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

Name: Simon Offenberger / Simon Vogelhuber — Aufwand in h: siehe Doku.

Mat.Nr: S2410306027 / S2410306014 — Punkte:

Übungsgruppe: 1 — korrigiert:

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

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

Eine Datei hat zusätzlich folgende Eigenschaften:

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

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

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

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

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

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

Name des Elementes auf das verwiesen wird ausgegeben werden.

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

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

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

Systemdokumentation
Projekt Filesystem

Version 1.0

S. Offenberger, S. Vogelhuber

Hagenberg, 9. Dezember 2025

Inhaltsverzeichnis

1 Organisatorisches

1.1 Team

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

1.2 Aufteilung der Verantwortlichkeitsbereiche

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

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

1.3 Aufwand

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

2 Anforderungsdefinition (Systemspezifikation)

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

2.1 Systemüberblick

Das System verwaltet drei Arten von Dateisystemelementen:

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

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

2.2 Funktionale Anforderungen

2.2.1 Dateien

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

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

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

- `Write(size_t const bytes)`

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

2.2.2 Ordner

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

2.2.3 Verweise

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

2.3 Erzeugung der Elemente

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

2.4 Besucher (Visitor) Anforderungen

2.4.1 Visitor: Dump

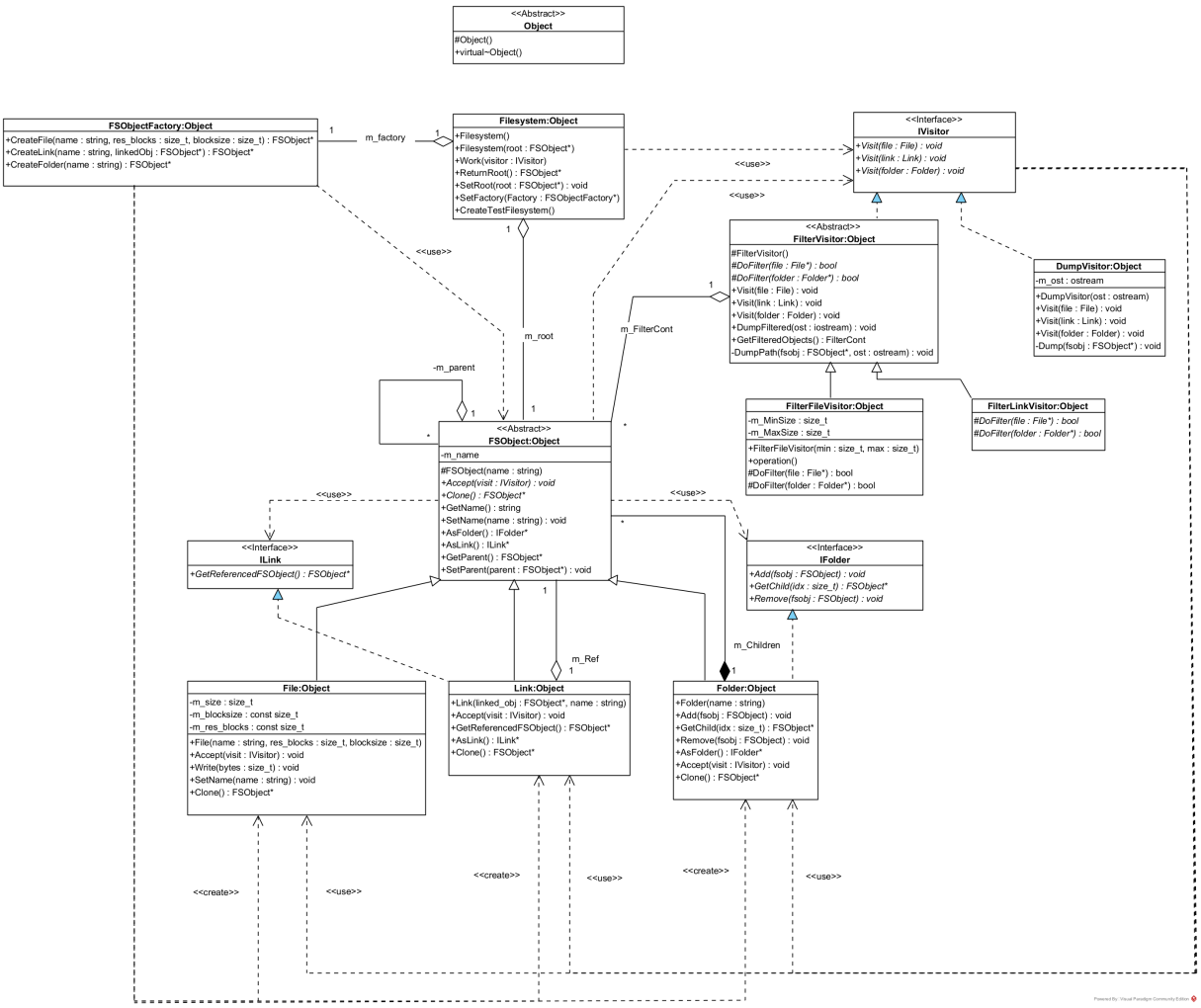
- Gibt die gesamte Dateisystemhierarchie aus.
- Ausgabe sowohl auf der Standardausgabe als auch in einer Datei möglich.
- Muss Dateien, Ordner und Verweise in strukturierter Form darstellen.

2.4.2 Visitor: FilterFiles

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

3 Systementwurf

3.1 Klassendiagramm



3.2 Designentscheidungen

Aus der Aufgabenstellung lassen sich folgenden Designpattern ableiten:

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

3.3 Composite Pattern

Dieses Pattern wird verwendet, um die hierarchische Struktur des Dateisystems abzubilden. Die Basisklasse `FSObject` definiert die gemeinsamen Schnittstellen für alle Dateisystemelemente.
Ordner implementieren die Fähigkeit, andere `FSObject`-Instanzen zu enthalten (wie Dateien, Verweise und weitere Ordner), wodurch eine Baumstruktur entsteht.
Bei der gewählten Implementierung wurde besonders darauf geachtet, dass das Liskovsersche Substitutionsprinzip eingehalten wird. Aus diesem Grund wurden die Methoden zur Verwaltung von Kindobjekten nur in der `Folder`-Klasse implementiert. Die Schnittstelle für die Methoden der besonderen Kindklassen wurden in capabiltiy Interfaces ausgelagert (`IFolder`, `ILink`).
Dadurch wird verhindert, dass Objekte, die keine Kinder enthalten können (wie Dateien und Verweise), diese Methoden erben und somit das Substitutionsprinzip verletzen.

3.3.1 Copy Ctor und Assignment Operator

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

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

3.4 Factory Pattern

Für die konkrete Implementierung der Objekterzeugung wurde das Pattern Simple Factory verwendet. Die Klasse `FSObjectFactory` kapselt die Logik zur Erstellung von Dateien, Ordnern und Verweisen. Dies ermöglicht eine zentrale Verwaltung der Erzeugungslogik und erleichtert zukünftige Erweiterungen. Beim konkreten Desing der Factory wurde auf das Interface zwischen Factory und Client verzichtet, da die Factory nur eine einzige Implementierung besitzt und keine weiteren Varianten geplant sind.
Dadurch wurde die Komplexität reduziert, jedoch bleibt die Erfüllung des Dependency Inversion Prinzips aus. Dies ist aber über die Verwendung der Simple Factory hinweg vertretbar.
(Dies wurde mit Prof. Wiesinger diskutiert, und ist hier zulässig.)

