

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



---

Name: Simon Offenberger / Simon Vogelhuber

Aufwand in h: siehe Doku.

---

Mat.Nr: S2410306027 / S2410306014

Punkte:

---

Übungsgruppe: 1

korrigiert:

---

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

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

Eine Datei hat zusätzlich folgende Eigenschaften:

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

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

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

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

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

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

Name des Elementes auf das verwiesen wird ausgegeben werden.

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

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

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



# **Systemdokumentation Projekt Filesystem**

**Version 1.0**

S. Offenberger, S. Vogelhuber

Hagenberg, 8. Dezember 2025

# Inhaltsverzeichnis

<b>1 Organisatorisches</b>	<b>6</b>
1.1 Team . . . . .	6
1.2 Aufteilung der Verantwortlichkeitsbereiche . . . . .	6
1.3 Aufwand . . . . .	7
<b>2 Anforderungsdefinition (Systemspezifikation)</b>	<b>8</b>
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
<b>3 Systementwurf</b>	<b>11</b>
3.1 Klassendiagramm . . . . .	11
3.2 Designentscheidungen . . . . .	12
3.3 Composite Pattern . . . . .	12
3.4 Factory Pattern . . . . .	13
3.5 Visitor Pattern . . . . .	13
3.6 Template Methode Pattern . . . . .	13
<b>4 Dokumentation der Komponenten (Klassen)</b>	<b>14</b>
<b>5 Testprotokollierung</b>	<b>15</b>
<b>6 Quellcode</b>	<b>25</b>
6.1 Object.hpp . . . . .	25
6.2 FSObjectFactory.hpp . . . . .	26
6.3 FSObjectFactory.cpp . . . . .	27
6.4 Filesystem.hpp . . . . .	28
6.5 Filesystem.cpp . . . . .	29
6.6 FSObject.hpp . . . . .	30

6.7	FSObject.cpp	32
6.8	File.hpp	33
6.9	File.cpp	34
6.10	IFolder.hpp	35
6.11	Folder.hpp	36
6.12	Folder.cpp	37
6.13	ILink.hpp	38
6.14	Link.hpp	39
6.15	Link.cpp	40
6.16	IVisitor.hpp	41
6.17	FilterVisitor.hpp	42
6.18	FilterVisitor.cpp	44
6.19	FilterFileVisitor.hpp	46
6.20	FilterFileVisitor.cpp	47
6.21	FilterLinkVisitor.hpp	48
6.22	FilterLinkVisitor.cpp	49
6.23	DumpVisitor.hpp	50
6.24	DumpVisitor.cpp	51
6.25	main.cpp	52
6.26	Test.hpp	70

# 1 Organisatorisches

## 1.1 Team

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

## 1.2 Aufteilung der Verantwortlichkeitsbereiche

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

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

### 1.3 Aufwand

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

## 2 Anforderungsdefinition (Systemspezifikation)

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

### 2.1 Systemüberblick

Das System verwaltet drei Arten von Dateisystemelementen:

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

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

### 2.2 Funktionale Anforderungen

#### 2.2.1 Dateien

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

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

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

- `Write(size_t const bytes)`

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

## 2.2.2 Ordner

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

## 2.2.3 Verweise

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

## 2.3 Erzeugung der Elemente

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

## 2.4 Besucher (Visitor) Anforderungen

### 2.4.1 Visitor: Dump

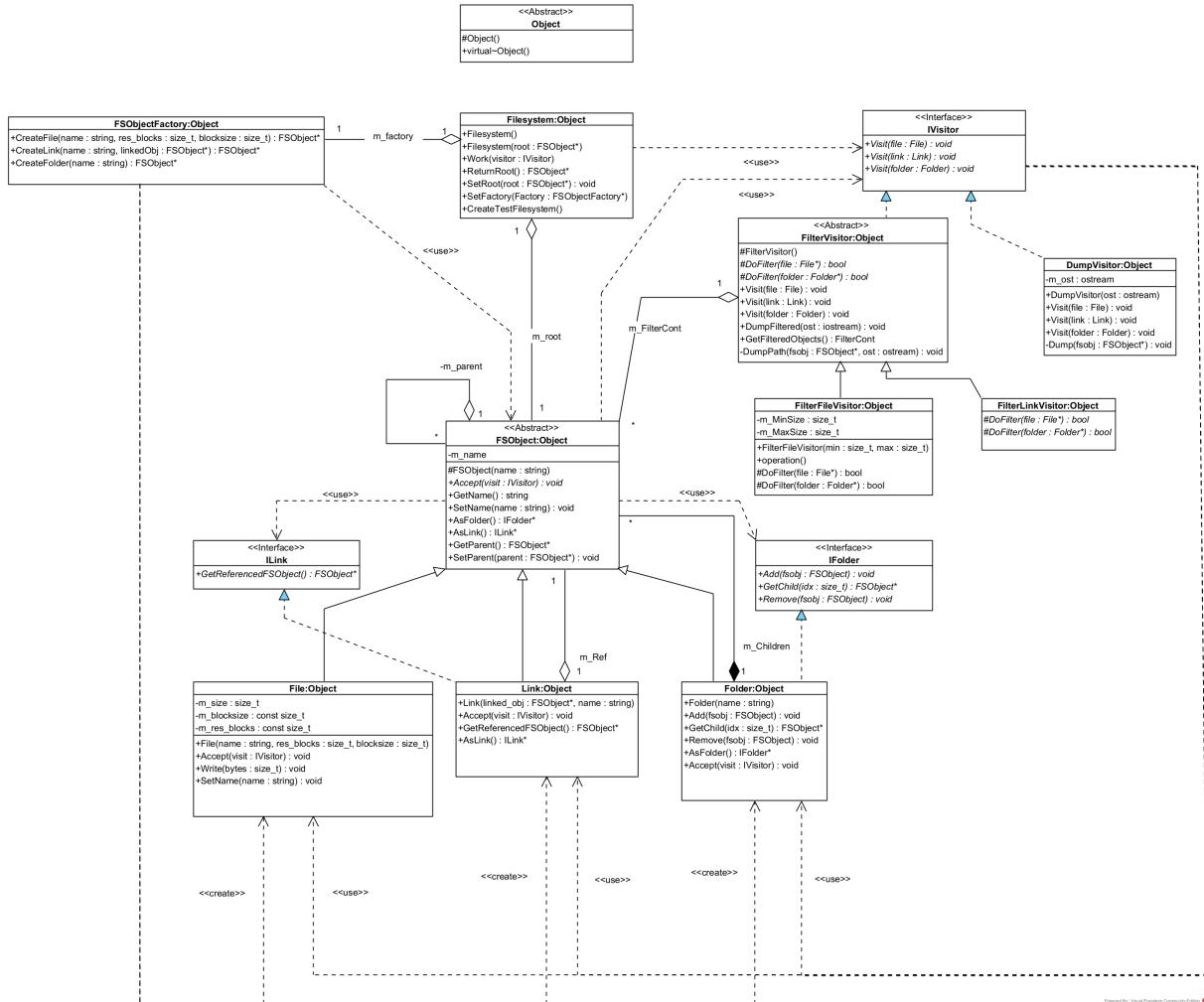
- Gibt die gesamte Dateisystemhierarchie aus.
- Ausgabe sowohl auf der Standardausgabe als auch in einer Datei möglich.
- Muss Dateien, Ordner und Verweise in strukturierter Form darstellen.

### 2.4.2 Visitor: FilterFiles

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

# 3 Systementwurf

## 3.1 Klassendiagramm



## 3.2 Designentscheidungen

Aus der Aufgabenstellung lassen sich folgenden Designpattern ableiten:

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

## 3.3 Composite Pattern

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

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

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

## 3.4 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 Design der Factory wurde auf das Interface zwischen Factory und Client verzichtet, da die Factory nur eine einzige Implementierung besitzt und keine weiteren Varianten geplant sind.

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

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

## 3.5 Visitor Pattern

Das Visitor Pattern wird verwendet, um verschiedene Operationen auf den Dateisystemelementen durchzuführen, ohne die Klassenhierarchie der Elemente zu verändern. Die Basisschnittstelle `IVisitor` definiert die Besuchsmethoden für jede Art von Dateisystemelement. Konkrete Besucherklassen wie `DumpVisitor` und `FilterFileVisitor` implementieren diese Methoden, um spezifische Funktionalitäten bereitzustellen.

