->GetChild(0)->
970                                     GetName());
971
972         file1->SetName("modified_file1.txt");
973         sub_file->SetName("modified_sub.txt");
974
975         TestOK = TestOK && check_dump(ost, "Test_Copy_Ctor_Folder_test_for_Deep_Copy", true,
976                                     file1->GetName() != folder_copy->AsFolder()->GetChild(0)->GetName());
977         TestOK = TestOK && check_dump(ost, "Test_Copy_Ctor_Folder_test_for_Deep_Copy_in_Sub_
978                                     Folder_File", true, sub_file->GetName() != folder_copy->AsFolder()->GetChild(2)->
979                                     AsFolder()->GetChild(0)->GetName());
980
981         TestOK = TestOK && check_dump(ost, "Test_Parent_of_Copied_Folder", static_pointer_cast<
982                                     FSObject>(folder_copy), folder_copy->AsFolder()->GetChild(0)->GetParent().lock());
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 = "Unhandelt_Exception";
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    {
1003        string_view folder_name = "MyFolder";
1004        Folder::Sptr folder = make_shared<Folder>(folder_name);
1005        File::Sptr file1 = make_shared<File>("file1.txt", 2048);
1006        File::Sptr file2 = make_shared<File>("file2.txt", 4096);
1007
1008        folder->Add(file1);
1009        folder->Add(file2);
1010
1011        Folder::Sptr folder_ass = make_shared<Folder>("Ass_folder");
1012        *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     catch (const string& err) {
1018         error_msg = err;
1019     }
1020     catch (bad_alloc const& error) {
1021         error_msg = error.what();
1022     }
1023     catch (const exception& err) {
1024         error_msg = err.what();
1025     }
1026     catch (...) {
1027         error_msg = "Unhandelt_Exception";
1028     }
1029
1030     TestOK = TestOK && check_dump(ost, "Test_Folder_-_error_buffer", error_msg.empty(), true);
1031     error_msg.clear();
1032
1033     // test Assign Operator of Folder Self Assign
1034     try
1035     {
1036         string_view folder_name = "MyFolder";
1037         Folder::Sptr folder = make_shared<Folder>(folder_name);
1038         File::Sptr file1 = make_shared<File>("file1.txt", 2048);
1039         File::Sptr file2 = make_shared<File>("file2.txt", 4096);
1040
1041         folder->Add(file1);
1042         folder->Add(file2);
1043         *folder = *folder;
1044
1045         TestOK = TestOK && check_dump(ost, "Test_Self_Assign_Folder_-_Child_0",
1046                                         static_pointer_cast<FSObject>(file1), folder->GetChild(0));
1047
1048     }
1049     catch (const string& err) {
1050         error_msg = err;
1051     }
1052     catch (bad_alloc const& error) {
1053         error_msg = error.what();
1054     }
1055     catch (const exception& err) {
1056         error_msg = err.what();
1057     }
1058     catch (...) {
1059         error_msg = "Unhandelt_Exception";
1060     }
1061
1062     TestOK = TestOK && check_dump(ost, "Test_Folder_-_error_buffer", error_msg.empty(), true);
1063     error_msg.clear();
1064
1065     // test remove child
1066     try
1067     {
1068         Folder::Sptr folder = make_shared<Folder>("MyFolder");
1069         File::Sptr file1 = make_shared<File>("file1.txt", 2048);
1070         File::Sptr file2 = make_shared<File>("file2.txt", 4096);
1071         folder->Add(file1);
1072         folder->Add(file2);
1073         folder->Remove(file1);
1074         TestOK = TestOK && check_dump(ost, "Test_Remove_Child_from_Folder", static_pointer_cast<
1075                                         FSObject>(file2), folder->GetChild(0));
1076
1077     }
1078     catch (const string& err) {
1079         error_msg = err;
1080     }
1081     catch (bad_alloc const& error) {
1082         error_msg = error.what();
1083     }
```



```
1078     }
1079     catch (const exception& err) {
1080         error_msg = err.what();
1081     }
1082     catch (...) {
1083         error_msg = "Unhandelt_Exception";
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         FSObject::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 = "Unhandelt_Exception";
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 = "Unhandelt_Exception";
1128     }
1129
1130     TestOK = TestOK && check_dump(ost, "Test_Folder_-_CTOR_with_empty_string", FSObject::
        ERROR_STRING_EMPTY, error_msg);
1131     error_msg.clear();
1132
1133     //Nested folder structure
1134     try
1135     {
1136         Folder::Sptr root = make_shared<Folder>("root");
1137         Folder::Sptr sub1 = make_shared<Folder>("sub1");
1138         Folder::Sptr sub2 = make_shared<Folder>("sub2");
1139
1140         root->Add(sub1);
1141         sub1->Add(sub2);
1142
1143         TestOK = TestOK && check_dump(ost, "Test_nested_folders_-_root_has_sub1",
            sub1, static_pointer_cast<Folder>(root->GetChild(0)));
1144         TestOK = TestOK && check_dump(ost, "Test_nested_folders_-_sub1_has_sub2",
            sub2, static_pointer_cast<Folder>(sub1->GetChild(0)));
1145     }
1146     catch (const exception& err) {
1147         error_msg = err.what();
1148     }
1149
1150     TestOK = TestOK && check_dump(ost, "Test_nested_folders_-_error_buffer", true, error_msg.empty());
1151 }
```