3.5 Visitor Pattern

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

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

3.6 Template Methode Pattern

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

4 Dokumentation der Komponenten (Klassen)

Die HTML-Startdatei befindet sich im Verzeichnis ./../doxy/html/index.html

5 Testprotokollierung

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

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

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

```
\root\sub_folder\sub_sub_folder\file1.txt
 == Result: \root\file3.txt
\root\sub_folder\sub_sub_folder\file1.txt
)

Test for Exception in Testcase
[Test OK] Result: (Expected: true == Result: true)

Test for Exception in Dump with bad Ostream
[Test OK] Result: (Expected: ERROR: bad output stream ==
    ↪ Result: ERROR: bad output stream)

Test for Exception in Filter File Visiter CTOR
[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)


****************************************


****************************************
              TESTCASE START
****************************************

Test if file was constructed
[Test OK] Result: (Expected: true == Result: true)

Test if Link was constructed
[Test OK] Result: (Expected: true == Result: true)

Test if Folder was constructed
[Test OK] Result: (Expected: true == Result: true)

Test for Execption in Tesstcase
[Test OK] Result: (Expected: true == Result: true)

Test Exception nullptr CTOR Link
[Test OK] Result: (Expected: ERROR Nullptr == Result: ERROR
    ↪ Nullptr)


****************************************

```

```
158
159 ****************************************
160               TESTCASE START
161 ****************************************
162
163 Test normal CTOR Link
164 [Test OK] Result: (Expected: MyFolder == Result: MyFolder)
165
166 Test normal CTOR Link
167 [Test OK] Result: (Expected: LinkToMyFolder == Result:
    ↪ LinkToMyFolder)
168
169 Test normal CTOR Link - error buffer
170 [Test OK] Result: (Expected: true == Result: true)
171
172 Test Copy CTOR of Link
173 [Test OK] Result: (Expected: 000001C8BDA974B8 == Result:
    ↪ 000001C8BDA974B8)
174
175 Test for shallow Copy of Link
176 [Test OK] Result: (Expected: 000001C8BDA974B8 == Result:
    ↪ 000001C8BDA974B8)
177
178 Test for parent of Copied Link
179 [Test OK] Result: (Expected: Modified == Result: Modified)
180
181 Test normal COPY CTOR Link - error buffer
182 [Test OK] Result: (Expected: true == Result: true)
183
184 Test Assign Op of Link
185 [Test OK] Result: (Expected: 000001C8BDA974B8 == Result:
    ↪ 000001C8BDA974B8)
186
187 Test Assign Op for Parent of Link
188 [Test OK] Result: (Expected: Modified == Result: Modified)
189
190 Test Assing Op Link - error buffer
191 [Test OK] Result: (Expected: true == Result: true)
192
193 Test Self Assing Op Link - error buffer
194 [Test OK] Result: (Expected: true == Result: true)
195
196 Test Exception nullptr CTOR Link
```

```
[Test OK] Result: (Expected: ERROR Nullptr == Result: ERROR Nullptr)

Test Exception empty string CTOR Link
[Test OK] Result: (Expected: ERROR String Empty == Result: ERROR String Empty)

Test GetReferencedFSObject
[Test OK] Result: (Expected: file1.txt == Result: file1.txt)

Empty error buffer
[Test OK] Result: (Expected: true == Result: true)

Test chained links
[Test OK] Result: (Expected: Link1 == Result: Link1)

Test chained links - error buffer
[Test OK] Result: (Expected: true == Result: true)

Test link before destruction
[Test OK] Result: (Expected: true == Result: true)

Test link after object destruction
[Test OK] Result: (Expected: true == Result: true)

Test weak_ptr expiration - error buffer
[Test OK] Result: (Expected: true == Result: true)

Test AsLink() returns valid pointer
[Test OK] Result: (Expected: true == Result: true)

Test AsLink() reference matches
[Test OK] Result: (Expected: file.txt == Result: file.txt)

Test AsLink() - error buffer
[Test OK] Result: (Expected: true == Result: true)

Test Link SetName
[Test OK] Result: (Expected: NewName == Result: NewName)

Test SetName - error buffer
[Test OK] Result: (Expected: true == Result: true)

Test Link SetName empty string
```

```
[Test OK] Result: (Expected: ERROR String Empty == Result: ERROR String Empty)

Test Link Accept visitor - not empty
[Test OK] Result: (Expected: false == Result: false)

Test Link Accept - error buffer
[Test OK] Result: (Expected: true == Result: true)


****************************************


****************************************
               TESTCASE START
****************************************

Test normal CTOR Folder
[Test OK] Result: (Expected: MyFolder == Result: MyFolder)

Get Child from folder
[Test OK] Result: (Expected: 000001C8BDA97580 == Result: 000001C8BDA97580)

Get next Child from folder
[Test OK] Result: (Expected: 000001C8BDAA78E0 == Result: 000001C8BDAA78E0)

Get Child for invalid index
[Test OK] Result: (Expected: 0000000000000000 == Result: 0000000000000000)

Test Folder - error buffer
[Test OK] Result: (Expected: true == Result: true)

Test Copy Ctor Folder - Child 0
[Test OK] Result: (Expected: file1.txt == Result: file1.txt)

Test Copy Ctor Folder - Sub Folder File
[Test OK] Result: (Expected: sub_file.txt == Result: sub_file.txt)

Test Copy Ctor Folder test for Deep Copy
[Test OK] Result: (Expected: true == Result: true)
```

```
278
279 Test Copy Ctor Folder test for Deep Copy in Sub Folder File
280 [Test OK] Result: (Expected: true == Result: true)
281
282 Test Parent of Copied Folder
283 [Test OK] Result: (Expected: 000001C8BDAA7B28 == Result:
    ↪ 000001C8BDAA7B28)
284
285 Test Folder - error buffer
286 [Test OK] Result: (Expected: true == Result: true)
287
288 Test Assign Op Folder - Child 0
289 [Test OK] Result: (Expected: file1.txt == Result: file1.txt)
290
291 Test Assign Op Folder Parent - Child 0
292 [Test OK] Result: (Expected: MyFolder == Result: MyFolder)
293
294 Test Folder - error buffer
295 [Test OK] Result: (Expected: true == Result: true)
296
297 Test Self Assign Folder - Child 0
298 [Test OK] Result: (Expected: 000001C8BDA97580 == Result:
    ↪ 000001C8BDA97580)
299
300 Test Folder - error buffer
301 [Test OK] Result: (Expected: true == Result: true)
302
303 Test Remove Child from Folder
304 [Test OK] Result: (Expected: 000001C8BDAA78E0 == Result:
    ↪ 000001C8BDAA78E0)
305
306 Test Folder - error buffer
307 [Test OK] Result: (Expected: true == Result: true)
308
309 Test Folder - add nullptr
310 [Test OK] Result: (Expected: ERROR Nullptr == Result: ERROR
    ↪ Nullptr)
311
312 Test Folder - CTOR with empty string
313 [Test OK] Result: (Expected: ERROR String Empty == Result:
    ↪ ERROR String Empty)
314
315 Test nested folders - root has sub1
```

```
316 [Test OK] Result: (Expected: 000001C8BDA97580 == Result:
    ↪ 000001C8BDA97580)
317
318 Test nested folders - sub1 has sub2
319 [Test OK] Result: (Expected: 000001C8BDAA78E0 == Result:
    ↪ 000001C8BDAA78E0)
320
321 Test nested folders - error buffer
322 [Test OK] Result: (Expected: true == Result: true)
323
324 Test parent pointer set on Add
325 [Test OK] Result: (Expected: parent == Result: parent)
326
327 Test parent pointer - error buffer
328 [Test OK] Result: (Expected: true == Result: true)
329
330 Test remove non-existent child
331 [Test OK] Result: (Expected: 000001C8BDA97580 == Result:
    ↪ 000001C8BDA97580)
332
333 Test remove non-existent - error buffer
334 [Test OK] Result: (Expected: true == Result: true)
335
336 Test mixed children - file
337 [Test OK] Result: (Expected: 000001C8BDA97580 == Result:
    ↪ 000001C8BDA97580)
338
339 Test mixed children - folder
340 [Test OK] Result: (Expected: 000001C8BDAA78E8 == Result:
    ↪ 000001C8BDAA78E8)
341
342 Test mixed children - link
343 [Test OK] Result: (Expected: 000001C8BDAA79B0 == Result:
    ↪ 000001C8BDAA79B0)
344
345 Test mixed children - error buffer
346 [Test OK] Result: (Expected: true == Result: true)
347
348 Test AsFolder() returns valid pointer
349 [Test OK] Result: (Expected: true == Result: true)
350
351 Test AsFolder() - error buffer
352 [Test OK] Result: (Expected: true == Result: true)
353
```

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

```
[Test OK] Result: (Expected: ERROR String Empty == Result:
    ↪ ERROR String Empty)

Test multiple writes
[Test OK] Result: (Expected: 6000 == Result: 6000)

Test multiple writes - error buffer
[Test OK] Result: (Expected: true == Result: true)

Test write to exact capacity
[Test OK] Result: (Expected: 5120 == Result: 5120)

Test exact capacity - error buffer
[Test OK] Result: (Expected: true == Result: true)

Test write exceeds capacity
[Test OK] Result: (Expected: Not enough space to write data ==
    ↪  Result: Not enough space to write data)

Test write zero bytes
[Test OK] Result: (Expected: 0 == Result: 0)

Test write zero - error buffer
[Test OK] Result: (Expected: true == Result: true)

Test multiple writes to capacity
[Test OK] Result: (Expected: 3000 == Result: 3000)

Test approach capacity - error buffer
[Test OK] Result: (Expected: true == Result: true)

Test write when full
[Test OK] Result: (Expected: Not enough space to write data ==
    ↪  Result: Not enough space to write data)

Test default blocksize
[Test OK] Result: (Expected: 10000 == Result: 10000)

Test default blocksize - error buffer
[Test OK] Result: (Expected: true == Result: true)

Test File Accept visitor
[Test OK] Result: (Expected: true == Result: true)