## 3.6 Template Methode Pattern

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

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

## 4 Dokumentation der Komponenten (Klassen)

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

## 5 Testprotokollierung

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

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

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

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

```
158
159 **** TESTCASE START ****
160
161 ****
162
163 Test normal CTOR Link
164 [Test OK] Result: (Expected: MyFolder == Result: MyFolder)
165
166 Test normal CTOR Link
167 [Test OK] Result: (Expected: LinkToMyFolder == Result:
168   ↪ LinkToMyFolder)
169
170 Test normal CTOR Link - error buffer
171 [Test OK] Result: (Expected: true == Result: true)
172
173 Test Exception nullptr CTOR Link
174 [Test OK] Result: (Expected: ERROR Nullptr == Result: ERROR
175   ↪ Nullptr)
176
177 Test Exception empty string CTOR Link
178 [Test OK] Result: (Expected: ERROR String Empty == Result:
179   ↪ ERROR String Empty)
180
181 Test GetReferencedFSObject
182 [Test OK] Result: (Expected: file1.txt == Result: file1.txt)
183
184 Test chained links
185 [Test OK] Result: (Expected: Link1 == Result: Link1)
186
187 Test chained links - error buffer
188 [Test OK] Result: (Expected: true == Result: true)
189
190 Test link before destruction
191 [Test OK] Result: (Expected: true == Result: true)
192
193 Test link after object destruction
194 [Test OK] Result: (Expected: true == Result: true)
195
196 Test weak_ptr expiration - error buffer
197 [Test OK] Result: (Expected: true == Result: true)
198
```

```
199 Test AsLink() returns valid pointer
200 [Test OK] Result: (Expected: true == Result: true)
201
202 Test AsLink() reference matches
203 [Test OK] Result: (Expected: file.txt == Result: file.txt)
204
205 Test AsLink() - error buffer
206 [Test OK] Result: (Expected: true == Result: true)
207
208 Test Link SetName
209 [Test OK] Result: (Expected: NewName == Result: NewName)
210
211 Test SetName - error buffer
212 [Test OK] Result: (Expected: true == Result: true)
213
214 Test Link SetName empty string
215 [Test OK] Result: (Expected: ERROR String Empty == Result:
    ↪ ERROR String Empty)
216
217 Test Link Accept visitor - not empty
218 [Test OK] Result: (Expected: false == Result: false)
219
220 Test Link Accept - error buffer
221 [Test OK] Result: (Expected: true == Result: true)
222
223
224 ****
225
226
227 ****
228         TESTCASE START
229 ****
230
231 Test normal CTOR Folder
232 [Test OK] Result: (Expected: MyFolder == Result: MyFolder)
233
234 Get Child from folder
235 [Test OK] Result: (Expected: 0000023F51138FF0 == Result:
    ↪ 0000023F51138FF0)
236
237 Get next Child from folder
238 [Test OK] Result: (Expected: 0000023F511390B0 == Result:
    ↪ 0000023F511390B0)
239
```

```
240 Get Child for invalid index
241 [Test OK] Result: (Expected: 0000000000000000 == Result:
242     ↪ 0000000000000000)
243
244 Test Folder - error buffer
245 [Test OK] Result: (Expected: true == Result: true)
246
247 Test Remove Child from Folder
248 [Test OK] Result: (Expected: 0000023F511390B0 == Result:
249     ↪ 0000023F511390B0)
250
251 Test Folder - error buffer
252 [Test OK] Result: (Expected: true == Result: true)
253
254 Test Folder - add nullptr
255 [Test OK] Result: (Expected: ERROR Nullptr == Result: ERROR
256     ↪ Nullptr)
257
258 Test Folder - CTOR with empty string
259 [Test OK] Result: (Expected: ERROR String Empty == Result:
260     ↪ ERROR String Empty)
261
262 Test nested folders - root has sub1
263 [Test OK] Result: (Expected: 0000023F51138FF0 == Result:
264     ↪ 0000023F51138FF0)
265
266 Test nested folders - sub1 has sub2
267 [Test OK] Result: (Expected: 0000023F511390C0 == Result:
268     ↪ 0000023F511390C0)
269
270 Test nested folders - error buffer
271 [Test OK] Result: (Expected: true == Result: true)
272
273 Test parent pointer set on Add
274 [Test OK] Result: (Expected: parent == Result: parent)
275
276 Test parent pointer - error buffer
277 [Test OK] Result: (Expected: true == Result: true)
278
279 Test remove non-existent child
280 [Test OK] Result: (Expected: 0000023F51138FF0 == Result:
281     ↪ 0000023F51138FF0)
282
283 Test remove non-existent - error buffer
```

```
277 [Test OK] Result: (Expected: true == Result: true)
278
279 Test mixed children - file
280 [Test OK] Result: (Expected: 0000023F51138FF0 == Result:
281   ↪ 0000023F51138FF0)
282
283 Test mixed children - folder
284 [Test OK] Result: (Expected: 0000023F511390B8 == Result:
285   ↪ 0000023F511390B8)
286
287 Test mixed children - link
288 [Test OK] Result: (Expected: 0000023F51146730 == Result:
289   ↪ 0000023F51146730)
290
291 Test AsFolder() returns valid pointer
292 [Test OK] Result: (Expected: true == Result: true)
293
294 Test AsFolder() - error buffer
295 [Test OK] Result: (Expected: true == Result: true)
296
297 Test Accept visits children
298 [Test OK] Result: (Expected: true == Result: true)
299
300 Test Accept visitor - error buffer
301 [Test OK] Result: (Expected: true == Result: true)
302
303 Test Folder SetName
304 [Test OK] Result: (Expected: renamed == Result: renamed)
305
306 Test Folder SetName - error buffer
307 [Test OK] Result: (Expected: true == Result: true)
308
309
310 ****
311
312
313 ****
314         TESTCASE START
315 ****
316
317 Test normal CTOR File
```

```
318 [Test OK] Result: (Expected: file1.txt == Result: file1.txt)
319
320 Test normal CTOR File - size
321 [Test OK] Result: (Expected: 0 == Result: 0)
322
323 Test normal - write file size
324 [Test OK] Result: (Expected: 4096 == Result: 4096)
325
326 Test normal - error buffer empty
327 [Test OK] Result: (Expected: true == Result: true)
328
329 Test CTOR Empty string - error buffer empty
330 [Test OK] Result: (Expected: ERROR String Empty == Result:
    ↪ ERROR String Empty)
331
332 Test multiple writes
333 [Test OK] Result: (Expected: 6000 == Result: 6000)
334
335 Test multiple writes - error buffer
336 [Test OK] Result: (Expected: true == Result: true)
337
338 Test write to exact capacity
339 [Test OK] Result: (Expected: 5120 == Result: 5120)
340
341 Test exact capacity - error buffer
342 [Test OK] Result: (Expected: true == Result: true)
343
344 Test write exceeds capacity
345 [Test OK] Result: (Expected: Not enough space to write data ==
    ↪ Result: Not enough space to write data)
346
347 Test write zero bytes
348 [Test OK] Result: (Expected: 0 == Result: 0)
349
350 Test write zero - error buffer
351 [Test OK] Result: (Expected: true == Result: true)
352
353 Test multiple writes to capacity
354 [Test OK] Result: (Expected: 3000 == Result: 3000)
355
356 Test approach capacity - error buffer
357 [Test OK] Result: (Expected: true == Result: true)
358
359 Test write when full
```

```
360 [Test OK] Result: (Expected: Not enough space to write data ==  
361     ↳ Result: Not enough space to write data)  
362  
363 Test default blocksize  
364 [Test OK] Result: (Expected: 10000 == Result: 10000)  
365  
366 Test default blocksize - error buffer  
367 [Test OK] Result: (Expected: true == Result: true)  
368  
369 Test File Accept visitor  
370 [Test OK] Result: (Expected: true == Result: true)  
371  
372 Test File Accept - error buffer  
373 [Test OK] Result: (Expected: true == Result: true)  
374  
375 Test File SetName  
376 [Test OK] Result: (Expected: new.txt == Result: new.txt)  
377  
378 Test File SetName - error buffer  
379 [Test OK] Result: (Expected: true == Result: true)  
380  
381 Test File AsFolder returns nullptr  
382 [Test OK] Result: (Expected: true == Result: true)  
383  
384 Test File AsFolder - error buffer  
385 [Test OK] Result: (Expected: true == Result: true)  
386  
387 *****  
388  
389 TEST OK!!
```

