

Name: Simon Offenberger / Simon Vogelhuber

Aufwand in h: siehe Doku.

Mat.Nr: S2410306027 / S2410306014

Punkte:

Übungsgruppe: 1

korrigiert:

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

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

Eine Datei hat zusätzlich folgende Eigenschaften:

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

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

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

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

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

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

Name des Elementes auf das verwiesen wird ausgegeben werden.

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

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

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



HSD

FH-HAGENBERG

Systemdokumentation Projekt Filesystem

Version 1.0

S. Offenberger, S. Vogelhuber

Hagenberg, 8. Dezember 2025

Inhaltsverzeichnis

1	Organisatorisches	6
1.1	Team	6
1.2	Aufteilung der Verantwortlichkeitsbereiche	6
1.3	Aufwand	7
2	Anforderungsdefinition (Systemspezifikation)	8
2.1	Systemüberblick	8
2.2	Funktionale Anforderungen	8
2.2.1	Dateien	8
2.2.2	Ordner	9
2.2.3	Verweise	9
2.3	Erzeugung der Elemente	9
2.4	Besucher (Visitor) Anforderungen	10
2.4.1	Visitor: Dump	10
2.4.2	Visitor: FilterFiles	10
3	Systementwurf	11
3.1	Klassendiagramm	11
3.2	Designentscheidungen	12
3.3	Composite Pattern	12
3.4	Factory Pattern	13
3.5	Visitor Pattern	13
3.6	Template Methode Pattern	13
4	Dokumentation der Komponenten (Klassen)	14
5	Testprotokollierung	15
6	Quellcode	25
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	FObject.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

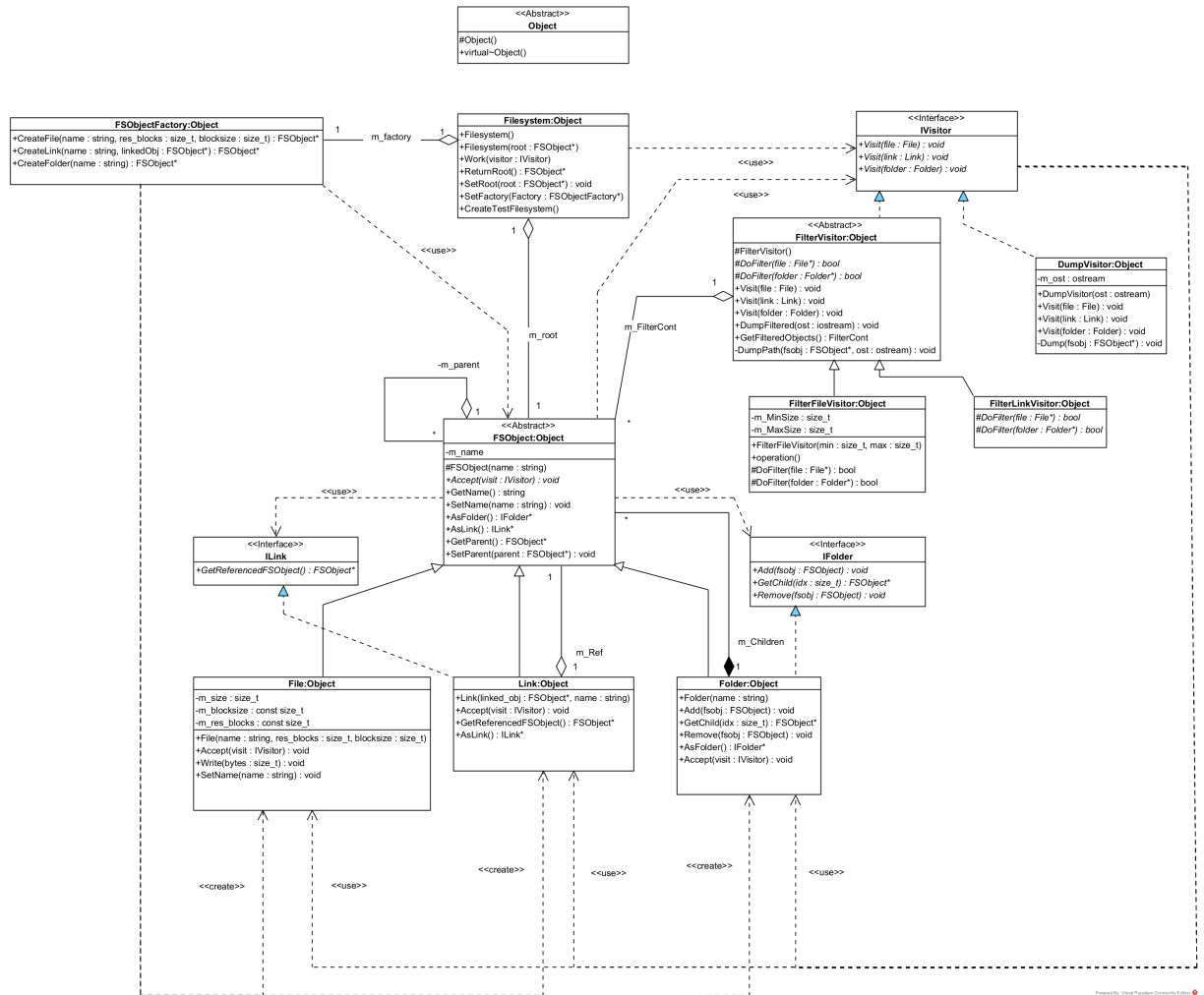
- Gibt die gesamte Dateisystemhierarchie aus.
- Ausgabe sowohl auf der Standardausgabe als auch in einer Datei möglich.
- Muss Dateien, Ordner und Verweise in strukturierter Form darstellen.