```

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_sptr = parent_wptr.lock();
1163
1164         TestOK = TestOK && check_dump(ost, "Test_parent_pointer_set_on_Add",
1165             parent->GetName(), parent_sptr->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         error_msg = err.what();
1250     }
1251     TestOK = TestOK && check_dump(ost, "Test_Accept_visitor_error_buffer", true, error_msg.empty());
1252     error_msg.clear();
1253
1254     //SetName on folder
1255     try
1256     {
1257         Folder::Sptr folder = make_shared<Folder>("original");
1258         folder->SetName("renamed");
1259
1260         TestOK = TestOK && check_dump(ost, "Test_Folder_SetName",
1261                                     string_view("renamed"), folder->GetName());
1262     }
1263     catch (const exception& err) {
1264         error_msg = err.what();
1265     }
1266     TestOK = TestOK && check_dump(ost, "Test_Folder_SetName_error_buffer", true, error_msg.empty());
1267     error_msg.clear();
1268
1269     ost << TestEnd;
1270     return TestOK;
1271 }
1272 bool TestFile(ostream& ost)
1273 {
1274     assert(ost.good());
1275
1276     ost << TestStart;
1277
1278     bool TestOK = true;
1279     string error_msg;
1280
1281     // File as intended
1282     try
1283     {
1284         string_view file_name = "file1.txt";
1285         size_t block_size = 2048;
1286         size_t res_blocks = 20;
1287         File::Sptr file = make_shared<File>(file_name, res_blocks, block_size);
1288
1289         TestOK = TestOK && check_dump(ost, "Test_normal_CTOR_File", file_name, file->GetName());
1290         TestOK = TestOK && check_dump(
1291             ost, "Test_normal_CTOR_File_size",
1292             static_cast<size_t>(0), file->GetSize());
1293
1294         // Write to file
1295         size_t write_size = 4096;
1296         file->Write(write_size);
1297         TestOK = TestOK && check_dump(ost, "Test_normal_write_file_size", write_size, file->GetSize()
1298                                     );
1299     }
1300     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 = "Unhandelt_Exception";
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 = "Unhandelt_Exception";
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 = "Unhandelt_Exception";
1371 }
1372
1373 TestOK = TestOK && check_dump(ost, "Test_CTOR_Empty_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     error_msg = err.what();
1389 }
1390 TestOK = TestOK && check_dump(ost, "Test_multiple_writes_-_error_buffer", true, error_msg.empty());
1391 error_msg.clear();
1392
1393 // Write exactly to capacity
1394 try
1395 {
1396     size_t blocks = 5;
1397     size_t blocksize = 1024;
1398     File::Sptr file = make_shared<File>("exact.txt", blocks, blocksize);
1399
1400     file->Write(blocks * blocksize); // Write exactly to capacity
1401
1402     TestOK = TestOK && check_dump(ost, "Test_write_to_exact_capacity",
1403         blocks * blocksize, file->GetSize());
1404 }
1405 catch (const exception& err) {
1406     error_msg = err.what();
1407 }
1408 TestOK = TestOK && check_dump(ost, "Test_exact_capacity_-_error_buffer", true, error_msg.empty());
1409 error_msg.clear();
1410
1411 // Write exceeds capacity (should throw)
1412 try
1413 {
1414     File::Sptr file = make_shared<File>("overflow.txt", 2, 1024);
1415     file->Write(3000); // Exceeds 2 * 1024 = 2048
1416 }
1417 catch (const exception& err) {
1418     error_msg = err.what();
1419 }
1420 TestOK = TestOK && check_dump(ost, "Test_write_exceeds_capacity",
1421     File::ERR_OUT_OF_SPACE, error_msg);
1422 error_msg.clear();
1423
1424 // Write zero bytes
1425 try
1426 {
1427     File::Sptr file = make_shared<File>("zero.txt", 10, 2048);
1428     file->Write(0);
1429
1430     TestOK = TestOK && check_dump(ost, "Test_write_zero_bytes",
1431         static_cast<size_t>(0), file->GetSize());
1432 }
1433 catch (const exception& err) {
1434     error_msg = err.what();
1435 }
1436 TestOK = TestOK && check_dump(ost, "Test_write_zero_-_error_buffer", true, error_msg.empty());
1437 error_msg.clear();