```

```
Test File Accept - error buffer
[Test OK] Result: (Expected: true == Result: true)

Test File SetName
[Test OK] Result: (Expected: new.txt == Result: new.txt)

Test File SetName - error buffer
[Test OK] Result: (Expected: true == Result: true)

Test File AsFolder returns nullptr
[Test OK] Result: (Expected: true == Result: true)

Test File AsFolder - error buffer
[Test OK] Result: (Expected: true == Result: true)


****************************************


****************************************
              TESTCASE START
****************************************

Dump of Test Filesystem via Dump Visitor:

|---[root/]
|   |---[file1.txt]
|   |---[file2.txt]
|   |---[file3.txt]
|   |---[file4.txt]
|   |---[sub_folder/]
|   |   |---[file5.txt]
|   |   |---[file6.txt]
|   |   |---[sub_sub_folder/]
|   |   |   |---[file7.txt]
|   |   |   |---[LinkToRoot->]


Test normal op Filesystem - error buffer empty
[Test OK] Result: (Expected: true == Result: true)

Test ReturnRoot matches
[Test OK] Result: (Expected: |---[root/]
 == Result: |---[root/]
```

```
)

Test normal op Filesystem - error buffer empty
[Test OK] Result: (Expected: true == Result: true)

Test Exception Set Null Root
[Test OK] Result: (Expected: ERROR Nullptr == Result: ERROR
    ↪ Nullptr)

Test Exception Set Null Factory
[Test OK] Result: (Expected: ERROR Nullptr == Result: ERROR
    ↪ Nullptr)

Test Exception no Factory in Create Test FileSystem
[Test OK] Result: (Expected: ERROR Nullptr == Result: ERROR
    ↪ Nullptr)

Test Exception Work with no root set
[Test OK] Result: (Expected: ERROR Nullptr == Result: ERROR
    ↪ Nullptr)


*****************************************

TEST OK!!
```

6 Quellcode

6.1 Object.hpp

```
/*****************************************************************//**
 * \file   Object.h
 * \brief  Root base class for all objects
 *
 * \author Simon
 * \date   December 2025
 *********************************************************************/
#ifndef OBJECT_H
#define OBJECT_H

#include <string>

class Object{
protected:
        /** \brief Prevent direct instantiation */
   Object() = default;
public:
        /** \brief Virtual destructor */
   virtual ~Object(){}
};

#endif // OBJECT_H
```

6.2 FSObjectFactory.hpp

```cpp
/*****************************************************************//**
 * \file FSObjectFactory.hpp
 * \brief Simple Factory class to create filesystem objects
 *
 * \author Simon
 * \date   November 2025
 *********************************************************************/
#ifndef FS_OBJECT_FACTORY_HPP
#define FS_OBJECT_FACTORY_HPP

#include "Object.h"
#include "FSObject.hpp"
#include "Folder.hpp"
#include "File.hpp"
#include "Link.hpp"
#include <memory>


class FSObjectFactory : public Object
{
public:
        using Uptr = std::unique_ptr<FSObjectFactory>;

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

        /** \brief Create a File FSObject
         * \param name Name of the file
         * \param res_blocks Reserved blocks
         * \param blocksize Block size (default 4096)
         * \return Shared pointer to created File FSObject
         */
        FSObject::Sptr CreateFile(std::string_view name,const size_t res_blocks,const size_t blocksize
             = 4096) const;

        /** \brief Create a Folder FSObject
         * \param name Name of the folder
         * \return Shared pointer to created Folder FSObject
         */
        FSObject::Sptr CreateFolder(std::string_view name = "") const;

        /** \brief Create a Link FSObject
         * \param name Name of the link
         * \param linkedObj Shared pointer to linked FSObject
         * \return Shared pointer to created Link FSObject
         */
        FSObject::Sptr CreateLink(std::string_view name, FSObject::Sptr linkedObj) const;

private:
};
#endif
```

6.3 FSObjectFactory.cpp

```cpp
/*****************************************************************//**
 * \file   FSObjectFactory.cpp
 * \brief  Simple Factory class to create filesystem objects
 *
 * \author Simon
 * \date   December 2025
 *********************************************************************/

#include "FSObjectFactory.hpp"


FSObject::Sptr FSObjectFactory::CreateFile(std::string_view name,size_t res_blocks, size_t blocksize)
      const
{
        return std::make_shared<File>(name, res_blocks,blocksize);
}

FSObject::Sptr FSObjectFactory::CreateFolder(std::string_view name) const
{
        return std::make_shared<Folder>(name);
}

FSObject::Sptr FSObjectFactory::CreateLink(std::string_view name, FSObject::Sptr linkedObj) const
{
        return std::make_shared<Link>(move(linkedObj),name);
}
```

6.4 Filesystem.hpp

```cpp
/*****************************************************************//**
 * \file Filesystem.hpp
 * \brief Filesystem class representing the root of a filesystem
 *
 * \author Simon
 * \date   November 2025
 *********************************************************************/
#ifndef FILE_SYSTEM_HPP
#define FILE_SYSTEM_HPP

#include "FSObject.hpp"
#include "IVisitor.hpp"
#include "FSObjectFactory.hpp"

class FileSystem : public Object
{
public:

        // Public Error Messages
        inline static const std::string ERROR_NULLPTR = "ERROR_Nullptr";

        FileSystem() = default;

        /** \brief Construct a FileSystem with a root FSObject
         * \param root Root FSObject shared pointer
         */
        FileSystem(FSObject::Sptr root);

        /** \brief Walk the filesystem with a visitor
         * \param visitor Visitor to apply
         * \return Reference to visitor
         */
        void Work(IVisitor& visitor);

        /** \brief Returns the root FSObject
         * \return Shared pointer to root
         */
        FSObject::Sptr ReturnRoot();

        /** \brief Set the filesystem root
         * \param root Shared pointer to new root
         */
        void SetRoot(FSObject::Sptr root);

        /** \brief Set the filesystem root
         * \param root Shared pointer to new root
         */
        void SetFactory(FSObjectFactory::Uptr Factory);

        /**
         * \brief Creates a Test Filesystem using the Factory.
         * \throw std::invalid_argument if Factory is nullptr.
         */
        void CreateTestFilesystem();

        // delete Copy and Assign Opertor to prevent untestet Behaviour
        void operator=(FileSystem visit) = delete;
        FileSystem(FileSystem& visit) = delete;

private:

        FSObject::Sptr m_Root;
        FSObjectFactory::Uptr m_Factory;
};
#endif
```

6.5 Filesystem.cpp

```cpp
/*****************************************************************//**
 * \file Filesystem.cpp
 * \brief Filesystem class representing the root of a filesystem
 *
 * \author Simon
 * \date   November 2025
 *********************************************************************/

#include "Filesystem.hpp"
#include <stdexcept>
#include <algorithm>

constexpr size_t BLOCKSIZE_SMALL = 2048;
constexpr size_t BLOCKSIZE_MEDIUM = 8192;
constexpr size_t BLOCKSIZE_LARGE = 32768;
constexpr size_t BLOCKSIZE_CUSTOM = 12288;


FileSystem::FileSystem(FSObject::Sptr root)
{
        if (root == nullptr) throw std::invalid_argument(ERROR_NULLPTR);

        m_Root = move(root);
}

void FileSystem::Work(IVisitor& visitor)
{
        if (m_Root == nullptr) throw std::invalid_argument(ERROR_NULLPTR);

        m_Root->Accept(visitor);
}

FSObject::Sptr FileSystem::ReturnRoot()
{
        return move(m_Root);
}

void FileSystem::SetRoot(FSObject::Sptr root)
{
        if (root == nullptr) throw std::invalid_argument(ERROR_NULLPTR);

        m_Root = move(root);
}

void FileSystem::SetFactory(FSObjectFactory::Uptr Factory)
{
        if (Factory == nullptr) throw std::invalid_argument(ERROR_NULLPTR);

        m_Factory = move(Factory);
}

void FileSystem::CreateTestFilesystem()
{
        if (m_Factory == nullptr) throw std::invalid_argument(ERROR_NULLPTR);

        FSObject::Sptr root_folder = m_Factory->CreateFolder("root");
        IFolder::Sptr root_folder_ptr = root_folder->AsFolder();
        FSObject::Sptr sub_folder = m_Factory->CreateFolder("sub");
        IFolder::Sptr sub_folder_ptr = sub_folder->AsFolder();
        FSObject::Sptr sub_sub_folder = m_Factory->CreateFolder("sub");
        IFolder::Sptr sub_sub_folder_ptr = sub_sub_folder->AsFolder();

        sub_folder->SetName("sub_folder");
        sub_sub_folder->SetName("sub_sub_folder");

        root_folder->SetName("root");
        root_folder_ptr->Add(m_Factory->CreateFile("file1.txt", BLOCKSIZE_SMALL));
        root_folder_ptr->Add(m_Factory->CreateFile("file2.txt", BLOCKSIZE_SMALL));
        root_folder_ptr->Add(m_Factory->CreateFile("file3.txt", BLOCKSIZE_SMALL));
        root_folder_ptr->Add(m_Factory->CreateFile("file4.txt", BLOCKSIZE_SMALL));
        root_folder_ptr->Add(sub_folder);
        sub_folder_ptr->Add(m_Factory->CreateFile("file5.txt", BLOCKSIZE_MEDIUM));
```

```
73          sub_folder_ptr->Add(m_Factory->CreateFile("file6.txt", BLOCKSIZE_LARGE));
74          sub_folder_ptr->Add(sub_sub_folder);
75          sub_sub_folder_ptr->Add(m_Factory->CreateFile("file7.txt", BLOCKSIZE_CUSTOM));
76          sub_sub_folder_ptr->Add(m_Factory->CreateLink("LinkToRoot", root_folder));
77
78          m_Root = move(root_folder);
79  }
```