## 6 Quellcode

## 6.1 Object.hpp

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

## 6.2 FSObjectFactory.hpp

```

1  /*****//** 
2  * \file FSObjectFactory.hpp
3  * \brief Simple Factory class to create filesystem objects
4  *
5  * \author Simon
6  * \date November 2025
7  *****/
8 #ifndef FS_OBJECT_FACTORY_HPP
9 #define FS_OBJECT_FACTORY_HPP
10
11 #include "Object.h"
12 #include "FSObject.hpp"
13 #include "Folder.hpp"
14 #include "File.hpp"
15 #include "Link.hpp"
16 #include <memory>
17
18
19 class FSObjectFactory : public Object
20 {
21 public:
22     using Uptr = std::unique_ptr<FSObjectFactory>;
23
24     inline static const std::string ERROR_NULLPTR = "ERROR_Nullptr";
25
26     /** \brief Create a File FSObject
27      * \param name Name of the file
28      * \param res_blocks Reserved blocks
29      * \param blocksize Block size (default 4096)
30      * \return Shared pointer to created File FSObject
31      */
32     FSObject::Sptr CreateFile(std::string_view name, const size_t res_blocks, const size_t blocksize
33                             = 4096) const;
34
35     /** \brief Create a Folder FSObject
36      * \param name Name of the folder
37      * \return Shared pointer to created Folder FSObject
38      */
39     FSObject::Spref CreateFolder(std::string_view name = "") const;
40
41     /** \brief Create a Link FSObject
42      * \param name Name of the link
43      * \param linkedObj Shared pointer to linked FSObject
44      * \return Shared pointer to created Link FSObject
45      */
46     FSObject::Spref CreateLink(std::string_view name, FSObject::Spref linkedObj) const;
47
48 private:
49 };
50 #endif

```

## 6.3 FSObjectFactory.cpp

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

## 6.4 Filesystem.hpp

```

1  /***** Filesystem.hpp ****/
2  * \file Filesystem.hpp
3  * \brief Filesystem class representing the root of a filesystem
4  *
5  * \author Simon
6  * \date November 2025
7  ****
8  #ifndef FILE_SYSTEM_HPP
9  #define FILE_SYSTEM_HPP
10
11 #include "FSObject.hpp"
12 #include "IVisitor.hpp"
13 #include "FSObjectFactory.hpp"
14
15 class FileSystem : public Object
16 {
17 public:
18
19     // Public Error Messages
20     inline static const std::string ERROR_NULLPTR = "ERROR_Nullptr";
21
22     FileSystem() = default;
23
24     /** \brief Construct a FileSystem with a root FSObject
25      * \param root Root FSObject shared pointer
26      */
27     FileSystem(FSObject::Spref 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::Spref ReturnRoot();
39
40     /** \brief Set the filesystem root
41      * \param root Shared pointer to new root
42      */
43     void SetRoot(FSObject::Spref root);
44
45     /** \brief Set the filesystem root
46      * \param root Shared pointer to new root
47      */
48     void SetFactory(FSObjectFactory::Upref 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 private:
57
58     FSObject::Spref m_Root;
59     FSObjectFactory::Upref m_Factory;
60 };
61 #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 FileSystem::FileSystem(FSObject::Sptr root)
14 {
15     if (root == nullptr) throw std::invalid_argument(ERROR_NULLPTR);
16
17     m_Root = move(root);
18 }
19 void FileSystem::Work(IVisitor& visitor)
20 {
21     if (m_Root == nullptr) throw std::invalid_argument(ERROR_NULLPTR);
22
23     m_Root->Accept(visitor);
24 }
25
26 FSObject::Sptr FileSystem::ReturnRoot()
27 {
28     return move(m_Root);
29 }
30
31 void FileSystem::SetRoot(FSObject::Sptr root)
32 {
33     if (root == nullptr) throw std::invalid_argument(ERROR_NULLPTR);
34
35     m_Root = move(root);
36 }
37
38 void FileSystem::SetFactory(FSObjectFactory::Uptr Factory)
39 {
40     if (Factory == nullptr) throw std::invalid_argument(ERROR_NULLPTR);
41
42     m_Factory = move(Factory);
43 }
44
45 void FileSystem::CreateTestFilesystem()
46 {
47     if (m_Factory == nullptr) throw std::invalid_argument(ERROR_NULLPTR);
48
49     FSObject::Sptr root_folder = m_Factory->CreateFolder("root");
50     IFolder::Sptr root_folder_ptr = root_folder->AsFolder();
51     FSObject::Sptr sub_folder = m_Factory->CreateFolder("sub");
52     IFolder::Sptr sub_folder_ptr = sub_folder->AsFolder();
53     FSObject::Sptr sub_sub_folder = m_Factory->CreateFolder("sub");
54     IFolder::Sptr sub_sub_folder_ptr = sub_sub_folder->AsFolder();
55
56     sub_folder->SetName("sub_folder");
57     sub_sub_folder->SetName("sub_sub_folder");
58
59     root_folder->SetName("root");
60     root_folder_ptr->Add(m_Factory->CreateFile("file1.txt", 2048));
61     root_folder_ptr->Add(m_Factory->CreateFile("file2.txt", 2048));
62     root_folder_ptr->Add(m_Factory->CreateFile("file3.txt", 2048));
63     root_folder_ptr->Add(m_Factory->CreateFile("file4.txt", 2048));
64     root_folder_ptr->Add(sub_folder);
65     sub_folder_ptr->Add(m_Factory->CreateFile("file5.txt", 8192));
66     sub_folder_ptr->Add(m_Factory->CreateFile("file6.txt", 32768));
67     sub_folder_ptr->Add(sub_sub_folder);
68     sub_sub_folder_ptr->Add(m_Factory->CreateFile("file7.txt", 12288));
69     sub_sub_folder_ptr->Add(m_Factory->CreateLink("LinkToRoot", root_folder));
70
71     m_Root = move(root_folder);
72 }
```

## 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 Try to "cast" this FSObject to a folder  

37      * \return Shared pointer to IFolder or nullptr  

38      */  

39     virtual IFolder::Sptr AsFolder();  

40  

41     /** \brief Try to "cast" this FSObject to a folder  

42      * \return Shared pointer to IFolder or nullptr  

43      */  

44     virtual std::shared_ptr<const IFolder> AsFolder() const;  

45  

46     /** \brief Try to cast this FSObject to a link  

47      * \return Shared pointer to ILink or nullptr  

48      */  

49     virtual std::shared_ptr<const ILink> AsLink() const;  

50  

51     /** \brief Get the name of the object  

52      * \return Name as std::string_view  

53      */  

54     std::string_view GetName() const;  

55  

56     /** \brief Set the name of the object  

57      * \param name New name  

58      */  

59     void SetName(std::string_view name);  

60  

61     /** \brief Get parent as weak pointer  

62      * \return Weak pointer to parent  

63      */  

64     FSObj_Wptr GetParent() const;  

65  

66     /** \brief Set parent of this FSObject  

67      * \param parent Shared pointer to parent FSObject  

68      */  

69     void SetParent(Sptr parent);  

70  

71 protected:  

72

```

```
73     /** \brief Construct an FSObject with optional name
74      * \param name Name of the FSObject
75      */
76     FSObject(std::string_view name = "");
77
78
79 private:
80     std::string m_Name;
81     FSOBJ_WPTR m_Parent;
82 }
83
84 #endif
```

## 6.7 FSObject.cpp

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

## 6.8 File.hpp

```

1  ****  

2  * \file File.hpp  

3  * \brief File class representing a file in the filesystem  

4  *  

5  * \author Simon  

6  * \date November 2025  

7  ****  

8  #ifndef FILE_HPP  

9  #define FILE_HPP  

10  

11 #include "FSObject.hpp"  

12  

13 class File : public FSObject, public std::enable_shared_from_this<File>  

14 {  

15 public:  

16     // Public Error Messages  

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

18     inline static const std::string ERR_OUT_OF_SPACE = "Not_enough_space_to_write_data";  

19  

20     // Smart pointer types  

21     using Uptr = std::unique_ptr<File>;  

22     using Sptr = std::shared_ptr<File>;  

23     using Wptr = std::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 private:  

53     size_t m_size;  

54     const size_t m_blocksize;  

55     const size_t m_res_blocks;  

56 };
57 #endif

```

## 6.9 File.cpp

## 6.10 IFolder.hpp

```

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

2  * \file IFolder.hpp  

3  * \brief Interface for folder-like FSObjects  

4  *  

5  * \author Simon  

6  * \date November 2025  

7  *****/  

8  #ifndef IFOLDER_HPP  

9  #define IFOLDER_HPP  

10 #include <memory>  

11  

12 // fwd declaration  

13 class FSObject;  

14  

15 // Type aliases  

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

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

18  

19 class IFolder  

20 {  

21 public:  