1438
1439 // Multiple writes approaching capacity
1440 try
1441 {
1442     File::Sptr file = make_shared<File>("approach.txt", 3, 1000);
1443     file->Write(1000);
1444     file->Write(1000);
1445     file->Write(1000); // Total = 3000, capacity = 3000
1446
1447     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     // Write after reaching capacity (should throw)
1457     try
1458     {
1459         File::Sptr file = make_shared<File>("full.txt", 2, 1024);
1460         file->Write(2048); // Fill to capacity
1461         file->Write(1);    // Should throw
1462     }
1463     catch (const exception& err) {
1464         error_msg = err.what();
1465     }
1466     TestOK = TestOK && check_dump(ost, "Test_write_when_full", File::ERR_OUT_OF_SPACE, error_msg);
1467     error_msg.clear();
1468     // File with default blocksize (4096)
1469     try
1470     {
1471         File::Sptr file = make_shared<File>("default.txt", 5); // Default blocksize = 4096
1472         file->Write(10000);
1473
1474         TestOK = TestOK && check_dump(ost, "Test_default_blocksize", static_cast<size_t>(10000), file->
1475         GetSize());
1476     }
1477     catch (const exception& err) {
1478         error_msg = err.what();
1479     }
1480     TestOK = TestOK && check_dump(ost, "Test_default_blocksize_-_error_buffer", true, error_msg.empty()
1481     );
1482     error_msg.clear();
1483     // Accept visitor
1484     try
1485     {
1486         File::Sptr file = make_shared<File>("visitor.txt", 10, 2048);
1487         stringstream result;
1488         DumpVisitor visitor(result);
1489
1490         file->Accept(visitor);
1491
1492         TestOK = TestOK && check_dump(ost, "Test_File_Accept_visitor", true, result.str().find("visitor
1493         .txt") != string::npos);
1494     }
1495     catch (const exception& err) {
1496         error_msg = err.what();
1497     }
1498     TestOK = TestOK && check_dump(ost, "Test_File_Accept_-_error_buffer", true, error_msg.empty());
1499     error_msg.clear();
1500     // SetName on file
1501     try
1502     {
1503         File::Sptr file = make_shared<File>("old.txt", 10, 2048);
1504         file->SetName("new.txt");
1505
1506         TestOK = TestOK && check_dump(ost, "Test_File_SetName",
1507         string_view("new.txt"), file->GetName());
1508     }
1509     catch (const exception& err) {
1510         error_msg = err.what();
1511     }
1512     TestOK = TestOK && check_dump(ost, "Test_File_SetName_-_error_buffer", true, error_msg.empty());
1513     error_msg.clear();
1514     // File AsFolder should return nullptr
1515     try
1516     {
1517         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 ==
            nullptr);
1522     }
1523     catch (const exception& err) {
1524         error_msg = err.what();
1525     }
1526     TestOK = TestOK && check_dump(ost, "Test_File_AsFolder_-_error_buffer", true, error_msg.empty());
1527     error_msg.clear();
1528
1529     ost << TestEnd;
1530     return TestOK;
1531 }
1532
1533 bool TestFileSystem(ostream& ost)
1534 {
1535     assert(ost.good());
1536
1537     ost << TestStart;
1538
1539     bool TestOK = true;
1540     string error_msg;
1541
1542     try
1543     {
1544         FileSystem fsys;
1545
1546         fsys.SetFactory(make_unique<FSObjectFactory>());
1547
1548         // build a Test filesystem using the set Factory
1549         fsys.CreateTestFileSystem();
1550
1551         DumpVisitor dumper(ost);
1552
1553         ost << "Dump_of_Test_FileSystem_via_Dump_Visitor:\n\n";
1554
1555         fsys.Work(dumper);
1556
1557         ost << "\n\n";
1558     }
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_buffer_empty", error_msg.
        empty(), true);
1573     error_msg.clear();
1574
1575     try
1576     {
1577         FileSystem fsys;
1578
1579         FSObjectFactory factory;
1580
1581         FSObject::Sptr root = factory.CreateFolder("root");
1582
1583         fsys.SetRoot(move(root));
1584
1585         stringstream result;
1586         stringstream expected;
1587
1588         DumpVisitor dumper(result);
```