2.4.2 Visitor: FilterFiles

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

3 Systementwurf

3.1 Klassendiagramm



3.2 Designentscheidungen

Aus der Aufgabenstellung lassen sich folgenden Designpattern ableiten:

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

3.3 Composite Pattern

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

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

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

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

3.4 Factory Pattern

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

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

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

3.5 Visitor Pattern

Das Visitor Pattern wird verwendet, um verschiedene Operationen auf den Dateisystemelementen durchzuführen, ohne die Klassenhierarchie der 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 *****
3             TESTCASE START
4 *****
5
6 DumpVisitor Test
7 [Test OK] Result: (Expected: |---[root/]
8 | |---[sub_folder/]
9 | | |---[sub_sub_folder/]
10 | | | |---[file1.txt]
11 == Result: |---[root/]
12 | |---[sub_folder/]
13 | | |---[sub_sub_folder/]
14 | | | |---[file1.txt]
15 )
16
17 Test Exception in TestCase
18 [Test OK] Result: (Expected: true == Result: true)
19
20 Test Exception Bad Ostream in DumpVisitor
21 [Test OK] Result: (Expected: ERROR: bad output stream ==
    ↪ Result: ERROR: bad output stream)
22
23
24 *****
25
26
27 *****
28             TESTCASE START
29 *****
30
31 Test Exception nullptr in Visit File
32 [Test OK] Result: (Expected: ERROR Nullptr == Result: ERROR
    ↪ Nullptr)
33
34 Test Exception nullptr in Visit Folder
35 [Test OK] Result: (Expected: ERROR Nullptr == Result: ERROR
    ↪ Nullptr)
36
37 Test Exception nullptr in Visit Link
38 [Test OK] Result: (Expected: ERROR Nullptr == Result: ERROR
    ↪ Nullptr)
```

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



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

```
118 | \root\sub_folder\sub_sub_folder\file1.txt
119 | == Result: \root\file3.txt
120 | \root\sub_folder\sub_sub_folder\file1.txt
121 | )
122 |
123 | Test for Exception in Testcase
124 | [Test OK] Result: (Expected: true == Result: true)
125 |
126 | Test for Exception in Dump with bad Ostream
127 | [Test OK] Result: (Expected: ERROR: bad output stream ==
    | ↪ Result: ERROR: bad output stream)
128 |
129 | Test for Exception in Filter File Visiter CTOR
130 | [Test OK] Result: (Expected: Invalid size range: minimum size
    | ↪ must be less than maximum size == Result: Invalid size
    | ↪ range: minimum size must be less than maximum size)
131 |
132 |
133 | *****
134 |
135 |
136 | *****
137 | TESTCASE START
138 | *****
139 |
140 | Test if file was constructed
141 | [Test OK] Result: (Expected: true == Result: true)
142 |
143 | Test if Link was constructed
144 | [Test OK] Result: (Expected: true == Result: true)
145 |
146 | Test if Folder was constructed
147 | [Test OK] Result: (Expected: true == Result: true)
148 |
149 | Test for 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
    | ↪ Nullptr)
154 |
155 |
156 | *****
157 |
```

```
158
159 *****
160             TESTCASE START
161 *****
162
163 Test normal CTOR Link
164 [Test OK] Result: (Expected: MyFolder == Result: MyFolder)
165
166 Test normal CTOR Link
167 [Test OK] Result: (Expected: LinkToMyFolder == Result:
    ↪ LinkToMyFolder)
168
169 Test normal CTOR Link - error buffer
170 [Test OK] Result: (Expected: true == Result: true)
171
172 Test Exception nullptr CTOR Link
173 [Test OK] Result: (Expected: ERROR Nullptr == Result: ERROR
    ↪ Nullptr)
174
175 Test Exception empty string CTOR Link
176 [Test OK] Result: (Expected: ERROR String Empty == Result:
    ↪ ERROR String Empty)
177
178 Test GetReferencedFSObject
179 [Test OK] Result: (Expected: file1.txt == Result: file1.txt)
180
181 Empty error buffer
182 [Test OK] Result: (Expected: true == Result: true)
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:
    ↪ 0000000000000000)
242
243 Test Folder - error buffer
244 [Test OK] Result: (Expected: true == Result: true)
245
246 Test Remove Child from Folder
247 [Test OK] Result: (Expected: 0000023F511390B0 == Result:
    ↪ 0000023F511390B0)
248
249 Test Folder - error buffer
250 [Test OK] Result: (Expected: true == Result: true)
251
252 Test Folder - add nullptr
253 [Test OK] Result: (Expected: ERROR Nullptr == Result: ERROR
    ↪ Nullptr)
254
255 Test Folder - CTOR with empty string
256 [Test OK] Result: (Expected: ERROR String Empty == Result:
    ↪ ERROR String Empty)
257
258 Test nested folders - root has sub1
259 [Test OK] Result: (Expected: 0000023F51138FF0 == Result:
    ↪ 0000023F51138FF0)
260
261 Test nested folders - sub1 has sub2
262 [Test OK] Result: (Expected: 0000023F511390C0 == Result:
    ↪ 0000023F511390C0)
263
264 Test nested folders - error buffer
265 [Test OK] Result: (Expected: true == Result: true)
266
267 Test parent pointer set on Add
268 [Test OK] Result: (Expected: parent == Result: parent)
269
270 Test parent pointer - error buffer
271 [Test OK] Result: (Expected: true == Result: true)
272
273 Test remove non-existent child
274 [Test OK] Result: (Expected: 0000023F51138FF0 == Result:
    ↪ 0000023F51138FF0)
275
276 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:
    ↪ 0000023F51138FF0)
281
282 Test mixed children - folder
283 [Test OK] Result: (Expected: 0000023F511390B8 == Result:
    ↪ 0000023F511390B8)
284
285 Test mixed children - link
286 [Test OK] Result: (Expected: 0000023F51146730 == Result:
    ↪ 0000023F51146730)
287
288 Test mixed children - error buffer
289 [Test OK] Result: (Expected: true == Result: true)
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 ==  
    ↪ Result: Not enough space to write data)  
361  
362 Test default blocksize  
363 [Test OK] Result: (Expected: 10000 == Result: 10000)  
364  
365 Test default blocksize - error buffer  
366 [Test OK] Result: (Expected: true == Result: true)  
367  
368 Test File Accept visitor  
369 [Test OK] Result: (Expected: true == Result: true)  
370  
371 Test File Accept - error buffer  
372 [Test OK] Result: (Expected: true == Result: true)  
373  
374 Test File SetName  
375 [Test OK] Result: (Expected: new.txt == Result: new.txt)  
376  
377 Test File SetName - error buffer  
378 [Test OK] Result: (Expected: true == Result: true)  
379  
380 Test File AsFolder returns nullptr  
381 [Test OK] Result: (Expected: true == Result: true)  
382  
383 Test File AsFolder - error buffer  
384 [Test OK] Result: (Expected: true == Result: true)  
385  
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::Sptr CreateFolder(std::string_view name = "") const;  
40  
41     /** \brief Create a Link FSObject  
42     * \param name Name of the link  
43     * \param linkedObj Shared pointer to linked FSObject  
44     * \return Shared pointer to created Link FSObject  
45     */  
46     FSObject::Sptr CreateLink(std::string_view name, FSObject::Sptr linkedObj) const;  
47 private:  
48 };  
49 #endif
```

6.3 FSObjectFactory.cpp

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

6.4 Filesystem.hpp