22  

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

24  

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

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

27      */  

28     virtual void Add(FSObj_Sptr fsobj) =0;  

29  

30     /** \brief Get a child by index  

31      * \param idx Index of the child  

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

33      */  

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

35  

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

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

38      */  

39     virtual void Remove(FSObj_Sptr fsobj) =0;  

40  

41     /** \brief Virtual destructor */  

42     virtual ~IFolder() = default;  

43  

44 private:  

45 };  

46  

47 #endiff

```

## 6.11 Folder.hpp

```

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

```

## 6.12 Folder.cpp

## 6.13 ILink.hpp

## 6.14 Link.hpp

```

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

```

## 6.15 Link.cpp

## 6.16 IVisitor.hpp

```

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

```

## 6.17 FilterVisitor.hpp

```
73     * \param ost Output stream
74     */
75     void DumpPath(const std::weak_ptr<const FSObject> & fsobj, std::ostream& ost) const;
76
77     TContFSobj m_FilterCont;
78 };
79
80 #endif
```

## 6.18 FilterVisitor.cpp

```

1  /**************************************************************************/ 
2  * \file FilterVisitor.cpp
3  * \brief Visitor that filters filesystem objects based on criteria defines in derived classes
4  *
5  * \author Simon
6  * \date November 2025
7  /**************************************************************************/
8 #include "FilterVisitor.hpp"
9 #include "Folder.hpp"
10 #include "File.hpp"
11 #include "Link.hpp"
12
13 #include <vector>
14 #include <iostream>
15 #include <cassert>
16 #include <stdexcept>
17
18 void FilterVisitor::DumpPath(const std::weak_ptr<const FSObject> & fsobj, std::ostream& ost) const
19 {
20     // end recursion on expired weak pointer
21     if (fsobj.expired()) return;
22
23     const auto obj = fsobj.lock();
24     if (!obj) return; // defensive: lock could fail
25
26     // first dump parent path
27     DumpPath(obj->GetParent(), ost);
28
29     if (!ost.good()) throw std::invalid_argument(FilterVisitor::ERROR_BAD_OSTREAM);
30
31     ost << "\\\" << obj->GetName();
32
33     const std::shared_ptr<const ILink> link_ptr = obj->AsLink();
34
35     if (link_ptr) {
36         const FSObject::Sptr linked_obj = link_ptr->GetReferencedFSObject();
37         if (linked_obj) {
38             ost << "\u2192" << linked_obj->GetName();
39         }
40         else {
41             ost << "\u2192" << "linked Object Expired!";
42         }
43     }
44 }
45
46 /**
47 * \brief Default visit for folder (no-op)
48 void FilterVisitor::Visit(const std::shared_ptr<const Folder> folder)
49 {
50     if (folder == nullptr) throw std::invalid_argument(ERROR_NULLPTR);
51 }
52
53 /**
54 * \brief Visit a file and if it matches add to filtered container
55 void FilterVisitor::Visit(const std::shared_ptr<const File> file)
56 {
57     if (file == nullptr) throw std::invalid_argument(ERROR_NULLPTR);
58
59     // if file matches filter add to container
60     if (DoFilter(file))
61     {
62         m_FilterCont.emplace_back(file);
63     }
64 }
65 /**
66 * \brief Visit a link and if it matches add to filtered container
67 void FilterVisitor::Visit(const std::shared_ptr<const Link> link)
68 {
69     if (link == nullptr) throw std::invalid_argument(ERROR_NULLPTR);
70
71     // if link matches filter add to container
72     if (DoFilter(link))
73     {

```

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

## 6.19 FilterFileVisitor.hpp

```

1  ****  

2  * \file FilterFileVisitor.hpp  

3  * \brief Visitor that filters files by size range  

4  *  

5  * \author Simon  

6  * \date November 2025  

7  ****  

8  #ifndef FILTER_FILE_VISITOR_HPP  

9  #define FILTER_FILE_VISITOR_HPP  

10  

11 #include "FilterVisitor.hpp"  

12  

13 class FilterFileVisitor : public FilterVisitor  

14 {  

15 public:  

16     // Public Error Messages  

17     inline static const std::string ERROR_INVALID_SIZE_RANGE = "Invalid_size_range:_minimum_size_  

18     must_be_less_than_maximum_size";  

19  

20     /** \brief Construct file filter with size range [min,max]  

21      * \param min Minimum size (inclusive) call by value for built-in type -> is faster than by  

22      * reference  

23      * \param max Maximum size (inclusive) call by value for built-in type -> is faster than by  

24      * reference  

25      */  

26     FilterFileVisitor(const size_t min, const size_t max);  

27  

28 protected:  

29  

30     /** \brief Do filter check for files  

31      * \param file File to check  

32      * \return true if file size is within range  

33      */  

34     virtual bool DoFilter(const std::shared_ptr<const File>& file) const override;  

35  

36     /** \brief Links are not accepted by this filter  

37      * \param link Link to check  

38      * \return false always  

39      */  

40     virtual bool DoFilter(const std::shared_ptr<const Link>& link) const override;  

41  

42 private:  

43     // cannot be const because there are checks in the constructor  

44     size_t m_MinSize;  

45     size_t m_MaxSize;  

46 };
47  

48 #endif

```

## 6.20 FilterFileVisitor.cpp

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

## 6.21 FilterLinkVisitor.hpp

```
1 //*****\n2 * \file FilterLinkVisitor.hpp\n3 * \brief Visitor that filters links in the filesystem\n4 *\n5 * \author Simon\n6 * \date December 2025\n7 *****\n8 #ifndef FILTER_LINK_VISITOR_HPP\n9 #define FILTER_LINK_VISITOR_HPP\n10\n11 #include "FilterVisitor.hpp"\n12\n13 class FilterLinkVisitor : public FilterVisitor\n14 {\n15 public:\n16\n17 protected:\n18\n19     /** \brief Links are accepted by this filter\n20      * \param file File to check\n21      * \return false always\n22      */\n23     virtual bool DoFilter(const std::shared_ptr<const File>& file) const override;\n24\n25     /** \brief Links are accepted by this filter\n26      * \param link Link to check\n27      * \return true if link is present\n28      */\n29     virtual bool DoFilter(const std::shared_ptr<const Link>& link) const override;\n30\n31 private:\n32 };\n33\n34 #endif
```

## 6.22 FilterLinkVisitor.cpp

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

## 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 private:
44     /** \brief Dump a single FSObject path to the output stream
45      * \param fsobj Shared pointer to object
46      */
47     void Dump(const std::shared_ptr<const FSObject> fsobj);
48
49     // Output stream reference
50     std::ostream & m_ost;
51 };
52
53 #endif

```

## 6.24 DumpVisitor.cpp

## 6.25 main.cpp

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

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

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

```
222     ost << TestStart;
223
224     bool TestOK = true;
225     string error_msg;
226
227
228     try {
229         FSObjectFactory factory;
230         FSObject::Sptr root_folder = factory.CreateFolder("root");
231         FSObject::Sptr sub_folder = factory.CreateFolder("sub_folder");
232         FSObject::Sptr sub_sub_folder = factory.CreateFolder("sub_sub_folder");
233         File::Sptr file = make_shared<File>("file1.txt", 2048);
234         Link::Sptr link = make_shared<Link>(file, "LinkToFile1");
235         sub_sub_folder->AsFolder()->Add(file);
236         sub_sub_folder->AsFolder()->Add(link);
237         sub_folder->AsFolder()->Add(sub_sub_folder);
238         root_folder->AsFolder()->Add(sub_folder);
239
240         FilterLinkVisitor link_filter;
241
242         root_folder->Accept(link_filter);
243
244         TestOK = TestOK && check_dump(ost, "FilterLinkVisitor_Test_filtered_amount",
245                                         static_cast<size_t>(1), link_filter.GetFilteredObjects().size());
246         TestOK = TestOK && check_dump(ost, "FilterLinkVisitor_Test_filtered_obj",
247                                         link->GetReferencedFSObject()->GetName(), link_filter.GetFilteredObjects().cbegin()->lock
248                                         ()->AsLink()->GetReferencedFSObject()->GetName());
249
250         stringstream result;
251         stringstream expected;
252
253         link_filter.DumpFiltered(result);
254
255         expected << "\\root\\sub_folder\\sub_sub_folder\\LinkToFile1->_file1.txt" << std::endl
256         ;
257
258         TestOK = TestOK && check_dump(ost, "Filter_Link_Visitor_Test_Dump",
259                                         expected.str(),
260                                         result.str());
261
262     }
263     catch (const string& err) {
264         error_msg = err;
265     }
266     catch (bad_alloc const& error) {
267         error_msg = error.what();
268     }
269     catch (const exception& err) {
270         error_msg = err.what();
271     }
272     catch (...) {
273         error_msg = "Unhandelt_Exception";
274     }
275
276     TestOK = TestOK && check_dump(ost, "Test_for_Exception_in_TestCase", true, error_msg.empty());
277     error_msg.clear();
278
279     try {
280
281         FilterLinkVisitor link_filter();
282
283         stringstream result;
284         result.setstate(ios::badbit);
285
286         link_filter.DumpFiltered(result);
287     }
288     catch (const string& err) {
289         error_msg = err;
290     }
291     catch (bad_alloc const& error) {
292         error_msg = error.what();
293     }
294     catch (const exception& err) {
295         error_msg = err.what();
296     }
```

