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!

OBERÖSTERREICH HSD

FH-HAGENBERG

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

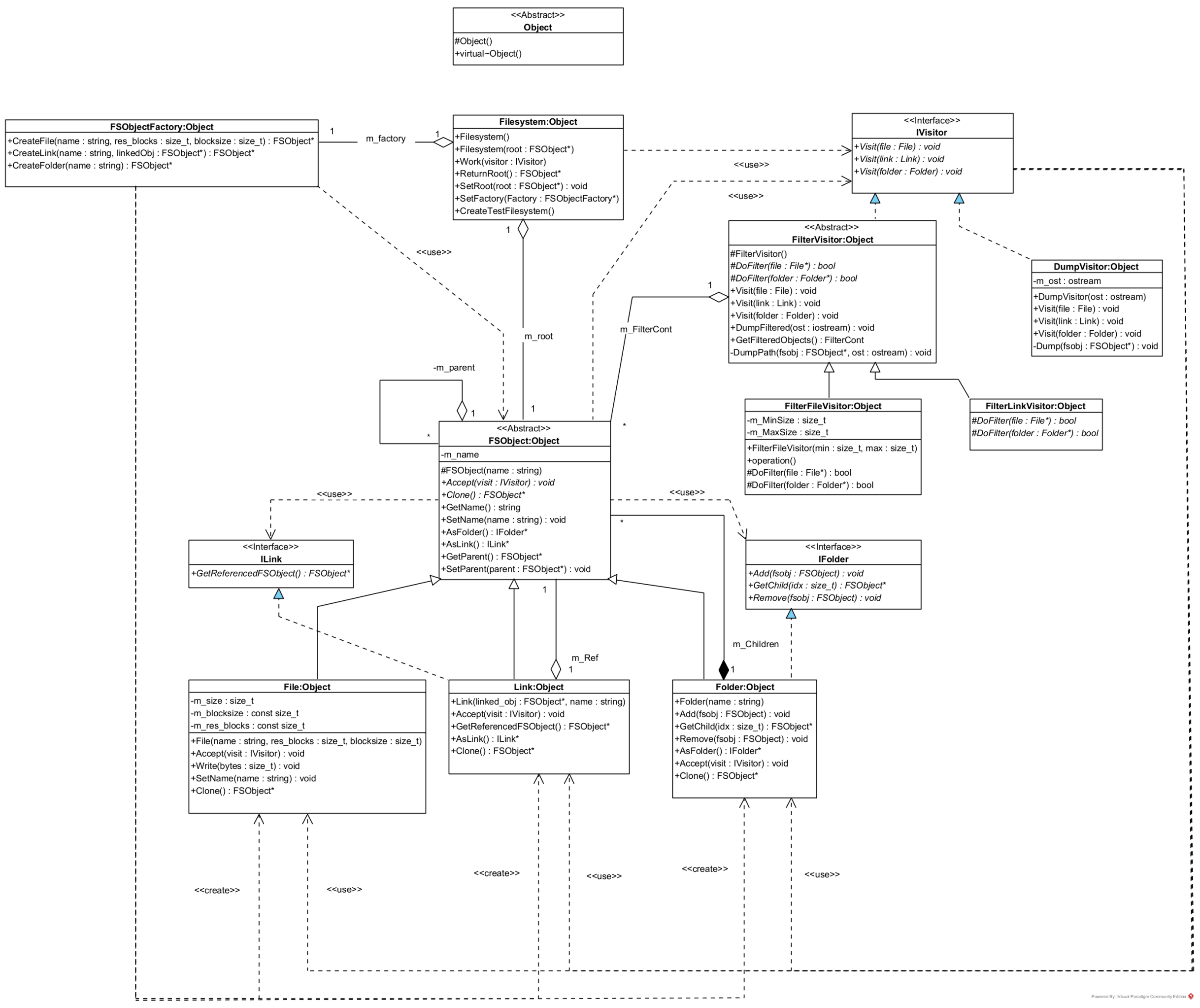
- Gibt die gesamte Dateisystemhierarchie aus.
- Ausgabe sowohl auf der Standardausgabe als auch in einer Datei möglich.
- Muss Dateien, Ordner und Verweise in strukturierter Form darstellen.