6.6 FSObject.hpp

```
/*****************************************************************//**
 * \file FSObject.hpp
 * \brief Base class for filesystem objects
 *
 * \author Simon
 * \date   November 2025
 *********************************************************************/
#ifndef FS_OBJECT_HPP
#define FS_OBJECT_HPP

#include "Object.h"
#include "IVisitor.hpp"
#include "IFolder.hpp"
#include "ILink.hpp"

#include <memory>
#include <vector>

class FSObject : public Object
{
public:
        // Public Error Messages
        inline static const std::string ERROR_NULLPTR = "ERROR_Nullptr";
    inline static const std::string ERROR_STRING_EMPTY = "ERROR_String_Empty";

        // Smart pointer types
        using Sptr = std::shared_ptr<FSObject>;
        using Uptr = std::unique_ptr<FSObject>;
        using Wptr = std::weak_ptr<FSObject>;

        /** \brief Accept a visitor (pure virtual)
         * \param visit Visitor to accept
         */
        virtual void Accept(IVisitor& visit) =0;

        /** \brief Clones it self as a new
         *  \return Shared pointer to the cloned FSObject
         */
        virtual FSObj_Sptr Clone() const = 0;

        /** \brief Try to "cast" this FSObject to a folder
         * \return Shared pointer to IFolder or nullptr
         */
        virtual IFolder::Sptr AsFolder();

        /** \brief Try to "cast" this FSObject to a folder
         * \return Shared pointer to IFolder or nullptr
         */
        virtual std::shared_ptr<const IFolder> AsFolder() const;

        /** \brief Try to cast this FSObject to a link
         * \return Shared pointer to ILink or nullptr
         */
        virtual std::shared_ptr<const ILink> AsLink() const;

        /** \brief Get the name of the object
         * \return Name as std::string_view
         */
        std::string_view GetName() const;

        /** \brief Set the name of the object
         * \param name New name
         */
        void SetName(std::string_view name);


        /** \brief Get parent as weak pointer
         * \return Weak pointer to parent
         */
        FSObj_Wptr GetParent() const;

        /** \brief Set parent of this FSObject
```

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

6.7 FSObject.cpp

```cpp
/*****************************************************************//**
 * \file FSObject.cpp
 * \brief Base class for filesystem objects
 *
 * \author Simon
 * \date   November 2025
 *********************************************************************/
#include "FSObject.hpp"
#include <string>
#include <stdexcept>

IFolder::Sptr FSObject::AsFolder()
{
    return nullptr;
}

std::shared_ptr<const IFolder> FSObject::AsFolder() const
{
    return nullptr;
}

std::shared_ptr<const ILink> FSObject::AsLink() const
{
        return nullptr;
}

std::string_view FSObject::GetName() const
{
    return std::string_view(m_Name);
}

void FSObject::SetName(std::string_view name)
{
    if (name.empty()) throw std::invalid_argument(ERROR_STRING_EMPTY);
    m_Name = name;
}

void FSObject::SetParent(Sptr parent)
{
        if (parent == nullptr) throw std::invalid_argument(ERROR_NULLPTR);
        m_Parent = move(parent);
}

FSObject::FSObject(std::string_view name)
{
    if (name.empty()) throw std::invalid_argument(ERROR_STRING_EMPTY);
    m_Name = name;
}

FSObj_Wptr FSObject::GetParent() const
{
        return m_Parent;
}
```

6.8 File.hpp

```cpp
/*****************************************************************//**
 * \file File.hpp
 * \brief File class representing a file in the filesystem
 *
 * \author Simon
 * \date   November 2025
 *********************************************************************/
#ifndef FILE_HPP
#define FILE_HPP

#include "FSObject.hpp"

class File : public FSObject, public std::enable_shared_from_this<File>
{
public:
        // Public Error Messages
    inline static const std::string ERROR_NULLPTR = "ERROR Nullptr";
    inline static const std::string ERR_OUT_OF_SPACE = "Not enough space to write data";

        // Smart pointer types
    using Uptr = std::unique_ptr<File>;
    using Sptr = std::shared_ptr<File>;
    using Wptr = std::weak_ptr<File>;

    /** \brief Construct a file
        * \param name File name
        * \param res_blocks Reserved blocks
        * \param blocksize Block size (default4096)
        */
    File(std::string_view name,const size_t res_blocks,const size_t blocksize =4096)
        : m_size(0), m_blocksize(blocksize), FSObject{ name },
        m_res_blocks(res_blocks)
    {}

    /** \brief Accept a visitor
        * \param visit Visitor to accept
        */
    virtual void Accept(IVisitor& visit) override;

    /** \brief Write bytes to the file (increases size)
        * \param bytes Number of bytes to write
        * Call by Value is intentional because it is faster than by reference for built-in types
        */
    void Write(const size_t bytes);

    /** \brief Get current size of the file
        * \return Size in bytes
        */
    size_t GetSize() const;

        /** \brief Clones it self as a new
        *  \return Shared pointer to the cloned FSObject
        */
        virtual FSObj_Sptr Clone() const override;

private:
        size_t m_size;
        const size_t m_blocksize;
        const size_t m_res_blocks;
};
#endif
```

6.9 File.cpp

```cpp
/*****************************************************************//**
 * \file File.cpp
 * \brief File class representing a file in the filesystem
 *
 * \author Simon
 * \date   November 2025
 *********************************************************************/

#include "File.hpp"
#include <stdexcept>
/** \brief Accept a visitor for this file */
void File::Accept(IVisitor& visit)
{
    visit.Visit(move(shared_from_this()));
}

/** \brief Write bytes to the file, throws on out of space */
void File::Write(const size_t bytes)
{
    if ((bytes + m_size) > m_blocksize * m_res_blocks)
        throw std::runtime_error(ERR_OUT_OF_SPACE);

    m_size += bytes;
}

/** \brief Return current size */
size_t File::GetSize() const
{
    return m_size;
}

FSObj_Sptr File::Clone() const
{
        // Call copy constructor
    return std::make_shared<File>(File::File( *this ));
}
```

6.10 IFolder.hpp

```cpp
/*****************************************************************//**
 * \file IFolder.hpp
 * \brief Interface for folder-like FSObjects
 *
 * \author Simon
 * \date   November 2025
 *********************************************************************/
#ifndef IFOLDER_HPP
#define IFOLDER_HPP
#include <memory>

// fwd declaration
class FSObject;

// Type aliases
using FSObj_Sptr = std::shared_ptr<FSObject>;
using FSObj_Wptr = std::weak_ptr<FSObject>;

class IFolder
{
public:

        using Sptr = std::shared_ptr<IFolder>;

        /** \brief Add a child FSObject to the folder
         * \param fsobj Shared pointer to the FSObject to add
         */
        virtual void Add(FSObj_Sptr fsobj) =0;

        /** \brief Get a child by index
         * \param idx Index of the child
         * \return Shared pointer to the child or nullptr if out of range
         */
        virtual FSObj_Sptr GetChild(size_t idx) const =0;

        /** \brief Remove a child FSObject from the folder
         * \param fsobj Shared pointer to the FSObject to remove
         */
        virtual void Remove(FSObj_Sptr fsobj) =0;

        /** \brief Virtual destructor */
        virtual ~IFolder() = default;

private:
};

#endif
```