```
1  /*****  
2  * \file Filesystem.hpp  
3  * \brief Filesystem class representing the root of a filesystem  
4  *  
5  * \author Simon  
6  * \date November 2025  
7  *****/  
8  #ifndef FILE_SYSTEM_HPP  
9  #define FILE_SYSTEM_HPP  
10  
11 #include "FSObject.hpp"  
12 #include "IVisitor.hpp"  
13 #include "FSObjectFactory.hpp"  
14  
15 class FileSystem : public Object  
16 {  
17 public:  
18  
19     // Public Error Messages  
20     inline static const std::string ERROR_NULLPTR = "ERROR_Nullptr";  
21  
22     FileSystem() = default;  
23  
24     /** \brief Construct a FileSystem with a root FSObject  
25     * \param root Root FSObject shared pointer  
26     */  
27     FileSystem(FSObject::Sptr root);  
28  
29     /** \brief Walk the filesystem with a visitor  
30     * \param visitor Visitor to apply  
31     * \return Reference to visitor  
32     */  
33     void Work(IVisitor& visitor);  
34  
35     /** \brief Returns the root FSObject  
36     * \return Shared pointer to root  
37     */  
38     FSObject::Sptr ReturnRoot();  
39  
40     /** \brief Set the filesystem root  
41     * \param root Shared pointer to new root  
42     */  
43     void SetRoot(FSObject::Sptr root);  
44  
45     /** \brief Set the filesystem root  
46     * \param root Shared pointer to new root  
47     */  
48     void SetFactory(FSObjectFactory::Uptr Factory);  
49  
50     /**  
51     * \brief Creates a Test Filesystem using the Factory.  
52     * \throw std::invalid_argument if Factory is nullptr.  
53     */  
54     void CreateTestFilesystem();  
55  
56 private:  
57     FSObject::Sptr m_Root;  
58     FSObjectFactory::Uptr m_Factory;  
59 };  
60 #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
62     /** \brief Get parent as weak pointer
63     * \return Weak pointer to parent
64     */
65     FSObj_Wptr GetParent() const;
66
67     /** \brief Set parent of this FSObject
68     * \param parent Shared pointer to parent FSObject
69     */
70     void SetParent(Sptr parent);
71
72 protected:

```

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


6.8 File.hpp

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

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

6.10 IFolder.hpp

```
1  /*****
2  * \file IFolder.hpp
3  * \brief Interface for folder-like FSObjects
4  *
5  * \author Simon
6  * \date   November 2025
7  *****/
8  #ifndef IFOLDER_HPP
9  #define IFOLDER_HPP
10 #include <memory>
11
12 // fwd declaration
13 class FSObject;
14
15 // Type aliases
16 using FSObj_Sptr = std::shared_ptr<FSObject>;
17 using FSObj_Wptr = std::weak_ptr<FSObject>;
18
19 class IFolder
20 {
21 public:
22
23     using Sptr = std::shared_ptr<IFolder>;
24
25     /** \brief Add a child FSObject to the folder
26      * \param fsobj Shared pointer to the FSObject to add
27      */
28     virtual void Add(FSObj_Sptr fsobj) =0;
29
30     /** \brief Get a child by index
31      * \param idx Index of the child
32      * \return Shared pointer to the child or nullptr if out of range
33      */
34     virtual FSObj_Sptr GetChild(size_t idx) const =0;
35
36     /** \brief Remove a child FSObject from the folder
37      * \param fsobj Shared pointer to the FSObject to remove
38      */
39     virtual void Remove(FSObj_Sptr fsobj) =0;
40
41     /** \brief Virtual destructor */
42     virtual ~IFolder() = default;
43
44 private:
45 };
46
47 #endif
```

6.11 Folder.hpp

```

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

```

6.12 Folder.cpp

```
1  /*****  
2  * \file Folder.cpp  
3  * \brief Folder class representing a folder in the filesystem  
4  *  
5  * \author Simon  
6  * \date   November 2025  
7  *****/  
8  #include "Folder.hpp"  
9  #include <stdexcept>  
10 /** \brief Add child to folder, sets parent pointer on child */  
11 void Folder::Add(FSObj_Sptr fsobj)  
12 {  
13     if (fsobj == nullptr) throw std::invalid_argument(FSObject::ERROR_NULLPTR);  
14  
15     fsobj->SetParant(std::move(shared_from_this()));  
16  
17     m_Children.emplace_back(move(fsobj));  
18 }  
19  
20 /** \brief Get child by index */  
21 FSObj_Sptr Folder::GetChild(const size_t idx) const  
22 {  
23     if (idx < m_Children.size())  
24     {  
25         return m_Children.at(idx);  
26     }  
27  
28     return nullptr;  
29 }  
30  
31 /** \brief Remove a child from container */  
32 void Folder::Remove(FSObj_Sptr fsobj)  
33 {  
34     m_Children.erase(  
35         std::remove(m_Children.begin(), m_Children.end(), fsobj), m_Children.end()  
36     );  
37 }  
38  
39 /** \brief Return this as IFolder shared pointer */  
40 std::shared_ptr<const IFolder> Folder::AsFolder() const  
41 {  
42     return shared_from_this();  
43 }  
44  
45 IFolder::Sptr Folder::AsFolder()  
46 {  
47     return shared_from_this();  
48 }  
49  
50 /** \brief Accept a visitor and forward to children */  
51 void Folder::Accept(IVisitor& visit)  
52 {  
53     visit.Visit(move(shared_from_this()));  
54  
55     for(auto& child : m_Children)  
56     {  
57         child->Accept(visit);  
58     }  
59 }
```

6.13 ILink.hpp

```
1  /*****  
2  * \file ILink.hpp  
3  * \brief Interface for folder-like FSObjects  
4  *  
5  * \author Simon  
6  * \date   November 2025  
7  *****/  
8  #ifndef ILink_HPP  
9  #define ILink_HPP  
10 #include <memory>  
11  
12 // fwd declaration  
13 class FSObject;  
14  
15 // Type aliases  
16 using FSObj_Sptr = std::shared_ptr<FSObject>;  
17 using FSObj_Wptr = std::weak_ptr<FSObject>;  
18  
19 class ILink  
20 {  
21 public:  
22  
23     using Sptr = std::shared_ptr<ILink>;  
24  
25     /** \brief Get the referenced FSObject  
26      * \return Shared pointer to the referenced FSObject or nullptr if expired  
27      */  
28     virtual FSObj_Sptr GetReferncedFSObject() const =0;  
29  
30     /** \brief Virtual destructor */  
31     virtual ~ILink() = default;  
32  
33 private:  
34 };  
35  
36 #endif
```

6.14 Link.hpp

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

6.15 Link.cpp

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


6.16 IVisitor.hpp

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

6.17 FilterVisitor.hpp