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

```
362     error_msg = error.what();
363 }
364 catch (const exception& err) {
365     error_msg = err.what();
366 }
367 catch (...) {
368     error_msg = "UnhandeltedException";
369 }
370 TestOK = TestOK && check_dump(ost, "Test_for_Exception_in_Testcase", true, error_msg.empty());
371 error_msg.clear();
372
373 try {
374     FilterFileVisitor file_filter(1,2);
375
376     stringstream result;
377     result.setstate(ios::badbit);
378
379     file_filter.DumpFiltered(result);
380 }
381 catch (const string& err) {
382     error_msg = err;
383 }
384 catch (bad_alloc const& error) {
385     error_msg = error.what();
386 }
387 catch (const exception& err) {
388     error_msg = err.what();
389 }
390 catch (...) {
391     error_msg = "UnhandeltedException";
392 }
393
394 TestOK = TestOK && check_dump(ost, "Test_for_Exception_in_Dump_with_bad_Ostream", error_msg,
395     FilterLinkVisitor::ERROR_BAD_OSTREAM);
396 error_msg.clear();
397
398 try {
399     FilterFileVisitor file_filter{ 2,1 }; // <= should throw invalid size range
400 }
401 catch (const string& err) {
402     error_msg = err;
403 }
404 catch (bad_alloc const& error) {
405     error_msg = error.what();
406 }
407 catch (const exception& err) {
408     error_msg = err.what();
409 }
410 catch (...) {
411     error_msg = "UnhandeltedException";
412 }
413
414 TestOK = TestOK && check_dump(ost, "Test_for_Exception_in_Filter_File_VisiterCTOR", error_msg,
415     FilterFileVisitor::ERROR_INVALID_SIZE_RANGE);
416 error_msg.clear();
417
418
419 ost << TestEnd;
420
421 return TestOK;
422 }
423
424 bool TestVisitor(ostream& ost, IVisitor& visit)
425 {
426     assert(ost.good());
427
428     ost << TestStart;
429
430     bool TestOK = true;
431     string error_msg;
432
433
434 }
```