6.11 Folder.hpp

```
/*****************************************************************//**
 * \file Folder.hpp
 * \brief Folder class representing a folder in the filesystem
 *
 * \author Simon
 * \date   November 2025
 *********************************************************************/
#ifndef FOLDER_HPP
#define FOLDER_HPP

#include "IFolder.hpp"
#include "IVisitor.hpp"
#include "FSObject.hpp"

#include <memory>
#include <vector>

class Folder : public IFolder, public FSObject, public std::enable_shared_from_this<Folder>
{
public:

        // Smart pointer types
        using Uptr = std::unique_ptr<Folder>;
        using Sptr = std::shared_ptr<Folder>;
        using Wptr = std::weak_ptr<Folder>;
        using Cont = std::vector<FSObj_Sptr>;

        /** \brief Construct a folder with a name
         * \param name Name of the folder
         */
        Folder(std::string_view name) : FSObject(name) {}

        /** \brief Add a child FSObject to this folder
         * \param fsobj Shared pointer to the child
         */
        virtual void Add(FSObj_Sptr fsobj);

        /** \brief Get child by index
         * \param idx Index (by value is faster than by reference)
         * \return Shared pointer to child or nullptr
         */
        virtual FSObj_Sptr GetChild(const size_t idx) const override;

        /** \brief Remove a child from the folder
         * \param fsobj Child to remove
         */
        virtual void Remove(FSObj_Sptr fsobj);

        /** \brief Cast this FSObject to a folder interface
         * \return Shared pointer to IFolder
         */
        virtual std::shared_ptr<const IFolder> AsFolder() const override;

        /** \brief Cast this FSObject to a folder interface
         * \return Shared pointer to IFolder
         */
        virtual IFolder::Sptr AsFolder() override;

        /** \brief Accept a visitor and propagate to children
         * \param visit Visitor to accept
         */
        virtual void Accept(IVisitor& visit) override;

        /** \brief Clones it self as a new
        *  \return Shared pointer to the cloned FSObject
        */
        virtual FSObj_Sptr Clone() const override;

        /** \brief Assignment operator for Folder
         * This makes a deep copy of the folder and its children.
         * \param fold Folder to copy from
         */
```

```
73          void operator=(const Folder& fold);
74
75  protected:
76          /**
77           * \brief Copy Constructor of a Folder .
78           * This makes a deep copy of the folder and its children.
79           * This is protected to prevent direct usage, use Clone() instead
80           * This is done because the parent pointer needs to be set correctly, and this
81           * cannot be done in the copy constructor directly.
82           * \param fold
83           */
84          Folder(const Folder& fold);
85
86          // DTOR is defaulted because no special action is needed!
87
88  private:
89          Folder::Cont m_Children;
90  };
91
92  #endif
```

6.12 Folder.cpp

```cpp
/*****************************************************************//**
 * \file Folder.cpp
 * \brief Folder class representing a folder in the filesystem
 *
 * \author Simon
 * \date   November 2025
 *********************************************************************/
#include "Folder.hpp"
#include <stdexcept>
#include <algorithm>


Folder::Folder(const Folder& fold) : FSObject(fold)
{
	m_Children.reserve(fold.m_Children.size());

	for (const auto & child : fold.m_Children)
	{
		// clone each child; do not call Add() because it needs shared_from_this()
		// and we are still in the constructor so shared_from_this() is not available yet.
		m_Children.emplace_back(move(child->Clone()));
	}
}


/** \brief Add child to folder, sets parent pointer on child */
void Folder::Add(FSObj_Sptr fsobj)
{
	if (fsobj == nullptr) throw std::invalid_argument(FSObject::ERROR_NULLPTR);

	fsobj->SetParent(std::move(shared_from_this()));

	m_Children.emplace_back(move(fsobj));
}

/** \brief Get child by index */
FSObj_Sptr Folder::GetChild(const size_t idx) const
{
	if(idx < m_Children.size())
	{
		return m_Children.at(idx);
	}

	return nullptr;
}

/** \brief Remove a child from container */
void Folder::Remove(FSObj_Sptr fsobj)
{
	m_Children.erase(
	std::remove(m_Children.begin(), m_Children.end(), fsobj), m_Children.end()
	);
}

/** \brief Return this as IFolder shared pointer */
std::shared_ptr<const IFolder> Folder::AsFolder() const
{
	return shared_from_this();
}

IFolder::Sptr Folder::AsFolder()
{
	return shared_from_this();
}

/** \brief Accept a visitor and forward to children */
void Folder::Accept(IVisitor& visit)
{
	visit.Visit(move(shared_from_this()));

	for(auto& child : m_Children)
	{
```

```
73                  child->Accept(visit);
74          }
75  }
76
77  FSObj_Sptr Folder::Clone() const
78  {
79          // Create a shared_ptr-owned copy so we can set parent pointers correctly
80          // Use explicit new here so protected copy ctor is accessible in this class context
81          auto newFolder = std::shared_ptr<Folder>(new Folder(*this));
82
83          // Set parent of each cloned child to the new folder
84          for (auto & child : newFolder->m_Children)
85          {
86                  if (child)
87                  {
88                          child->SetParent(newFolder);
89                  }
90          }
91
92          return newFolder;
93  }
94
95
96  void Folder::operator=(const Folder& fold)
97  {
98          // prevent self-assignment
99          if (this != &fold)
100         {
101                 // call base class assignment
102                 FSObject::operator=(fold);
103
104                 // clear current children
105                 m_Children.clear();
106
107                 // deep copy of children
108                 m_Children.reserve(fold.m_Children.size());
109
110                 for (const auto& child : fold.m_Children)
111                 {
112                         Add(move(child->Clone()));
113                 }
114         }
115 }
```

6.13 ILink.hpp

```cpp
/*****************************************************************//**
 * \file ILink.hpp
 * \brief Interface for folder-like FSObjects
 *
 * \author Simon
 * \date   November 2025
 *********************************************************************/
#ifndef ILINK_HPP
#define ILINK_HPP
#include <memory>

 // fwd declaration
class FSObject;

// Type aliases
using FSObj_Sptr = std::shared_ptr<FSObject>;
using FSObj_Wptr = std::weak_ptr<FSObject>;

class ILink
{
public:

        using Sptr = std::shared_ptr<ILink>;

        /** \brief Get the referenced FSObject
         * \return Shared pointer to the referenced FSObject or nullptr if expired
         */
        virtual FSObj_Sptr GetReferncedFSObject() const =0;

        /** \brief Virtual destructor */
        virtual ~ILink() = default;

private:
};

#endif
```

6.14 Link.hpp

```cpp
/*****************************************************************//**
 * \file Link.hpp
 * \brief A link to another FSObject
 *
 * \author Simon
 * \date   November 2025
 *********************************************************************/
#ifndef LINK_HPP
#define LINK_HPP

#include "FSObject.hpp"
#include "IVisitor.hpp"

class Link : public FSObject, public ILink, public std::enable_shared_from_this<Link>
{
public:

        // Public Error Messages
        using Sptr = std::shared_ptr<Link>;
        using Uptr = std::unique_ptr<Link>;
        using Wptr = std::weak_ptr<Link>;

    /** \brief Constructor taking a shared pointer to the linked FSObject
         * \param linked_obj Shared pointer to the referenced FSObject
         * \param name Optional name for the link
         */
        explicit Link(FSObj_Sptr linked_obj, std::string_view name = "");

        /** \brief Cast this object to link interface
         * \return Shared pointer to ILink
         */
        virtual std::shared_ptr<const ILink> AsLink() const override;

    /** \brief Get the referenced FSObject
         * \return Shared pointer to the referenced FSObject or nullptr if expired
         */
        virtual FSObj_Sptr GetReferncedFSObject() const override;

    /** \brief Accept a visitor
         * \param visit Visitor to accept
         */
        virtual void Accept(IVisitor& visit) override;

        /** \brief Clones it self as a new
        *  \return Shared pointer to the cloned FSObject
        */
        virtual FSObj_Sptr Clone() const override;

private:
    /** \brief Weak pointer to the linked FSObject
        */
        FSObj_Wptr m_Ref;
};

#endif
```