```

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

```

```
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
19 void FilterVisitor::DumpPath(const std::weak_ptr<const FSObject> & fsobj, std::ostream& ost) const
20 {
21     // end recursion on expired weak pointer
22     if (fsobj.expired()) return;
23
24     const auto obj = fsobj.lock();
25     if (!obj) return; // defensive: lock could fail
26
27     // first dump parent path
28     DumpPath(obj->GetParent(), ost);
29
30     if (!ost.good()) throw std::invalid_argument(FilterVisitor::ERROR_BAD_OSTREAM);
31
32     ost << "\\\" << obj->GetName();
33
34     const std::shared_ptr<const ILink> link_ptr = obj->AsLink();
35
36     if (link_ptr) {
37         const FSObject::Sptr linked_obj = link_ptr->GetReferncedFSObject();
38         if (linked_obj) {
39             ost << "└─>" << linked_obj->GetName();
40         }
41         else {
42             ost << "└─>" << "linked_Object_Expired!";
43         }
44     }
45 }
46
47 /** \brief Default visit for folder (no-op) */
48 void FilterVisitor::Visit(const std::shared_ptr<const Folder> folder)
49 {
50     if (folder == nullptr) throw std::invalid_argument(ERROR_NULLPTR);
51 }
52
53 /** \brief Visit a file and if it matches add to filtered container */
54 void FilterVisitor::Visit(const std::shared_ptr<const File> file)
55 {
56     if (file == nullptr) throw std::invalid_argument(ERROR_NULLPTR);
57
58     // if file matches filter add to container
59     if (DoFilter(file))
60     {
61         m_FilterCont.emplace_back(file);
62     }
63 }
64
65 /** \brief Visit a link and if it matches add to filtered container */
66 void FilterVisitor::Visit(const std::shared_ptr<const Link> link)
67 {
68     if (link == nullptr) throw std::invalid_argument(ERROR_NULLPTR);
69
70     // if link matches filter add to container
71     if (DoFilter(link))
72     {

```

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

6.19 FilterFileVisitor.hpp

```
1  /*****  
2  * \file FilterFileVisitor.hpp  
3  * \brief Visitor that filters files by size range  
4  *  
5  * \author Simon  
6  * \date   November 2025  
7  *****/  
8  #ifndef FILTER_FILE_VISITOR_HPP  
9  #define FILTER_FILE_VISITOR_HPP  
10  
11 #include "FilterVisitor.hpp"  
12  
13 class FilterFileVisitor : public FilterVisitor  
14 {  
15 public:  
16     // Public Error Messages  
17     inline static const std::string ERROR_INVALID_SIZE_RANGE = "Invalid_size_range:_minimum_size_  
18         must_be_less_than_maximum_size";  
19  
20     /** \brief Construct file filter with size range [min,max]  
21     * \param min Minimum size (inclusive) call by value for built-in type -> is faster than by  
22         reference  
23     * \param max Maximum size (inclusive) call by value for built-in type -> is faster than by  
24         reference  
25     */  
26     FilterFileVisitor(const size_t min, const size_t max);  
27  
28 protected:  
29     /** \brief Do filter check for files  
30     * \param file File to check  
31     * \return true if file size is within range  
32     */  
33     virtual bool DoFilter(const std::shared_ptr<const File>& file) const override;  
34  
35     /** \brief Links are not accepted by this filter  
36     * \param link Link to check  
37     * \return false always  
38     */  
39     virtual bool DoFilter(const std::shared_ptr<const Link>& link) const override;  
40  
41 private:  
42     // cannot be const because there are checks in the constructor  
43     size_t m_MinSize;  
44     size_t m_MaxSize;  
45 };  
46 #endif
```

6.20 FilterFileVisitor.cpp

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

6.21 FilterLinkVisitor.hpp

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


6.22 FilterLinkVisitor.cpp

```
1  /**
2   * \file   FilterLinkVisitor.cpp
3   * \brief  Visitor that filters links in the filesystem
4   *
5   * \author Simon
6   * \date   December 2025
7   */
8  #include "FilterLinkVisitor.hpp"
9  #include <cassert>
10
11 /** \brief Files are not accepted by link filter */
12 bool FilterLinkVisitor::DoFilter(const std::shared_ptr<const File>& file) const
13 {
14     assert(file != nullptr);
15     return false;
16 }
17
18 /** \brief Links are accepted by link filter */
19 bool FilterLinkVisitor::DoFilter(const std::shared_ptr<const Link>& link) const
20 {
21     assert(link != nullptr);
22     return true;
23 }
```

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

```

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

```

6.25 main.cpp

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

```
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 << "Unhandelt_Exception" << 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         FSObjectFactory factory;
146         FSObject::Sptr root_folder = factory.CreateFolder("root");
147         FSObject::Sptr sub_folder = factory.CreateFolder("sub_folder");
```

```

148         FSObject::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         FSObjectFactory factory;
187         FSObject::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::
        ERROR_BAD_OSTREAM, error_msg);
212     error_msg.clear();
213
214     ost << TestEnd;
215
216     return TestOK;
217 }
218
219 bool TestFilterLinkVisitor(ostream& ost)
220 {
221     assert(ost.good());

```