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

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

```
578 }
579
580 bool TestLink(ostream& ost)
581 {
582     assert(ost.good());
583
584     ost << TestStart;
585
586     bool TestOK = true;
587     string error_msg;
588
589     // test normal operation
590     try
591     {
592         std::string_view folder_name = "MyFolder";
593         std::string_view link_name = "LinkToMyFolder";
594         Folder::Sptr folder = make_shared<Folder>(folder_name);
595         Link::Sptr link = make_shared<Link>(folder, link_name);
596
597         TestOK = TestOK && check_dump(ost, "Test_normal_CTOR_Link", folder_name, link->
598                                         GetReferencedFSObject()->GetName());
599         TestOK = TestOK && check_dump(ost, "Test_normal_CTOR_Link", link_name, link->GetName());
600
601     }
602     catch (const string& err) {
603         error_msg = err;
604     }
605     catch (bad_alloc const& error) {
606         error_msg = error.what();
607     }
608     catch (const exception& err) {
609         error_msg = err.what();
610     }
611     catch (...) {
612         error_msg = "Unhandelt_Exception";
613     }
614
615     TestOK = TestOK && check_dump(ost, "Test_normal_CTOR_Link_-_error_buffer", true, error_msg.empty())
616         ;
617     error_msg.clear();
618
619     // test link to nullptr
620     try
621     {
622         Link::Sptr link = make_shared<Link>(nullptr, "LinkToNothing");
623     }
624     catch (const string& err) {
625         error_msg = err;
626     }
627     catch (bad_alloc const& error) {
628         error_msg = error.what();
629     }
630     catch (const exception& err) {
631         error_msg = err.what();
632     }
633     catch (...) {
634         error_msg = "Unhandelt_Exception";
635     }
636
637     TestOK = TestOK && check_dump(ost, "Test_Exception_nullptr_CTOR_Link", Link::ERROR_NULLPTR,
638                                     error_msg);
639     error_msg.clear();
640
641     // test Link with empty string
642     try
643     {
644         File::Sptr file = make_shared<File>("file1.txt", 2048);
645         Link::Sptr link = make_shared<Link>(file, "");
646     }
647     catch (const string& err) {
648         error_msg = err;
649     }
649     catch (bad_alloc const& error) {
650         error_msg = error.what();
651     }
```

```
650     catch (const exception& err) {
651         error_msg = err.what();
652     }
653     catch (...) {
654         error_msg = "UnhandeltedException";
655     }
656 }
657 TestOK = TestOK && check_dump(ost, "Test_Exception_empty_stringCTOR_Link", Link::
658     ERROR_STRING_EMPTY, error_msg);
659 error_msg.clear();
660
661 // test Link GetReferencedFSObject
662 try
663 {
664     File::Sptr file = make_shared<File>("file1.txt", 2048);
665     Link::Sptr link = make_shared<Link>(file, file->GetName());
666
667     FSObj_Sptr ref = link->GetReferencedFSObject(); // <= should be File not Folder
668
669     TestOK = TestOK && check_dump(ost, "Test_GetReferencedFSObject", file->GetName(), ref->GetName()
670         ());
671     catch (const string& err) {
672         error_msg = err;
673     }
674     catch (bad_alloc const& error) {
675         error_msg = error.what();
676     }
677     catch (const exception& err) {
678         error_msg = err.what();
679     }
680     catch (...) {
681         error_msg = "UnhandeltedException";
682     }
683
684 TestOK = TestOK && check_dump(ost, "Empty_error_buffer", true, error_msg.empty());
685 error_msg.clear();
686
687 // Link to a Link (chained links)
688 try
689 {
690     File::Sptr file = make_shared<File>("original.txt", 2048);
691     Link::Sptr link1 = make_shared<Link>(file, "Link1");
692     Link::Sptr link2 = make_shared<Link>(link1, "Link2");
693
694     TestOK = TestOK && check_dump(ost, "Test_chained_links",
695         link1->GetName(), link2->GetReferencedFSObject()->GetName());
696 }
697 catch (const exception& err) {
698     error_msg = err.what();
699 }
700 TestOK = TestOK && check_dump(ost, "Test_chained_links_-_error_buffer", true, error_msg.empty());
701 error_msg.clear();
702
703 //Link when referenced object is destroyed (weak_ptr expiration)
704 try
705 {
706     Link::Sptr link;
707     {
708         File::Sptr file = make_shared<File>("temp.txt", 2048);
709         link = make_shared<Link>(file, "LinkToTemp");
710         TestOK = TestOK && check_dump(ost, "Test_link_before_destruction",
711             true, link->GetReferencedFSObject() != nullptr);
712     } // file goes out of scope here
713
714     FSObj_Sptr expired_ref = link->GetReferencedFSObject();
715     TestOK = TestOK && check_dump(ost, "Test_link_after_object_destruction",
716         true, expired_ref == nullptr);
717 }
718 catch (const exception& err) {
719     error_msg = err.what();
720 }
721 TestOK = TestOK && check_dump(ost, "Test_weak_ptr_expiration_-_error_buffer", true, error_msg.empty()
722     ());
```

```
722     error_msg.clear();
723
724     //AsLink() method returns valid pointer
725     try
726     {
727         File::Sptr file = make_shared<File>("file.txt", 2048);
728         Link::Spref link = make_shared<Link>(file, "TestLink");
729
730         std::shared_ptr<const ILink> ilink = link->AsLink();
731         TestOK = TestOK && check_dump(ost, "Test_AsLink()_returns_valid_pointer",
732             true, ilink != nullptr);
733         TestOK = TestOK && check_dump(ost, "Test_AsLink()_reference_matches",
734             file->GetName(), ilink->GetReferencedFSObject()->GetName());
735     }
736     catch (const exception& err)
737     {
738         error_msg = err.what();
739     }
740     TestOK = TestOK && check_dump(ost, "Test_AsLink()_error_buffer", true, error_msg.empty());
741     error_msg.clear();
742
743     //Link SetName functionality
744     try
745     {
746         File::Spref file = make_shared<File>("file.txt", 2048);
747         Link::Spref link = make_shared<Link>(file, "OriginalName");
748
749         link->SetName("NewName");
750         TestOK = TestOK && check_dump(ost, "Test_Link_SetName",
751             string_view("NewName"), link->GetName());
752     }
753     catch (const exception& err)
754     {
755         error_msg = err.what();
756     }
757     TestOK = TestOK && check_dump(ost, "Test_SetName_error_buffer", true, error_msg.empty());
758     error_msg.clear();
759
760     //Link SetName with empty string (should throw)
761     try
762     {
763         File::Spref file = make_shared<File>("file.txt", 2048);
764         Link::Spref link = make_shared<Link>(file, "OriginalName");
765         link->SetName(""); // should throw
766     }
767     catch (const exception& err)
768     {
769         error_msg = err.what();
770     }
771     TestOK = TestOK && check_dump(ost, "Test_Link_SetName_empty_string",
772         FSObject::ERROR_STRING_EMPTY, error_msg);
773     error_msg.clear();
774
775     // Link Accept visitor
776     try
777     {
778         File::Spref file = make_shared<File>("file.txt", 2048);
779         Link::Spref link = make_shared<Link>(file, "TestLink");
780         stringstream result;
781         DumpVisitor visitor(result);
782
783         link->Accept(visitor);
784         TestOK = TestOK && check_dump(ost, "Test_Link_Accept_visitor_not_empty",
785             false, result.str().empty());
786     }
787     catch (const exception& err)
788     {
789         error_msg = err.what();
790     }
791     TestOK = TestOK && check_dump(ost, "Test_Link_Accept_error_buffer", true, error_msg.empty());
792     error_msg.clear();
793
794     ost << TestEnd;
795     return TestOK;
796 }
797 bool TestFolder(ostream& ost)
798 {
799     assert(ost.good());
800 }
```

```
797     ost << TestStart;
798
799     bool TestOK = true;
800     string error_msg;
801
802     // test folder as intended
803     try
804     {
805         string_view folder_name = "MyFolder";
806         Folder::Sptr folder = make_shared<Folder>(folder_name);
807         TestOK = TestOK && check_dump(ost, "Test_normalCTOR_Folder", folder_name, folder->GetName());
808
809         File::Sptr file1 = make_shared<File>("file1.txt", 2048);
810         File::Sptr file2 = make_shared<File>("file2.txt", 4096);
811
812         folder->Add(file1);
813         folder->Add(file2);
814
815         FSObject::Sptr err_file = folder->GetChild(2); // <= should be nullptr
816         FSObject::Sptr shared_null = nullptr;
817
818         TestOK = TestOK && check_dump(ost, "Get_Child_from_folder", static_pointer_cast<FSObject>(file1),
819                                       folder->GetChild(0));
820         TestOK = TestOK && check_dump(ost, "Get_next_Child_from_folder", static_pointer_cast<FSObject>(
821                                       file2), folder->GetChild(1));
822         TestOK = TestOK && check_dump(ost, "Get_Child_for_invalid_index", err_file, shared_null);
823     }
824     catch (const string& err)
825     {
826         error_msg = err;
827     }
828     catch (bad_alloc const& error)
829     {
830         error_msg = error.what();
831     }
832     catch (...) {
833         error_msg = "UnhandeltException";
834     }
835
836     TestOK = TestOK && check_dump(ost, "Test_Folder_error_buffer", error_msg.empty(), true);
837     error_msg.clear();
838
839     // test remove child
840     try
841     {
842         Folder::Sptr folder = make_shared<Folder>("MyFolder");
843         File::Sptr file1 = make_shared<File>("file1.txt", 2048);
844         File::Sptr file2 = make_shared<File>("file2.txt", 4096);
845         folder->Add(file1);
846         folder->Add(file2);
847         folder->Remove(file1);
848         TestOK = TestOK && check_dump(ost, "Test_Remove_Child_from_Folder", static_pointer_cast<
849                                       FSObject>(file2), folder->GetChild(0));
850     }
851     catch (const string& err)
852     {
853         error_msg = err;
854     }
855     catch (bad_alloc const& error)
856     {
857         error_msg = error.what();
858     }
859     catch (...) {
860         error_msg = "UnhandeltException";
861     }
862
863     TestOK = TestOK && check_dump(ost, "Test_Folder_error_buffer", error_msg.empty(), true);
864     error_msg.clear();
865
866     // test add nullptr
867     try
868     {
869         Folder::Sptr folder = make_shared<Folder>("MyFolder");
```

```
869     FSOObject::Sptr null_ptr = nullptr;
870     folder->Add(null_ptr); // <= should throw Exception
871 }
872 catch (const string& err) {
873     error_msg = err;
874 }
875 catch (bad_alloc const& error) {
876     error_msg = error.what();
877 }
878 catch (const exception& err) {
879     error_msg = err.what();
880 }
881 catch (...) {
882     error_msg = "UnhandeltedException";
883 }
884
885 TestOK = TestOK && check_dump(ost, "Test_Folder_-_add_nullptr", Folder::ERROR_NULLPTR, error_msg);
886 error_msg.clear();
887
888 // test Folder with empty string
889 try
890 {
891     Folder::Sptr folder = make_shared<Folder>("");
892 }
893 catch (const string& err) {
894     error_msg = err;
895 }
896 catch (bad_alloc const& error) {
897     error_msg = error.what();
898 }
899 catch (const exception& err) {
900     error_msg = err.what();
901 }
902 catch (...) {
903     error_msg = "UnhandeltedException";
904 }
905
906 TestOK = TestOK && check_dump(ost, "Test_Folder_-_CTOR_with_empty_string", FSOObject::
907     ERROR_STRING_EMPTY, error_msg);
908 error_msg.clear();
909
910 //Nested folder structure
911 try
912 {
913     Folder::Sptr root = make_shared<Folder>("root");
914     Folder::Sptr sub1 = make_shared<Folder>("sub1");
915     Folder::Sptr sub2 = make_shared<Folder>("sub2");
916
917     root->Add(sub1);
918     sub1->Add(sub2);
919
920     TestOK = TestOK && check_dump(ost, "Test_nested_folders_-_root_has_sub1",
921         sub1, static_pointer_cast<Folder>(root->GetChild(0)));
922     TestOK = TestOK && check_dump(ost, "Test_nested_folders_-_sub1_has_sub2",
923         sub2, static_pointer_cast<Folder>(sub1->GetChild(0)));
924 }
925 catch (const exception& err) {
926     error_msg = err.what();
927 }
928 TestOK = TestOK && check_dump(ost, "Test_nested_folders_-_error_buffer", true, error_msg.empty());
929 error_msg.clear();
930
931 //Parent pointer is set correctly when adding child
932 try
933 {
934     Folder::Sptr parent = make_shared<Folder>("parent");
935     File::Sptr child = make_shared<File>("child.txt", 2048);
936
937     parent->Add(child);
938     FSOObj_Wptr parent_wptr = child->GetParent();
939     FSOObj_Sptr parent_sptra = parent_wptr.lock();
940
941     TestOK = TestOK && check_dump(ost, "Test_parent_pointer_set_on_Add",
942         parent->GetName(), parent_sptra->GetName());
943 }
```

```
943     catch (const exception& err) {
944         error_msg = err.what();
945     }
946     TestOK = TestOK && check_dump(ost, "Test_parent_pointer_error_buffer", true, error_msg.empty());
947     error_msg.clear();
948
949     //Remove non-existent child (should not crash)
950     try
951     {
952         Folder::Sptr folder = make_shared<Folder>("folder");
953         File::Sptr file1 = make_shared<File>("file1.txt", 2048);
954         File::Sptr file2 = make_shared<File>("file2.txt", 2048);
955
956         folder->Add(file1);
957         folder->Remove(file2); // file2 was never added
958
959         TestOK = TestOK && check_dump(ost, "Test_remove_non-existent_child",
960             static_pointer_cast<FSObject>(file1), folder->GetChild(0));
961     }
962     catch (const exception& err) {
963         error_msg = err.what();
964     }
965     TestOK = TestOK && check_dump(ost, "Test_remove_non-existent_error_buffer", true, error_msg.empty()
966         ());
967     error_msg.clear();
968
969     //Multiple children of different types
970     try
971     {
972         Folder::Sptr folder = make_shared<Folder>("mixed");
973         File::Sptr file = make_shared<File>("file.txt", 2048);
974         Folder::Sptr subfolder = make_shared<Folder>("subfolder");
975         Link::Sptr link = make_shared<Link>(file, "link");
976
977         folder->Add(file);
978         folder->Add(subfolder);
979         folder->Add(link);
980
981         TestOK = TestOK && check_dump(ost, "Test_mixed_children_file",
982             static_pointer_cast<FSObject>(file), folder->GetChild(0));
983         TestOK = TestOK && check_dump(ost, "Test_mixed_children_folder",
984             static_pointer_cast<FSObject>(subfolder), folder->GetChild(1));
985         TestOK = TestOK && check_dump(ost, "Test_mixed_children_link",
986             static_pointer_cast<FSObject>(link), folder->GetChild(2));
987     }
988     catch (const exception& err) {
989         error_msg = err.what();
990     }
991     TestOK = TestOK && check_dump(ost, "Test_mixed_children_error_buffer", true, error_msg.empty());
992     error_msg.clear();
993
994     //AsFolder() returns valid pointer
995     try
996     {
997         Folder::Sptr folder = make_shared<Folder>("test");
998         IFolder::Sptr ifolder = folder->AsFolder();
999
1000        TestOK = TestOK && check_dump(ost, "Test_AsFolder_returns_valid_pointer",
1001            true, ifolder != nullptr);
1002    }
1003    catch (const exception& err) {
1004        error_msg = err.what();
1005    }
1006    TestOK = TestOK && check_dump(ost, "Test_AsFolder_error_buffer", true, error_msg.empty());
1007    error_msg.clear();
1008
1009    //Accept visitor with children
1010    try
1011    {
1012        Folder::Sptr folder = make_shared<Folder>("root");
1013        File::Sptr file = make_shared<File>("file.txt", 2048);
1014        folder->Add(file);
1015
1016        stringstream result;
1017        DumpVisitor visitor(result);
```

```
1017     folder->Accept(visitor);
1018
1019     // Should visit both folder and file
1020     TestOK = TestOK && check_dump(ost, "Test_Accept_visits_children",
1021         true, result.str().find("root") != string::npos &&
1022         result.str().find("file.txt") != string::npos);
1023 }
1024 catch (const exception& err) {
1025     error_msg = err.what();
1026 }
1027 TestOK = TestOK && check_dump(ost, "Test_Accept_visitor_error_buffer", true, error_msg.empty());
1028 error_msg.clear();
1029
1030 //SetName on folder
1031 try
1032 {
1033     Folder::Sptr folder = make_shared<Folder>("original");
1034     folder->SetName("renamed");
1035
1036     TestOK = TestOK && check_dump(ost, "Test_Folder_SetName",
1037         string_view("renamed"), folder->GetName());
1038 }
1039 catch (const exception& err) {
1040     error_msg = err.what();
1041 }
1042 TestOK = TestOK && check_dump(ost, "Test_Folder_SetName_error_buffer", true, error_msg.empty());
1043 error_msg.clear();
1044
1045 ost << TestEnd;
1046 return TestOK;
1047 }
1048 bool TestFile(ostream& ost)
1049 {
1050     assert(ost.good());
1051
1052     ost << TestStart;
1053
1054     bool TestOK = true;
1055     string error_msg;
1056
1057     // File as intended
1058     try
1059     {
1060         string_view file_name = "file1.txt";
1061         size_t block_size = 2048;
1062         size_t res_blocks = 20;
1063         File::Sptr file = make_shared<File>(file_name, res_blocks, block_size);
1064
1065         TestOK = TestOK && check_dump(ost, "Test_normal_CTOR_File", file_name, file->GetName());
1066         TestOK = TestOK && check_dump(
1067             ost, "Test_normal_CTOR_File_size",
1068             static_cast<size_t>(0), file->GetSize());
1069
1070         // Write to file
1071         size_t write_size = 4096;
1072         file->Write(write_size);
1073         TestOK = TestOK && check_dump(ost, "Test_normal_write_file_size", write_size, file->GetSize()
1074             );
1075     }
1076     catch (const string& err) {
1077         error_msg = err;
1078     }
1079     catch (bad_alloc const& error) {
1080         error_msg = error.what();
1081     }
1082     catch (const exception& err) {
1083         error_msg = err.what();
1084     }
1085     catch (...) {
1086         error_msg = "Unhandelt_Exception";
1087     }
1088
1089     TestOK = TestOK && check_dump(ost, "Test_normal_error_buffer_empty", error_msg.empty(), true);
1090     error_msg.clear();
1091 }
```

```
1091 // File with empty string
1092 try
1093 {
1094     File::Sptr file = make_shared<File>("", 20, 2048);
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_CTOR_Empty_string_-_error_buffer_empty", error_msg, File::
1110     ERROR_STRING_EMPTY);
1111 error_msg.clear();
1112
1113 // Write multiple times
1114 try
1115 {
1116     File::Sptr file = make_shared<File>("multi.txt", 10, 2048);
1117
1118     file->Write(1000);
1119     file->Write(2000);
1120     file->Write(3000);
1121
1122     TestOK = TestOK && check_dump(ost, "Test_multiple_writes",
1123         static_cast<size_t>(6000), file->GetSize());
1124 }
1125 catch (const exception& err) {
1126     error_msg = err.what();
1127 }
1128 TestOK = TestOK && check_dump(ost, "Test_multiple_writes_-_error_buffer", true, error_msg.empty());
1129 error_msg.clear();
1130
1131 // Write exactly to capacity
1132 try
1133 {
1134     size_t blocks = 5;
1135     size_t blocksize = 1024;
1136     File::Sptr file = make_shared<File>("exact.txt", blocks, blocksize);
1137
1138     file->Write(blocks * blocksize); // Write exactly to capacity
1139
1140     TestOK = TestOK && check_dump(ost, "Test_write_to_exact_capacity",
1141         blocks * blocksize, file->GetSize());
1142 }
1143 catch (const exception& err) {
1144     error_msg = err.what();
1145 }
1146 TestOK = TestOK && check_dump(ost, "Test_exact_capacity_-_error_buffer", true, error_msg.empty());
1147 error_msg.clear();
1148
1149 // Write exceeds capacity (should throw)
1150 try
1151 {
1152     File::Sptr file = make_shared<File>("overflow.txt", 2, 1024);
1153     file->Write(3000); // Exceeds 2 * 1024 = 2048
1154 }
1155 catch (const exception& err) {
1156     error_msg = err.what();
1157 }
1158 TestOK = TestOK && check_dump(ost, "Test_write_exceeds_capacity",
1159     File::ERR_OUT_OF_SPACE, error_msg);
1160 error_msg.clear();
1161
1162 // Write zero bytes
1163 try
1164 {
1165     File::Sptr file = make_shared<File>("zero.txt", 10, 2048);
```

```
1165     file->Write(0);
1166 
1167     TestOK = TestOK && check_dump(ost, "Test_write_zero_bytes",
1168         static_cast<size_t>(0), file->GetSize());
1169 }
1170 catch (const exception& err) {
1171     error_msg = err.what();
1172 }
1173 TestOK = TestOK && check_dump(ost, "Test_write_zero_error_buffer", true, error_msg.empty());
1174 error_msg.clear();
1175 
1176 // Multiple writes approaching capacity
1177 try
1178 {
1179     File::Sptr file = make_shared<File>("approach.txt", 3, 1000);
1180     file->Write(1000);
1181     file->Write(1000);
1182     file->Write(1000); // Total = 3000, capacity = 3000
1183 
1184     TestOK = TestOK && check_dump(ost, "Test_multiple_writes_to_capacity",
1185         static_cast<size_t>(3000), file->GetSize());
1186 }
1187 catch (const exception& err) {
1188     error_msg = err.what();
1189 }
1190 TestOK = TestOK && check_dump(ost, "Test_approach_capacity_error_buffer", true, error_msg.empty())
1191 );
1192 error_msg.clear();
1193 
1194 // Write after reaching capacity (should throw)
1195 try
1196 {
1197     File::Sptr file = make_shared<File>("full.txt", 2, 1024);
1198     file->Write(2048); // Fill to capacity
1199     file->Write(1); // Should throw
1200 }
1201 catch (const exception& err) {
1202     error_msg = err.what();
1203 }
1204 TestOK = TestOK && check_dump(ost, "Test_write_when_full", File::ERR_OUT_OF_SPACE, error_msg);
1205 error_msg.clear();
1206 
1207 // File with default blocksize (4096)
1208 try
1209 {
1210     File::Sptr file = make_shared<File>("default.txt", 5); // Default blocksize = 4096
1211     file->Write(10000);
1212 
1213     TestOK = TestOK && check_dump(ost, "Test_default_blocksize", static_cast<size_t>(10000), file->
1214         GetSize());
1215 }
1216 catch (const exception& err) {
1217     error_msg = err.what();
1218 }
1219 TestOK = TestOK && check_dump(ost, "Test_default_blocksize_error_buffer", true, error_msg.empty())
1220 );
1221 error_msg.clear();
1222 
1223 // Accept visitor
1224 try
1225 {
1226     File::Sptr file = make_shared<File>("visitor.txt", 10, 2048);
1227     stringstream result;
1228     DumpVisitor visitor(result);
1229 
1230     file->Accept(visitor);
1231 
1232     TestOK = TestOK && check_dump(ost, "Test_File_Accept_visitor", true, result.str().find("visitor
1233         .txt") != string::npos);
1234 }
1235 catch (const exception& err) {
1236     error_msg = err.what();
1237 }
1238 TestOK = TestOK && check_dump(ost, "Test_File_Accept_error_buffer", true, error_msg.empty());
1239 error_msg.clear();
```

```
1236 // SetName on file
1237 try
1238 {
1239     File::Sptr file = make_shared<File>("old.txt", 10, 2048);
1240     file->SetName("new.txt");
1241
1242     TestOK = TestOK && check_dump(ost, "Test_File_SetName",
1243                                     string_view("new.txt"), file->GetName());
1244 }
1245 catch (const exception& err) {
1246     error_msg = err.what();
1247 }
1248 TestOK = TestOK && check_dump(ost, "Test_File_SetName_error_buffer", true, error_msg.empty());
1249 error_msg.clear();
1250
1251 // File AsFolder should return nullptr
1252 try
1253 {
1254     File::Sptr file = make_shared<File>("notfolder.txt", 10, 2048);
1255     IFolder::Sptr folder_ptr = file->AsFolder();
1256
1257     TestOK = TestOK && check_dump(ost, "Test_File_AsFolder_returns nullptr", true, folder_ptr ==
1258                                   nullptr);
1259 }
1260 catch (const exception& err) {
1261     error_msg = err.what();
1262 }
1263 TestOK = TestOK && check_dump(ost, "Test_File_AsFolder_error_buffer", true, error_msg.empty());
1264 error_msg.clear();
1265
1266 ost << TestEnd;
1267 return TestOK;
1268 }
```