```
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_Fileystem_error_buffer_empty", error_msg.
        empty(), true);
1618     error_msg.clear();
1619
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 ,
        error_msg);
1642     error_msg.clear();
1643
1644     try
1645     {
1646         FileSystem fsys;
1647         FSObjectFactory::Uptr factory = nullptr;
1648
1649         fsys.SetFactory(move(factory)); // <= should throw
1650     }
1651     catch (const string& err) {
1652         error_msg = err;
1653     }
1654     catch (bad_alloc const& error) {
1655         error_msg = error.what();
1656     }
1657     catch (const exception& err) {
1658         error_msg = err.what();
1659     }
1660     catch (...) {
1661         error_msg = "Unhandelt_Exception";
1662     }
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
1671         fsys.CreateTestFilesystem(); // <= should throw because no factory set
1672     }
1673     catch (const string& err) {
1674         error_msg = err;
1675     }
1676     catch (bad_alloc const& error) {
1677         error_msg = error.what();
1678     }
1679     catch (const exception& err) {
1680         error_msg = err.what();
1681     }
1682     catch (...) {
1683         error_msg = "Unhandelt_Exception";
1684     }
1685
1686     TestOK = TestOK && check_dump(ost, "Test_Exception_no_Factory_in_Create_Test_FileSystem",
1687                                   FileSystem::ERROR_NULLPTR ,error_msg);
1688     error_msg.clear();
1689     try
1690     {
1691         FileSystem fsys;
1692         DumpVisitor dumper(ost);
1693         fsys.Work(dumper); // <= should throw because root is null
1694     }
1695     catch (const string& err) {
1696         error_msg = err;
1697     }
1698     catch (bad_alloc const& error) {
1699         error_msg = error.what();
1700     }
1701     catch (const exception& err) {
1702         error_msg = err.what();
1703     }
1704     catch (...) {
1705         error_msg = "Unhandelt_Exception";
1706     }
1707
1708     TestOK = TestOK && check_dump(ost, "Test_Exception_Work_with_no_root_set", FileSystem::
1709                                   ERROR_NULLPTR ,error_msg);
1710     error_msg.clear();
1711
1712     ost << TestEnd;
1713
1714     return TestOK;
1715 }
```

6.26 Test.hpp

```

1  /*****
2  * \file   Test.hpp
3  * \brief  File that provides a Test Function with a formatted output
4  *
5  * \author Simon
6  * \date   April 2025
7  *****/
8  #ifndef TEST_HPP
9  #define TEST_HPP
10
11 #include <string>
12 #include <iostream>
13 #include <vector>
14 #include <list>
15 #include <queue>
16 #include <forward_list>
17
18 #define ON 1
19 #define OFF 0
20 #define COLOR_OUTPUT OFF
21
22 // Definitions of colors in order to change the color of the output stream.
23 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 << "*****" << std::endl;
50         ost << "~~~~~TESTCASE_START~" << std::endl;
51         ost << "*****" << std::endl;
52         ost << std::endl;
53     }
54     return ost;
55 }
56
57 inline std::ostream& TestEnd(std::ostream& ost) {
58     if (ost.good()) {
59         ost << std::endl;
60         ost << "*****" << std::endl;
61         ost << std::endl;
62     }
63     return ost;
64 }
65
66 inline std::ostream& TestCaseOK(std::ostream& ost) {
67     #if COLOR_OUTPUT
68         if (ost.good()) {
69             ost << colorGreen() << "TEST_OK!!" << colorWhite() << std::endl;
70         }
71     #else
72

```

```

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 << "_==" << "_Result
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:_" << result << ")" << std::noboolalpha << std::endl << std::endl;
116         }
117 #else
118         if (expected == result) {
119             ostr << testcase << std::endl << "[Test_OK]" << "Result:_(Expected:_" << std::
120             boolalpha << expected << "_==" << "_Result:_" << result << ")" << std::
121             noboolalpha << std::endl << std::endl;
122         }
123         else {
124             ostr << testcase << std::endl << "[Test_FAILED]" << "Result:_(Expected:_" <<
125             std::boolalpha << expected << "_!=" << "_Result:_" << result << ")" << std
126             ::noboolalpha << std::endl << std::endl;
127         }
128 #endif
129
130         if (ostr.fail()) {
131             std::cerr << "Error:_Write_Ostream" << std::endl;
132         }
133     }
134     else {
135         std::cerr << "Error:_Bad_Ostream" << std::endl;
136     }
137     return expected == result;
138 }
139
140 template <typename T1, typename T2>
141 std::ostream& operator<< (std::ostream& ost, const std::pair<T1,T2> & p) {
142     if (!ost.good()) throw std::runtime_error("Error:_bad_Ostream!");
143     ost << "(" << p.first << "," << p.second << ")";
144     return ost;
145 }

```

```
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
```