```
222     ost << TestStart;
223
224
225     bool TestOK = true;
226     string error_msg;
227
228
229     try {
230         FSObjectFactory factory;
231         FSObject::Sptr root_folder = factory.CreateFolder("root");
232         FSObject::Sptr sub_folder = factory.CreateFolder("sub_folder");
233         FSObject::Sptr sub_sub_folder = factory.CreateFolder("sub_sub_folder");
234         File::Sptr file = make_shared<File>("file1.txt", 2048);
235         Link::Sptr link = make_shared<Link>(file, "LinkToFile1");
236         sub_sub_folder->AsFolder()->Add(file);
237         sub_sub_folder->AsFolder()->Add(link);
238         sub_folder->AsFolder()->Add(sub_sub_folder);
239         root_folder->AsFolder()->Add(sub_folder);
240
241         FilterLinkVisitor link_filter;
242
243         root_folder->Accept(link_filter);
244
245         TestOK = TestOK && check_dump(ost, "FilterLinkVisitor_Test_filtered_amount",
246                                     static_cast<size_t>(1), link_filter.GetFilteredObjects().size());
247         TestOK = TestOK && check_dump(ost, "FilterLinkVisitor_Test_filtered_obj", link->
248                                     GetReferncedFSObject()->GetName(), link_filter.GetFilteredObjects().cbegin()->lock
249                                     ()->AsLink()->GetReferncedFSObject()->GetName());
250
251         stringstream result;
252         stringstream expected;
253
254         link_filter.DumpFiltered(result);
255
256         expected << "\\root\\sub_folder\\sub_sub_folder\\LinkToFile1->_file1.txt" << std::endl
257                 ;
258
259         TestOK = TestOK && check_dump(ost, "Filter_Link_Visitor_Test_Dump_", 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     }
289     catch (const string& err) {
290         error_msg = err;
291     }
292     catch (bad_alloc const& error) {
293         error_msg = error.what();
294     }
295     catch (const exception& err) {
296         error_msg = err.what();
297     }
298     catch (...) {
299         error_msg = "Unhandelt_Exception";
300     }
```

```
292     catch (...) {
293         error_msg = "Unhandelt_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     try {
315         FSObjectFactory factory;
316         FSObject::Sptr root_folder = factory.CreateFolder("root");
317         FSObject::Sptr sub_folder = factory.CreateFolder("sub_folder");
318         FSObject::Sptr sub_sub_folder = factory.CreateFolder("sub_sub_folder");
319         File::Sptr file = make_shared<File>("file1.txt", 10);
320         File::Sptr file1 = make_shared<File>("file2.txt", 10);
321         File::Sptr file2 = make_shared<File>("file3.txt", 10);
322         File::Sptr file3 = make_shared<File>("file4.txt", 10);
323         Link::Sptr link = make_shared<Link>(file, "LinkToFile1");
324
325         file->Write(8192);
326         file1->Write(4096);
327         file2->Write(6000);
328         file3->Write(1000);
329
330         sub_sub_folder->AsFolder()->Add(file);
331         root_folder->AsFolder()->Add(file2);
332         sub_sub_folder->AsFolder()->Add(link);
333         sub_folder->AsFolder()->Add(sub_sub_folder);
334         sub_folder->AsFolder()->Add(file3);
335         root_folder->AsFolder()->Add(sub_folder);
336         root_folder->AsFolder()->Add(file1);
337
338         FilterFileVisitor file_filter(5000,9000);
339
340         root_folder->Accept(file_filter);
341
342         TestOK = TestOK && check_dump(ost, "FilterFileVisitor_Test_filtered_amount", static_cast<size_t>
343             >(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     catch (const string& err) {
361         error_msg = err;
362     }
363     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 = "Unhandelt_Exception";
369 }
370
371 TestOK = TestOK && check_dump(ost, "Test_for_Exception_in_Testcase", true, error_msg.empty());
372 error_msg.clear();
373
374 try {
375     FilterFileVisitor file_filter(1,2);
376
377     stringstream result;
378     result.setstate(ios::badbit);
379
380     file_filter.DumpFiltered(result);
381 }
382 catch (const string& err) {
383     error_msg = err;
384 }
385 catch (bad_alloc const& error) {
386     error_msg = error.what();
387 }
388 catch (const exception& err) {
389     error_msg = err.what();
390 }
391 catch (...) {
392     error_msg = "Unhandelt_Exception";
393 }
394
395 TestOK = TestOK && check_dump(ost, "Test_for_Exception_in_Dump_with_bad_Ostream", error_msg,
396     FilterLinkVisitor::ERROR_BAD_OSTREAM);
397 error_msg.clear();
398
399 try {
400     FilterFileVisitor file_filter( 2,1 ); // <= should throw invalid size range
401 }
402 catch (const string& err) {
403     error_msg = err;
404 }
405 catch (bad_alloc const& error) {
406     error_msg = error.what();
407 }
408 catch (const exception& err) {
409     error_msg = err.what();
410 }
411 catch (...) {
412     error_msg = "Unhandelt_Exception";
413 }
414
415 TestOK = TestOK && check_dump(ost, "Test_for_Exception_in_Filter_File_Visiter_CTOR", error_msg,
416     FilterFileVisitor::ERROR_INVALID_SIZE_RANGE);
417 error_msg.clear();
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
437         stringstream result;
438
439         File::Sptr file = nullptr;
440
441         visit.Visit(file); // <= sould throw Exception Nullptr
442
443     }
444     catch (const string& err) {
445         error_msg = err;
446     }
447     catch (bad_alloc const& error) {
448         error_msg = error.what();
449     }
450     catch (const exception& err) {
451         error_msg = err.what();
452     }
453     catch (...) {
454         error_msg = "Unhandelt_Exception";
455     }
456
457     TestOK = TestOK && check_dump(ost, "Test_Exception_nullptr_in_Visit_File", DumpVisitor::
458         ERROR_NULLPTR, error_msg);
459     error_msg.clear();
460
461     try {
462
463         stringstream result;
464
465         Folder::Sptr folder = nullptr;
466
467         visit.Visit(folder); // <= sould throw Exception Nullptr
468
469     }
470     catch (const string& err) {
471         error_msg = err;
472     }
473     catch (bad_alloc const& error) {
474         error_msg = error.what();
475     }
476     catch (const exception& err) {
477         error_msg = err.what();
478     }
479     catch (...) {
480         error_msg = "Unhandelt_Exception";
481     }
482
483     TestOK = TestOK && check_dump(ost, "Test_Exception_nullptr_in_Visit_Folder", DumpVisitor::
484         ERROR_NULLPTR, error_msg);
485     error_msg.clear();
486
487     try {
488
489         stringstream result;
490
491         Link::Sptr lnk = nullptr;
492
493         visit.Visit(lnk); // <= sould throw Exception Nullptr
494
495     }
496     catch (const string& err) {
497         error_msg = err;
498     }
499     catch (bad_alloc const& error) {
500         error_msg = error.what();
501     }
502     catch (const exception& err) {
503         error_msg = err.what();
504     }
505     catch (...) {
506         error_msg = "Unhandelt_Exception";
507     }
```