## 6.26 Test.hpp

```
1 //*****\n2 * \file Test.hpp\n3 * \brief File that provides a Test Function with a formated output\n4 *\n5 * \author Simon\n6 * \date April 2025\n7 *****\n8 #ifndef TEST_HPP\n9 #define TEST_HPP\n10\n11 #include <string>\n12 #include <iostream>\n13 #include <vector>\n14 #include <list>\n15 #include <queue>\n16 #include <forward_list>\n17\n18 #define ON 1\n19 #define OFF 0\n20 #define COLOR_OUTPUT OFF\n21\n22 // Definitions of colors in order to change the color of the output stream.\n23 inline const char* colorRed() { return "\x1B[31m"; }\n24 inline const char* colorGreen() { return "\x1B[32m"; }\n25 inline const char* colorWhite() { return "\x1B[37m"; }\n26\n27 inline std::ostream& RED(std::ostream& ost) {\n28     if (ost.good()) {\n29         ost << colorRed();\n30     }\n31     return ost;\n32 }\n33 inline std::ostream& GREEN(std::ostream& ost) {\n34     if (ost.good()) {\n35         ost << colorGreen();\n36     }\n37     return ost;\n38 }\n39 inline std::ostream& WHITE(std::ostream& ost) {\n40     if (ost.good()) {\n41         ost << colorWhite();\n42     }\n43     return ost;\n44 }\n45\n46 inline std::ostream& TestStart(std::ostream& ost) {\n47     if (ost.good()) {\n48         ost << std::endl;\n49         ost << "*****\n50         ost << "TESTCASE_START" << std::endl;\n51         ost << "*****\n52         ost << std::endl;\n53     }\n54     return ost;\n55 }\n56\n57 inline std::ostream& TestEnd(std::ostream& ost) {\n58     if (ost.good()) {\n59         ost << std::endl;\n60         ost << "*****\n61         ost << std::endl;\n62     }\n63     return ost;\n64 }\n65\n66 inline std::ostream& TestCaseOK(std::ostream& ost) {\n67\n68 #if COLOR_OUTPUT\n69     if (ost.good()) {\n70         ost << colorGreen() << "TEST_OK!!" << colorWhite() << std::endl;\n71     }\n72 #else
```

```

73     if (ost.good()) {
74         ost << "TEST_OK!!" << std::endl;
75     }
76 #endif // COLOR_OUTPUT
77
78     return ost;
79 }
80
81 inline std::ostream& TestCaseFail(std::ostream& ost) {
82
83 #if COLOR_OUTPUT
84     if (ost.good()) {
85         ost << colorRed() << "TEST_FAILED_!!" << colorWhite() << std::endl;
86
87     }
88 #else
89     if (ost.good()) {
90         ost << "TEST_FAILED_!!" << std::endl;
91
92     }
93 #endif // COLOR_OUTPUT
94
95     return ost;
96 }
97
98 /**
99  * \brief function that reports if the testcase was successful.
100 *
101 * \param testcase      String that indicates the testcase
102 * \param successful true -> reports to cout test OK
103 * \param successful false -> reports test failed
104 */
105 template <typename T>
106 bool check_dump(std::ostream& ostr, const std::string& testcase, const T& expected, const T& result) {
107     if (ostr.good()) {
108 #if COLOR_OUTPUT
109         if (expected == result) {
110             ostr << testcase << std::endl << colorGreen() << "[Test_OK]" << colorWhite()
111             << "Result:(Expected:" << std::boolalpha << expected << "!=" << "Result"
112             : " " << result << ")" << std::noboolalpha << std::endl << std::endl;
113         }
114         else {
115             ostr << testcase << std::endl << colorRed() << "[Test_FAILED]" << colorWhite()
116             << "Result:(Expected:" << std::boolalpha << expected << "!=" << "Result"
117             : " " << result << ")" << std::noboolalpha << std::endl << std::endl;
118         }
119     }
120     else {
121         if (expected == result) {
122             ostr << testcase << std::endl << "[Test_OK]" << "Result:(Expected:" << std::boolalpha
123             << expected << "!=" << "Result:" << result << ")" << std::noboolalpha << std::endl << std::endl;
124         }
125         else {
126             ostr << testcase << std::endl << "[Test_FAILED]" << "Result:(Expected:" << std::boolalpha
127             << expected << "!=" << "Result:" << result << ")" << std::noboolalpha << std::endl << std::endl;
128         }
129     }
130     return expected == result;
131 }
132
133 template <typename T1, typename T2>
134 std::ostream& operator<< (std::ostream& ost, const std::pair<T1,T2> & p) {
135     if (!ost.good()) throw std::runtime_error("Error:Bad_Ostream!");
136     ost << "(" << p.first << "," << p.second << ")";
137     return ost;
138 }
139