6.15 Link.cpp

```cpp
/*****************************************************************//**
 * \file Link.cpp
 * \brief A link to another FSObject
 *
 * \author Simon
 * \date   November 2025
 *********************************************************************/
#include "Link.hpp"
#include <stdexcept>

/** \brief Construct a link to another FSObject */
Link::Link(FSObj_Sptr linked_obj, std::string_view name) : FSObject(name)
{
    if (linked_obj == nullptr) throw std::invalid_argument(Link::ERROR_NULLPTR);
    if (name.empty())          throw std::invalid_argument(Link::ERROR_STRING_EMPTY);

    m_Ref = move(linked_obj);
}

/** \brief Cast to ILink */
std::shared_ptr<const ILink> Link::AsLink() const
{
    return move(shared_from_this());
}

/** \brief Get referenced FSObject (shared_ptr) or nullptr */
FSObj_Sptr Link::GetReferncedFSObject() const
{
    return m_Ref.lock();
}

/** \brief Accept a visitor */
void Link::Accept(IVisitor& visit)
{
    visit.Visit(move(shared_from_this()));
}

FSObj_Sptr Link::Clone() const
{
        // Call Copy Constructor of Link
    return make_shared<Link>(Link::Link(*this));
}
```

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

```cpp
/*****************************************************************//**
 * \file FilterVisitor.hpp
 * \brief Visitor that filters filesystem objects based on criteria defines in derived classes
 *
 * \author Simon
 * \date   November 2025
 *********************************************************************/
#ifndef FILTER_VISITOR_HPP
#define FILTER_VISITOR_HPP

#include "IVisitor.hpp"
#include "FSObject.hpp"

#include <vector>
#include <ostream>

class FilterVisitor : public Object, public IVisitor
{
public:

        // Public Error Messages
        inline static const std::string ERROR_NULLPTR = "ERROR: Nullptr";
        inline static const std::string ERROR_BAD_OSTREAM = "ERROR: bad output stream";

        // constainer Alias for filtered objects (weak pointers)
        using TContFSobj = std::vector<std::weak_ptr<const FSObject>>;

        /** \brief Visit a folder (default no-op)
         * \param folder Folder to visit
         */
        virtual void Visit(const std::shared_ptr<const Folder>folder) override;

        /** \brief Visit a file and apply filter
         * \param file File to visit
         */
        virtual void Visit(const std::shared_ptr<const File>file) override;

        /** \brief Visit a link and apply filter
         * \param link Link to visit
         */
        virtual void Visit(const std::shared_ptr<const Link> link) override;

        /** \brief Dump filtered objects to stream
         * \param ost Output stream
         */
        void DumpFiltered(std::ostream& ost) const;

        /** \brief Get the container of filtered objects (weak pointers)
         * \return Const reference to container
         */
        const TContFSobj & GetFilteredObjects() const;

        // delete Copy and Assign Opertor to prevent untestet Behaviour
        void operator=(FilterVisitor visit) = delete;
        FilterVisitor(FilterVisitor& visit) = delete;

protected:

        /** \brief Check if a file matches the filter
         * \param file File to check
         * \return true if accepted
         */
        virtual bool DoFilter(const std::shared_ptr<const File>& file) const = 0;

        /** \brief Check if a link matches the filter
         * \param link Link to check
         * \return true if accepted
         */
        virtual bool DoFilter(const std::shared_ptr<const Link>& link) const = 0;

        FilterVisitor() = default;

```

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

6.18 FilterVisitor.cpp

```cpp
/*****************************************************************//**
 * \file FilterVisitor.cpp
 * \brief Visitor that filters filesystem objects based on criteria defines in derived classes
 *
 * \author Simon
 * \date   November 2025
 *********************************************************************/
#include "FilterVisitor.hpp"
#include "Folder.hpp"
#include "File.hpp"
#include "Link.hpp"

#include <vector>
#include <iostream>
#include <cassert>
#include <stdexcept>


void FilterVisitor::DumpPath(const std::weak_ptr<const FSObject> & fsobj, std::ostream& ost) const
{
	// end recursion on expired weak pointer
	if (fsobj.expired()) return;

	const auto obj = fsobj.lock();
	if (!obj) return; // defensive: lock could fail

	// first dump parent path
	DumpPath(obj->GetParent(), ost);

	if (!ost.good()) throw std::invalid_argument(FilterVisitor::ERROR_BAD_OSTREAM);

	ost << "\\" << obj->GetName();

	const std::shared_ptr<const ILink> link_ptr = obj->AsLink();

	if (link_ptr) {
		const FSObject::Sptr linked_obj = link_ptr->GetReferncedFSObject();
		if (linked_obj) {
			ost << " -> " << linked_obj->GetName();
		}
		else {
			ost << " -> " << "linked Object Expired!";
		}
	}
}

/** \brief Default visit for folder (no-op) */
void FilterVisitor::Visit(const std::shared_ptr<const Folder> folder)
{
	if (folder == nullptr) throw std::invalid_argument(ERROR_NULLPTR);
}

/** \brief Visit a file and if it matches add to filtered container */
void FilterVisitor::Visit(const std::shared_ptr<const File> file)
{
	if (file == nullptr) throw std::invalid_argument(ERROR_NULLPTR);

	// if file matches filter add to container
	if(DoFilter(file))
	{
		m_FilterCont.emplace_back(file);
	}
}

/** \brief Visit a link and if it matches add to filtered container */
void FilterVisitor::Visit(const std::shared_ptr<const Link> link)
{
	if (link == nullptr) throw std::invalid_argument(ERROR_NULLPTR);

	// if link matches filter add to container
	if (DoFilter(link))
	{
```

```
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

```
/*****************************************************************//**
 * \file FilterFileVisitor.hpp
 * \brief Visitor that filters files by size range
 *
 * \author Simon
 * \date   November 2025
 *********************************************************************/
#ifndef FILTER_FILE_VISITOR_HPP
#define FILTER_FILE_VISITOR_HPP

#include "FilterVisitor.hpp"

class FilterFileVisitor : public FilterVisitor
{
public:
        // Public Error Messages
        inline static const std::string ERROR_INVALID_SIZE_RANGE = "Invalid size range: minimum size must be less than maximum size";

        /** \brief Construct file filter with size range [min,max]
         * \param min Minimum size (inclusive) call by value for built-in type -> is faster than by reference
         * \param max Maximum size (inclusive) call by value for built-in type -> is faster than by reference
         */
        FilterFileVisitor(const size_t min, const size_t max);

        // delete Copy and Assign Opertor to prevent untestet Behaviour
        void operator=(FilterFileVisitor visit) = delete;
        FilterFileVisitor(FilterFileVisitor& visit) = delete;

protected:

        /** \brief Do filter check for files
         * \param file File to check
         * \return true if file size is within range
         */
        virtual bool DoFilter(const std::shared_ptr<const File>& file) const override;

        /** \brief Links are not accepted by this filter
         * \param link Link to check
         * \return false always
         */
        virtual bool DoFilter(const std::shared_ptr<const Link>& link) const override;

private:
        // cannot be const because there are checks in the constructor
        size_t m_MinSize;
        size_t m_MaxSize;
};

#endif
```

6.20 FilterFileVisitor.cpp

```cpp
/*****************************************************************//**
 * \file FilterFileVisitor.cpp
 * \brief Visitor that filters files by size range
 *
 * \author Simon
 * \date   November 2025
 *********************************************************************/
#include "FilterFileVisitor.hpp"
#include "Folder.hpp"
#include "File.hpp"
#include "Link.hpp"

/** \brief Construct filter with size bounds */
FilterFileVisitor::FilterFileVisitor(const size_t min, const size_t max)
{
	if (min >= max) throw std::invalid_argument(ERROR_INVALID_SIZE_RANGE);

	m_MinSize = min;
	m_MaxSize = max;
}

/** \brief Accept files whose size is within range */
bool FilterFileVisitor::DoFilter(const std::shared_ptr<const File>& file) const
{
	if (file == nullptr) throw std::invalid_argument(ERROR_NULLPTR);

	return file->GetSize() >= m_MinSize && file->GetSize() <= m_MaxSize;
}

/** \brief Links are not accepted by file filter */
bool FilterFileVisitor::DoFilter(const std::shared_ptr<const Link>& link) const
{
	if (link == nullptr) throw std::invalid_argument(ERROR_NULLPTR);

	return false;
}
```

6.21 FilterLinkVisitor.hpp

```cpp
/*****************************************************************//**
 * \file   FilterLinkVisitor.hpp
 * \brief  Visitor that filters links in the filesystem
 *
 * \author Simon
 * \date   December 2025
 *********************************************************************/
#ifndef FILTER_LINK_VISITOR_HPP
#define FILTER_LINK_VISITOR_HPP

#include "FilterVisitor.hpp"

class FilterLinkVisitor : public FilterVisitor
{
public:

        FilterLinkVisitor() = default;

        // delete Copy and Assign Opertor to prevent untestet Behaviour
        void operator=(FilterLinkVisitor visit) = delete;
        FilterLinkVisitor(FilterLinkVisitor& visit) = delete;

protected:

        /** \brief Links are accepted by this filter
         * \param file File to check
         * \return false always
         */
        virtual bool DoFilter(const std::shared_ptr<const File>& file) const override;

        /** \brief Links are accepted by this filter
         * \param link Link to check
         * \return true if link is present
         */
        virtual bool DoFilter(const std::shared_ptr<const Link>& link) const override;

private:
};

#endif
```

6.22 FilterLinkVisitor.cpp

```cpp
/*****************************************************************//**
 * \file   FilterLinkVisitor.cpp
 * \brief  Visitor that filters links in the filesystem
 *
 * \author Simon
 * \date   December 2025
 *********************************************************************/
#include "FilterLinkVisitor.hpp"
#include <cassert>
#include <stdexcept>

/** \brief Files are not accepted by link filter */
bool FilterLinkVisitor::DoFilter(const std::shared_ptr<const File>& file) const
{
        if(file == nullptr) throw std::invalid_argument(ERROR_NULLPTR);
        return false;
}

/** \brief Links are accepted by link filter */
bool FilterLinkVisitor::DoFilter(const std::shared_ptr<const Link>& link) const
{
        if(link == nullptr) throw std::invalid_argument(ERROR_NULLPTR);

        return true;
}
```

6.23 DumpVisitor.hpp

```cpp
/*****************************************************************//**
 * \file DumpVisitor.hpp
 * \brief Visitor that dumps filesystem object paths to an output stream
 *
 * \author Simon
 * \date   November 2025
 *********************************************************************/
#ifndef DUMP_VISITOR_HPP
#define DUMP_VISITOR_HPP

#include <iostream>
#include "IVisitor.hpp"
#include "FSObject.hpp"

class DumpVisitor : public Object, public IVisitor
{
public:

        // Public Error Messages
        inline static const std::string ERROR_NULLPTR = "ERROR Nullptr";
        inline static const std::string ERROR_BAD_OSTREAM = "ERROR: bad output stream";

        /** \brief Construct a dumper that writes to given ostream
         * \param ost Output stream reference
         */
        DumpVisitor(std::ostream& ost) : m_ost{ ost } {}

        /** \brief Visit folder
         * \param folder Folder to visit
         */
        virtual void Visit(const std::shared_ptr<const Folder> folder) override;

        /** \brief Visit file
         * \param file File to visit
         */
        virtual void Visit(const std::shared_ptr<const File> file) override;

        /** \brief Visit link
         * \param Link Link to visit
         */
        virtual void Visit(const std::shared_ptr<const Link> Link) override;

        // delete Copy and Assign Opertor to prevent untestet Behaviour
        void operator=(DumpVisitor visit) = delete;
        DumpVisitor(DumpVisitor& visit) = delete;

private:
        /** \brief Dump a single FSObject path to the output stream
         * \param fsobj Shared pointer to object
         */
        void Dump(const std::shared_ptr<const FSObject> fsobj);

        // Output stream reference
        std::ostream & m_ost;
};

#endif
```

6.24 DumpVisitor.cpp

```cpp
/*****************************************************************//**
 * \file DumpVisitor.cpp
 * \brief Visitor that dumps filesystem object paths to an output stream
 *
 * \author Simon
 * \date   November 2025
 *********************************************************************/
#include "DumpVisitor.hpp"
#include "Folder.hpp"
#include "File.hpp"
#include "Link.hpp"

#include <vector>
#include <algorithm>
#include <cassert>



/** \brief Visit folder and dump its path */
void DumpVisitor::Visit(const std::shared_ptr<const Folder> folder)
{
	if (m_ost.fail()) throw std::invalid_argument(ERROR_BAD_OSTREAM);
	if (folder == nullptr) throw std::invalid_argument(ERROR_NULLPTR);

	Dump(folder);
}

/** \brief Visit file and dump its path */
void DumpVisitor::Visit(const std::shared_ptr<const File> file)
{
	if (m_ost.fail()) throw std::invalid_argument(ERROR_BAD_OSTREAM);
	if (file == nullptr) throw std::invalid_argument(ERROR_NULLPTR);

	Dump(file);
}

/** \brief Visit link and dump its path */
void DumpVisitor::Visit(const std::shared_ptr<const Link> Link)
{
	if (m_ost.fail()) throw std::invalid_argument(ERROR_BAD_OSTREAM);
	if (Link == nullptr) throw std::invalid_argument(ERROR_NULLPTR);

	Dump(Link);
}

/** \brief Dump full path for a FSObject to the internal ostream */
void DumpVisitor::Dump(const std::shared_ptr<const FSObject> fsobj)
{
	assert(m_ost.good());
	assert(fsobj != nullptr);

	// Get parent pointer
	FSObject::Sptr parent = fsobj->GetParent().lock();

	// Print an indentation token for each ancestor
	while (parent != nullptr) {
		m_ost << "|␣␣";
		parent = parent->GetParent().lock();
	}

	m_ost << "|---[" << fsobj->GetName();

	if (fsobj->AsFolder()) {
		m_ost << "/]\n";
	}
	else if (fsobj->AsLink()) {
		m_ost << "->]\n";
	}
	else {
		m_ost << "]\n";
	}
}
```

6.25 main.cpp

```cpp
/*****************************************************************//**
 * \file   main.cpp
 * \brief  Testdriver for the filesystem
 *
 * \author Simon
 * \date   December 2025
 *********************************************************************/

#include <iostream>
#include <string>
#include <memory>
#include "FSObject.hpp"
#include "IFolder.hpp"
#include "ILink.hpp"
#include "FSObjectFactory.hpp"
#include "DumpVisitor.hpp"
#include "FilterFileVisitor.hpp"
#include "FilterLinkVisitor.hpp"
#include "Filesystem.hpp"
#include <cassert>
#include <sstream>
#include "Test.hpp"
#include "fstream"
#include "vld.h"

using namespace std;

#define WriteOutputFile ON

static bool TestDumpVisitor(ostream& ost);
static bool TestFilterLinkVisitor(ostream& ost);
static bool TestFilterFileVisitor(ostream& ost);
static bool TestVisitor(ostream& ost,IVisitor & visit);
static bool TestFactory(ostream& ost);
static bool TestLink(ostream& ost);
static bool TestFolder(ostream& ost);
static bool TestFile(ostream& ost);
static bool TestFileSystem(ostream& ost);

int main()
{

    ofstream output{ "Testoutput.txt" };
    if (!output.is_open()) {
        cerr << "Konnte Testoutput.txt nicht oeffnen" << TestCaseFail;
        return 1;
    }

    try {
            DumpVisitor visitor(std::cout);

            FilterLinkVisitor filter_link_visitor;

            FilterFileVisitor filter_file_visitor(4096, 16384);

            FileSystem homework;

            homework.SetFactory(std::make_unique<FSObjectFactory>());
            homework.CreateTestFilesystem();

            homework.Work(visitor);

            std::cout <<"-----------------------------------" << std::endl;
        homework.Work(filter_link_visitor);

            filter_link_visitor.DumpFiltered(std::cout);

            std::cout << "-----------------------------------" << std::endl;

        homework.Work(filter_file_visitor);

            filter_file_visitor.DumpFiltered(std::cout);
```

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

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