```
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     return TestOK;
513 }
514
515 bool TestFactory(ostream& ost)
516 {
517     assert(ost.good());
518
519     ost << TestStart;
520
521     bool TestOK = true;
522     string error_msg;
523
524     try {
525         FSObjectFactory fact;
526         FSObj_Sptr file = fact.CreateFile("file1.txt", 10);
527         FSObj_Sptr folder = fact.CreateFolder("root");
528         FSObj_Sptr lnk = fact.CreateLink("link_to_file", file);
529
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 = "Unhandelt_Exception";
548     }
549
550     TestOK = TestOK && check_dump(ost, "Test_for_Exception_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 = "Unhandelt_Exception";
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     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->
            GetReferncedFSObject()->GetName());
598         TestOK = TestOK && check_dump(ost, "Test_normal_CTOR_Link", link_name, link->GetName());
599     }
600     catch (const string& err) {
601         error_msg = err;
602     }
603     catch (bad_alloc const& error) {
604         error_msg = error.what();
605     }
606     catch (const exception& err) {
607         error_msg = err.what();
608     }
609     catch (...) {
610         error_msg = "Unhandelt_Exception";
611     }
612
613     TestOK = TestOK && check_dump(ost, "Test_normal_CTOR_Link_-_error_buffer", true, error_msg.empty())
        ;
614     error_msg.clear();
615
616     // test link to nullptr
617     try
618     {
619         Link::Sptr link = make_shared<Link>(nullptr, "LinkToNothing");
620     }
621     catch (const string& err) {
622         error_msg = err;
623     }
624     catch (bad_alloc const& error) {
625         error_msg = error.what();
626     }
627     catch (const exception& err) {
628         error_msg = err.what();
629     }
630     catch (...) {
631         error_msg = "Unhandelt_Exception";
632     }
633
634     TestOK = TestOK && check_dump(ost, "Test_Exception_nullptr_CTOR_Link", Link::ERROR_NULLPTR,
        error_msg);
635     error_msg.clear();
636
637     // test Link with empty string
638     try
639     {
640         File::Sptr file = make_shared<File>("file1.txt", 2048);
641         Link::Sptr link = make_shared<Link>(file, "");
642     }
643     catch (const string& err) {
644         error_msg = err;
645     }
646     catch (bad_alloc const& error) {
647         error_msg = error.what();
648     }
649 }
```

```
650     catch (const exception& err) {
651         error_msg = err.what();
652     }
653     catch (...) {
654         error_msg = "Unhandelt_Exception";
655     }
656
657     TestOK = TestOK && check_dump(ost, "Test_Exception_empty_string_CTOR_Link", Link::
        ERROR_STRING_EMPTY, error_msg);
658     error_msg.clear();
659
660     // test Link GetReferencedFSObject
661     try
662     {
663         File::Sptr file = make_shared<File>("file1.txt", 2048);
664         Link::Sptr link = make_shared<Link>(file, file->GetName());
665
666         FSObj_Sptr ref = link->GetReferncedFSObject(); // <= should be File not Folder
667
668         TestOK = TestOK && check_dump(ost, "Test_GetReferencedFSObject", file->GetName(), ref->GetName
            ());
669     }
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 = "Unhandelt_Exception";
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",
            link1->GetName(), link2->GetReferncedFSObject()->GetName());
695     }
696     catch (const exception& err) {
697         error_msg = err.what();
698     }
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",
                true, link->GetReferncedFSObject() != nullptr);
711         } // file goes out of scope here
712
713         FSObj_Sptr expired_ref = link->GetReferncedFSObject();
714         TestOK = TestOK && check_dump(ost, "Test_link_after_object_destruction",
            true, expired_ref == nullptr);
715     }
716     catch (const exception& err) {
717         error_msg = err.what();
718     }
719
720
721     TestOK = TestOK && check_dump(ost, "Test_weak_ptr_expiration_-_error_buffer", true, error_msg.empty
        ());
```

```

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::Sptr 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->GetReferncedFSObject()->GetName());
735 }
736 catch (const exception& err) {
737     error_msg = err.what();
738 }
739 TestOK = TestOK && check_dump(ost, "Test_AsLink()._error_buffer", true, error_msg.empty());
740 error_msg.clear();
741
742 //Link SetName functionality
743 try
744 {
745     File::Sptr file = make_shared<File>("file.txt", 2048);
746     Link::Sptr link = make_shared<Link>(file, "OriginalName");
747
748     link->SetName("NewName");
749     TestOK = TestOK && check_dump(ost, "Test_Link_SetName",
750     string_view("NewName"), link->GetName());
751 }
752 catch (const exception& err) {
753     error_msg = err.what();
754 }
755 TestOK = TestOK && check_dump(ost, "Test_SetName()._error_buffer", true, error_msg.empty());
756 error_msg.clear();
757
758 //Link SetName with empty string (should throw)
759 try
760 {
761     File::Sptr file = make_shared<File>("file.txt", 2048);
762     Link::Sptr link = make_shared<Link>(file, "OriginalName");
763     link->SetName(""); // should throw
764 }
765 catch (const exception& err) {
766     error_msg = err.what();
767 }
768 TestOK = TestOK && check_dump(ost, "Test_Link_SetName._empty_string",
769     FSOBJECT::ERROR_STRING_EMPTY, error_msg);
770 error_msg.clear();
771
772 // Link Accept visitor
773 try
774 {
775     File::Sptr file = make_shared<File>("file.txt", 2048);
776     Link::Sptr link = make_shared<Link>(file, "TestLink");
777     stringstream result;
778     DumpVisitor visitor(result);
779
780     link->Accept(visitor);
781     TestOK = TestOK && check_dump(ost, "Test_Link_Accept._visitor._not_empty",
782     false, result.str().empty());
783 }
784 catch (const exception& err) {
785     error_msg = err.what();
786 }
787 TestOK = TestOK && check_dump(ost, "Test_Link_Accept._error_buffer", true, error_msg.empty());
788 error_msg.clear();
789
790 ost << TestEnd;
791 return TestOK;
792 }
793 bool TestFolder(ostream& ost)
794 {
795     assert(ost.good());
796

```