```

```
140 template <typename T>
141 std::ostream& operator<< (std::ostream& ost, const std::vector<T> & cont) {
142     if (!ost.good()) throw std::runtime_error("Error_bad_Ostream!");
143     std::copy(cont.cbegin(), cont.cend(), std::ostream_iterator<T>(ost, " "));
144     return ost;
145 }
146
147 template <typename T>
148 std::ostream& operator<< (std::ostream& ost, const std::list<T> & cont) {
149     if (!ost.good()) throw std::runtime_error("Error_bad_Ostream!");
150     std::copy(cont.cbegin(), cont.cend(), std::ostream_iterator<T>(ost, " "));
151     return ost;
152 }
153
154 template <typename T>
155 std::ostream& operator<< (std::ostream& ost, const std::deque<T> & cont) {
156     if (!ost.good()) throw std::runtime_error("Error_bad_Ostream!");
157     std::copy(cont.cbegin(), cont.cend(), std::ostream_iterator<T>(ost, " "));
158     return ost;
159 }
160
161 template <typename T>
162 std::ostream& operator<< (std::ostream& ost, const std::forward_list<T> & cont) {
163     if (!ost.good()) throw std::runtime_error("Error_bad_Ostream!");
164     std::copy(cont.cbegin(), cont.cend(), std::ostream_iterator<T>(ost, " "));
165     return ost;
166 }
167
168
169 #endif // !TEST_HPP
```