```
222     return TestOK;
223 }
224
225 bool TestFilterLinkVisitor(ostream& ost)
226 {
227     assert(ost.good());
228
229     ost << TestStart;
230
231     bool TestOK = true;
232     string error_msg;
233
234
235     try {
236         FSObjectFactory factory;
237         FSObject::Sptr root_folder = factory.CreateFolder("root");
238         FSObject::Sptr sub_folder = factory.CreateFolder("sub_folder");
239         FSObject::Sptr sub_sub_folder = factory.CreateFolder("sub_sub_folder");
240         File::Sptr file = make_shared<File>("file1.txt", 2048);
241         Link::Sptr link = make_shared<Link>(file,"LinkToFile1");
242         sub_sub_folder->AsFolder()->Add(file );
243         sub_sub_folder->AsFolder()->Add(link);
244         sub_folder->AsFolder()->Add(sub_sub_folder);
245         root_folder->AsFolder()->Add(sub_folder);
246
247                 FilterLinkVisitor link_filter;
248
249                 root_folder->Accept(link_filter);
250
251                 TestOK = TestOK && check_dump(ost, "FilterLinkVisitor Test filtered amount",
                          static_cast<size_t>(1), link_filter.GetFilteredObjects().size());
252                 TestOK = TestOK && check_dump(ost, "FilterLinkVisitor Test filtered obj", link->
                          GetReferncedFSObject()->GetName(), link_filter.GetFilteredObjects().cbegin()->lock
                          ()->AsLink()->GetReferncedFSObject()->GetName());
253
254         stringstream result;
255                 stringstream expected;
256
257                 link_filter.DumpFiltered(result);
258
259                 expected << "\\root\\sub_folder\\sub_sub_folder\\LinkToFile1 -> file1.txt" << std::endl
                          ;
260
261                 TestOK = TestOK && check_dump(ost, "Filter Link Visitor Test Dump ", expected.str(),
                          result.str());
262
263     }
264     catch (const string& err) {
265         error_msg = err;
266     }
267     catch (bad_alloc const& error) {
268         error_msg = error.what();
269     }
270     catch (const exception& err) {
271         error_msg = err.what();
272     }
273     catch (...) {
274         error_msg = "Unhandelt Exception";
275     }
276
277     TestOK = TestOK && check_dump(ost, "Test for Exception in Testcase", true, error_msg.empty());
278     error_msg.clear();
279
280     try {
281
282         FilterLinkVisitor link_filter{};
283
284         stringstream result;
285                 result.setstate(ios::badbit);
286
287                 link_filter.DumpFiltered(result);
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     }
301 
302     TestOK = TestOK && check_dump(ost, "Test for Exception in Dump with bad Ostream", error_msg,
          FilterLinkVisitor::ERROR_BAD_OSTREAM);
303     error_msg.clear();
304 
305 
306     ost << TestEnd;
307 
308     return TestOK;
309 }
310 
311 bool TestFilterFileVisitor(ostream& ost)
312 {
313     assert(ost.good());
314 
315     ost << TestStart;
316 
317     bool TestOK = true;
318     string error_msg;
319 
320 
321     try {
322         FSObjectFactory factory;
323         FSObject::Sptr root_folder = factory.CreateFolder("root");
324         FSObject::Sptr sub_folder = factory.CreateFolder("sub_folder");
325         FSObject::Sptr sub_sub_folder = factory.CreateFolder("sub_sub_folder");
326         File::Sptr file = make_shared<File>("file1.txt", 10);
327         File::Sptr file1 = make_shared<File>("file2.txt", 10);
328         File::Sptr file2 = make_shared<File>("file3.txt", 10);
329         File::Sptr file3 = make_shared<File>("file4.txt", 10);
330         Link::Sptr link = make_shared<Link>(file, "LinkToFile1");
331 
332                 file->Write(8192);
333                 file1->Write(4096);
334                 file2->Write(6000);
335                 file3->Write(1000);
336 
337         sub_sub_folder->AsFolder()->Add(file);
338         root_folder->AsFolder()->Add(file2);
339         sub_sub_folder->AsFolder()->Add(link);
340         sub_folder->AsFolder()->Add(sub_sub_folder);
341         sub_folder->AsFolder()->Add(file3);
342         root_folder->AsFolder()->Add(sub_folder);
343         root_folder->AsFolder()->Add(file1);
344 
345         FilterFileVisitor file_filter(5000,9000);
346 
347         root_folder->Accept(file_filter);
348 
349         TestOK = TestOK && check_dump(ost, "FilterFileVisitor Test filtered amount", static_cast<size_t
              >(2), file_filter.GetFilteredObjects().size());
350         TestOK = TestOK && check_dump(ost, "FilterFileVisitor Test for filtered file", file2->GetName()
              , file_filter.GetFilteredObjects().cbegin()->lock()->GetName());
351         TestOK = TestOK && check_dump(ost, "FilterFileVisitor Test for filtered file", file->GetName(),
               file_filter.GetFilteredObjects().crbegin()->lock()->GetName());
352 
353         stringstream result;
354         stringstream expected;
355 
356         file_filter.DumpFiltered(result);
357 
358         expected << "\\root\\file3.txt" << std::endl
359                  << "\\root\\sub_folder\\sub_sub_folder\\file1.txt" << std::endl;
360 
361         TestOK = TestOK && check_dump(ost, "Filter File Visitor Test Dump ", expected.str(), result.str
              ());
```

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

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

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

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

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

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

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

```
866     error_msg.clear();
867
868     //Link SetName with empty string (should throw)
869     try
870     {
871         File::Sptr file = make_shared<File>("file.txt", 2048);
872         Link::Sptr link = make_shared<Link>(file, "OriginalName");
873         link->SetName(""); // should throw
874     }
875     catch (const exception& err) {
876         error_msg = err.what();
877     }
878     TestOK = TestOK && check_dump(ost, "Test Link SetName empty string",
879         FSObject::ERROR_STRING_EMPTY, error_msg);
880     error_msg.clear();
881
882     // Link Accept visitor
883     try
884     {
885         File::Sptr file = make_shared<File>("file.txt", 2048);
886         Link::Sptr link = make_shared<Link>(file, "TestLink");
887         stringstream result;
888         DumpVisitor visitor(result);
889
890         link->Accept(visitor);
891         TestOK = TestOK && check_dump(ost, "Test Link Accept visitor - not empty",
892             false, result.str().empty());
893     }
894     catch (const exception& err) {
895         error_msg = err.what();
896     }
897     TestOK = TestOK && check_dump(ost, "Test Link Accept - error buffer", true, error_msg.empty());
898     error_msg.clear();
899
900     ost << TestEnd;
901     return TestOK;
902 }
903 bool TestFolder(ostream& ost)
904 {
905     assert(ost.good());
906
907     ost << TestStart;
908
909     bool TestOK = true;
910     string error_msg;
911
912     // test folder as intended
913     try
914     {
915         string_view folder_name = "MyFolder";
916         Folder::Sptr folder = make_shared<Folder>(folder_name);
917         TestOK = TestOK && check_dump(ost, "Test normal CTOR Folder", folder_name, folder->GetName());
918
919         File::Sptr file1 = make_shared<File>("file1.txt", 2048);
920         File::Sptr file2 = make_shared<File>("file2.txt", 4096);
921
922         folder->Add(file1);
923         folder->Add(file2);
924
925         FSObject::Sptr err_file = folder->GetChild(2); // <= should be nullptr
926         FSObject::Sptr shared_null = nullptr;
927
928         TestOK = TestOK && check_dump(ost, "Get Child from folder", static_pointer_cast<FSObject>(file1
                ), folder->GetChild(0));
929         TestOK = TestOK && check_dump(ost, "Get next Child from folder", static_pointer_cast<FSObject>(
                file2), folder->GetChild(1));
930         TestOK = TestOK && check_dump(ost, "Get Child for invalid index", err_file, shared_null);
931     }
932     catch (const string& err) {
933         error_msg = err;
934     }
935     catch (bad_alloc const& error) {
936         error_msg = error.what();
937     }
938     catch (const exception& err) {
```

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

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

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

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

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

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

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

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

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

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

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

6.26 Test.hpp

```
/*****************************************************************//**
 * \file   Test.hpp
 * \brief  File that provides a Test Function with a formated output
 *
 * \author Simon
 * \date   April 2025
 *********************************************************************/
#ifndef TEST_HPP
#define TEST_HPP

#include <string>
#include <iostream>
#include <vector>
#include <list>
#include <queue>
#include <forward_list>

#define ON 1
#define OFF 0
#define COLOR_OUTPUT OFF

// Definitions of colors in order to change the color of the output stream.
inline const char* colorRed() { return "\x1B[31m"; }
inline const char* colorGreen() { return "\x1B[32m"; }
inline const char* colorWhite() { return "\x1B[37m"; }

inline std::ostream& RED(std::ostream& ost) {
        if (ost.good()) {
                ost << colorRed();
        }
        return ost;
}
inline std::ostream& GREEN(std::ostream& ost) {
        if (ost.good()) {
                ost << colorGreen();
        }
        return ost;
}
inline std::ostream& WHITE(std::ostream& ost) {
        if (ost.good()) {
                ost << colorWhite();
        }
        return ost;
}

inline std::ostream& TestStart(std::ostream& ost) {
        if (ost.good()) {
                ost << std::endl;
                ost << "****************************************" << std::endl;
                ost << "          TESTCASE START          " << std::endl;
                ost << "****************************************" << std::endl;
                ost << std::endl;
        }
        return ost;
}

inline std::ostream& TestEnd(std::ostream& ost) {
        if (ost.good()) {
                ost << std::endl;
                ost << "****************************************" << std::endl;
                ost << std::endl;
        }
        return ost;
}

inline std::ostream& TestCaseOK(std::ostream& ost) {

#if COLOR_OUTPUT
        if (ost.good()) {
                ost << colorGreen() << "TEST OK!!" << colorWhite() << std::endl;
        }
#else
```

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

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