```
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_normal_CTOR_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         error_msg = err;
826     }
827     catch (bad_alloc const& error) {
828         error_msg = error.what();
829     }
830     catch (const exception& err) {
831         error_msg = err.what();
832     }
833     catch (...) {
834         error_msg = "Unhandelt_Exception";
835     }
836
837     TestOK = TestOK && check_dump(ost, "Test_Folder_-_error_buffer", error_msg.empty(), true);
838     error_msg.clear();
839
840     // test remove child
841     try
842     {
843         Folder::Sptr folder = make_shared<Folder>("MyFolder");
844         File::Sptr file1 = make_shared<File>("file1.txt", 2048);
845         File::Sptr file2 = make_shared<File>("file2.txt", 4096);
846         folder->Add(file1);
847         folder->Add(file2);
848         folder->Remove(file1);
849         TestOK = TestOK && check_dump(ost, "Test_Remove_Child_from_Folder", static_pointer_cast<
850                                     FSObject>(file2), folder->GetChild(0));
851     }
852     catch (const string& err) {
853         error_msg = err;
854     }
855     catch (bad_alloc const& error) {
856         error_msg = error.what();
857     }
858     catch (const exception& err) {
859         error_msg = err.what();
860     }
861     catch (...) {
862         error_msg = "Unhandelt_Exception";
863     }
864
865     TestOK = TestOK && check_dump(ost, "Test_Folder_-_error_buffer", error_msg.empty(), true);
866     error_msg.clear();
867
868     // test add nullptr
869     try
870     {
871         Folder::Sptr folder = make_shared<Folder>("MyFolder");
```

```

869         FSObject::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 = "Unhandelt_Exception";
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 = "Unhandelt_Exception";
904     }
905
906     TestOK = TestOK && check_dump(ost, "Test_Folder_-_CTOR_with_empty_string", FSObject::
        ERROR_STRING_EMPTY, error_msg);
907     error_msg.clear();
908
909     //Nested folder structure
910     try
911     {
912         Folder::Sptr root = make_shared<Folder>("root");
913         Folder::Sptr sub1 = make_shared<Folder>("sub1");
914         Folder::Sptr sub2 = make_shared<Folder>("sub2");
915
916         root->Add(sub1);
917         sub1->Add(sub2);
918
919         TestOK = TestOK && check_dump(ost, "Test_nested_folders_-_root_has_sub1",
920             sub1, static_pointer_cast<Folder>(root->GetChild(0)));
921         TestOK = TestOK && check_dump(ost, "Test_nested_folders_-_sub1_has_sub2",
922             sub2, static_pointer_cast<Folder>(sub1->GetChild(0)));
923     }
924     catch (const exception& err) {
925         error_msg = err.what();
926     }
927     TestOK = TestOK && check_dump(ost, "Test_nested_folders_-_error_buffer", true, error_msg.empty());
928     error_msg.clear();
929
930     //Parent pointer is set correctly when adding child
931     try
932     {
933         Folder::Sptr parent = make_shared<Folder>("parent");
934         File::Sptr child = make_shared<File>("child.txt", 2048);
935
936         parent->Add(child);
937         FSObj_Wptr parent_wptr = child->GetParent();
938         FSObj_Sptr parent_sptr = parent_wptr.lock();
939
940         TestOK = TestOK && check_dump(ost, "Test_parent_pointer_set_on_Add",
941             parent->GetName(), parent_sptr->GetName());
942     }

```



```
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     error_msg.clear();
967
968     //Multiple children of different types
969     try
970     {
971         Folder::Sptr folder = make_shared<Folder>("mixed");
972         File::Sptr file = make_shared<File>("file.txt", 2048);
973         Folder::Sptr subfolder = make_shared<Folder>("subfolder");
974         Link::Sptr link = make_shared<Link>(file, "link");
975
976         folder->Add(file);
977         folder->Add(subfolder);
978         folder->Add(link);
979
980         TestOK = TestOK && check_dump(ost, "Test_mixed_children_ file",
981             static_pointer_cast<FSObject>(file), folder->GetChild(0));
982         TestOK = TestOK && check_dump(ost, "Test_mixed_children_ folder",
983             static_pointer_cast<FSObject>(subfolder), folder->GetChild(1));
984         TestOK = TestOK && check_dump(ost, "Test_mixed_children_ link",
985             static_pointer_cast<FSObject>(link), folder->GetChild(2));
986     }
987     catch (const exception& err) {
988         error_msg = err.what();
989     }
990     TestOK = TestOK && check_dump(ost, "Test_mixed_children_ error_buffer", true, error_msg.empty());
991     error_msg.clear();
992
993     //AsFolder() returns valid pointer
994     try
995     {
996         Folder::Sptr folder = make_shared<Folder>("test");
997         IFolder::Sptr ifolder = folder->AsFolder();
998
999         TestOK = TestOK && check_dump(ost, "Test_AsFolder() returns_valid_pointer",
1000             true, ifolder != nullptr);
1001     }
1002     catch (const exception& err) {
1003         error_msg = err.what();
1004     }
1005     TestOK = TestOK && check_dump(ost, "Test_AsFolder()_ error_buffer", true, error_msg.empty());
1006     error_msg.clear();
1007
1008     //Accept visitor with children
1009     try
1010     {
1011         Folder::Sptr folder = make_shared<Folder>("root");
1012         File::Sptr file = make_shared<File>("file.txt", 2048);
1013         folder->Add(file);
1014
1015         stringstream result;
1016         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 // 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::
    ERROR_STRING_EMPTY);
    error_msg.clear();
1110
1111 // Write multiple times
1112 try
1113 {
1114     File::Sptr file = make_shared<File>("multi.txt", 10, 2048);
1115
1116     file->Write(1000);
1117     file->Write(2000);
1118     file->Write(3000);
1119
1120     TestOK = TestOK && check_dump(ost, "Test_multiple_writes",
1121         static_cast<size_t>(6000), file->GetSize());
1122 }
1123 catch (const exception& err) {
1124     error_msg = err.what();
1125 }
1126
1127 TestOK = TestOK && check_dump(ost, "Test_multiple_writes_-_error_buffer", true, error_msg.empty());
1128 error_msg.clear();
1129
1130 // Write exactly to capacity
1131 try
1132 {
1133     size_t blocks = 5;
1134     size_t blocksize = 1024;
1135     File::Sptr file = make_shared<File>("exact.txt", blocks, blocksize);
1136
1137     file->Write(blocks * blocksize); // Write exactly to capacity
1138
1139     TestOK = TestOK && check_dump(ost, "Test_write_to_exact_capacity",
1140         blocks * blocksize, file->GetSize());
1141 }
1142 catch (const exception& err) {
1143     error_msg = err.what();
1144 }
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
1159 TestOK = TestOK && check_dump(ost, "Test_write_exceeds_capacity",
1160     File::ERR_OUT_OF_SPACE, error_msg);
1161 error_msg.clear();
1162
1163 // Write zero bytes
1164 try
1165 {
1166     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     error_msg.clear();
1192
1193     // Write after reaching capacity (should throw)
1194     try
1195     {
1196         File::Sptr file = make_shared<File>("full.txt", 2, 1024);
1197         file->Write(2048); // Fill to capacity
1198         file->Write(1); // Should throw
1199     }
1200     catch (const exception& err) {
1201         error_msg = err.what();
1202     }
1203     TestOK = TestOK && check_dump(ost, "Test_write_when_full", File::ERR_OUT_OF_SPACE, error_msg);
1204     error_msg.clear();
1205
1206     // File with default blocksize (4096)
1207     try
1208     {
1209         File::Sptr file = make_shared<File>("default.txt", 5); // Default blocksize = 4096
1210         file->Write(10000);
1211
1212         TestOK = TestOK && check_dump(ost, "Test_default_blocksize", static_cast<size_t>(10000), file->
            GetSize());
1213     }
1214     catch (const exception& err) {
1215         error_msg = err.what();
1216     }
1217     TestOK = TestOK && check_dump(ost, "Test_default_blocksize_error_buffer", true, error_msg.empty());
1218     error_msg.clear();
1219
1220     // Accept visitor
1221     try
1222     {
1223         File::Sptr file = make_shared<File>("visitor.txt", 10, 2048);
1224         stringstream result;
1225         DumpVisitor visitor(result);
1226
1227         file->Accept(visitor);
1228
1229         TestOK = TestOK && check_dump(ost, "Test_File_Accept_visitor", true, result.str().find("visitor
            .txt") != string::npos);
1230     }
1231     catch (const exception& err) {
1232         error_msg = err.what();
1233     }
1234     TestOK = TestOK && check_dump(ost, "Test_File_Accept_error_buffer", true, error_msg.empty());
1235     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  /*****
2  * \file   Test.hpp
3  * \brief  File that provides a Test Function with a formatted output
4  *
5  * \author Simon
6  * \date   April 2025
7  *****/
8  #ifndef TEST_HPP
9  #define TEST_HPP
10
11 #include <string>
12 #include <iostream>
13 #include <vector>
14 #include <list>
15 #include <queue>
16 #include <forward_list>
17
18 #define ON 1
19 #define OFF 0
20 #define COLOR_OUTPUT OFF
21
22 // Definitions of colors in order to change the color of the output stream.
23 inline const char* colorRed() { return "\x1B[31m"; }
24 inline const char* colorGreen() { return "\x1B[32m"; }
25 inline const char* colorWhite() { return "\x1B[37m"; }
26
27 inline std::ostream& RED(std::ostream& ost) {
28     if (ost.good()) {
29         ost << colorRed();
30     }
31     return ost;
32 }
33 inline std::ostream& GREEN(std::ostream& ost) {
34     if (ost.good()) {
35         ost << colorGreen();
36     }
37     return ost;
38 }
39 inline std::ostream& WHITE(std::ostream& ost) {
40     if (ost.good()) {
41         ost << colorWhite();
42     }
43     return ost;
44 }
45
46 inline std::ostream& TestStart(std::ostream& ost) {
47     if (ost.good()) {
48         ost << std::endl;
49         ost << "*****" << std::endl;
50         ost << "          TESTCASE_START_" << std::endl;
51         ost << "*****" << std::endl;
52         ost << std::endl;
53     }
54     return ost;
55 }
56
57 inline std::ostream& TestEnd(std::ostream& ost) {
58     if (ost.good()) {
59         ost << std::endl;
60         ost << "*****" << std::endl;
61         ost << std::endl;
62     }
63     return ost;
64 }
65
66 inline std::ostream& TestCaseOK(std::ostream& ost) {
67
68 #if COLOR_OUTPUT
69     if (ost.good()) {
70         ost << colorGreen() << "TEST_OK!!" << colorWhite() << std::endl;
71     }
72 #else

```

```

73         if (ost.good()) {
74             ost << "TEST_OK!!" << std::endl;
75         }
76     #endif // COLOR_OUTPUT
77
78     return ost;
79 }
80
81 inline std::ostream& TestCaseFail(std::ostream& ost) {
82
83     #if COLOR_OUTPUT
84         if (ost.good()) {
85             ost << colorRed() << "TEST_FAILED!!" << colorWhite() << std::endl;
86         }
87     #else
88         if (ost.good()) {
89             ost << "TEST_FAILED!!" << std::endl;
90         }
91     #endif // COLOR_OUTPUT
92
93     return ost;
94 }
95
96 /**
97  * \brief function that reports if the testcase was successful.
98  *
99  * \param testcase      String that indicates the testcase
100  * \param successful true -> reports to cout test OK
101  * \param successful false -> reports test failed
102  */
103
104 template <typename T>
105 bool check_dump(std::ostream& ostr, const std::string& testcase, const T& expected, const T& result) {
106     if (ostr.good()) {
107         #if COLOR_OUTPUT
108             if (expected == result) {
109                 ostr << testcase << std::endl << colorGreen() << "[Test_OK]" << colorWhite()
110                     << "Result:_(Expected:_" << std::boolalpha << expected << "_==" << "_Result
111                     :_" << result << ")" << std::noboolalpha << std::endl << std::endl;
112             }
113             else {
114                 ostr << testcase << std::endl << colorRed() << "[Test_FAILED]" << colorWhite()
115                     << "Result:_(Expected:_" << std::boolalpha << expected << "_!=" << "_
116                     Result:_" << result << ")" << std::noboolalpha << std::endl << std::endl;
117             }
118         #else
119             if (expected == result) {
120                 ostr << testcase << std::endl << "[Test_OK]" << "Result:_(Expected:_" << std::
121                 boolalpha << expected << "_==" << "_Result:_" << result << ")" << std::
122                 noboolalpha << std::endl << std::endl;
123             }
124             else {
125                 ostr << testcase << std::endl << "[Test_FAILED]" << "Result:_(Expected:_" <<
126                 std::boolalpha << expected << "_!=" << "_Result:_" << result << ")" << std
127                 :noboolalpha << std::endl << std::endl;
128             }
129         #endif
130
131         if (ostr.fail()) {
132             std::cerr << "Error:_Write_Ostream" << std::endl;
133         }
134     }
135     else {
136         std::cerr << "Error:_Bad_Ostream" << std::endl;
137     }
138     return expected == result;
139 }
140
141 template <typename T1, typename T2>
142 std::ostream& operator<< (std::ostream& ost, const std::pair<T1,T2> & p) {
143     if (!ost.good()) throw std::runtime_error("Error:_bad_Ostream!");
144     ost << "(" << p.first << "," << p.second << ")";
145     return ost;
146 }

```

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