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



<<Abstract>>
Object
#Object()
+virtual~Object()
FSObjectFactory:Object
+CreateFile(name : string, res_blocks : size_t, blocksize : size_t) : FSObject*
+CreateLink(name : string, linkedObj : FSObject*) : FSObject*
+CreateFolder(name : string) : FSObject*
1
m_factory
1
Filesystem:Object
+Filesystem()
+Filesystem(root : FSObject*)
+Work(visitor : IVisitor)
+ReturnRoot() : FSObject*
+SetRoot(root : FSObject*) : void
+SetFactory(Factory : FSObjectFactory*)
+CreateTestFilesystem()
<<Interface>>
IVisitor
+Visit(file : File) : void
+Visit(link : Link) : void
+Visit(folder : Folder) : void
<<use>>
<<use>>
<<use>>
1
m_root
<<Abstract>>
FilterVisitor:Object
#FilterVisitor()
#DoFilter(file : File*) : bool
#DoFilter(folder : Folder*) : bool
+Visit(file : File) : void
+Visit(link : Link) : void
+Visit(folder : Folder) : void
+DumpFiltered(ost : iostream) : void
+GetFilteredObjects() : FilterCont
-DumpPath(fsobj : FSObject*, ost : ostream) : void
1
m_FilterCont
DumpVisitor:Object
-m_ost : ostream
+DumpVisitor(ost : ostream)
+Visit(file : File) : void
+Visit(link : Link) : void
+Visit(folder : Folder) : void
-Dump(fsobj : FSObject*) : void
-m_parent
1
1
*
*
FilterFileVisitor:Object
-m_MinSize : size_t
-m_MaxSize : size_t
+FilterFileVisitor(min : size_t, max : size_t)
+operation()
#DoFilter(file : File*) : bool
#DoFilter(folder : Folder*) : bool
FilterLinkVisitor:Object
#DoFilter(file : File*) : bool
#DoFilter(folder : Folder*) : bool
<<Abstract>>
FSObject:Object
-m_name
#FSObject(name : string)
+Accept(visit : IVisitor) : void
+Clone() : FSObject*
+GetName() : string
+SetName(name : string) : void
+AsFolder() : IFolder*
+AsLink() : ILink*
+GetParent() : FSObject*
+SetParent(parent : FSObject*) : void
<<use>>
<<use>>
*
<<Interface>>
ILink
+GetReferencedFSObject() : FSObject*
<<Interface>>
IFolder
+Add(fsobj : FSObject) : void
+GetChild(idx : size_t) : FSObject*
+Remove(fsobj : FSObject) : void
1
m_Ref
1
m_Children
1
File:Object
-m_size : size_t
-m_blocksize : const size_t
-m_res_blocks : const size_t
+File(name : string, res_blocks : size_t, blocksize : size_t)
+Accept(visit : IVisitor) : void
+Write(bytes : size_t) : void
+SetName(name : string) : void
+Clone() : FSObject*
Link:Object
+Link(linked_obj : FSObject*, name : string)
+Accept(visit : IVisitor) : void
+GetReferencedFSObject() : FSObject*
+AsLink() : ILink*
+Clone() : FSObject*
Folder:Object
+Folder(name : string)
+Add(fsobj : FSObject) : void
+GetChild(idx : size_t) : FSObject*
+Remove(fsobj : FSObject) : void
+AsFolder() : IFolder*
+Accept(visit : IVisitor) : void
+Clone() : FSObject*
<<create>>
<<use>>
<<create>>
<<use>>
<<create>>
<<use>>
Powered By: Visual Paradigm Community Edition

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

3.6 Template Methode Pattern

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

4 Dokumentation der Komponenten (Klassen)

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

5 Testprotokollierung

```

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

DumpVisitor Test
[Test OK] Result: (Expected: |---[root/]
|   |---[sub_folder/]
|   |   |---[sub_sub_folder/]
|   |   |   |---[file1.txt]
 == Result: |---[root/]
|   |---[sub_folder/]
|   |   |---[sub_sub_folder/]
|   |   |   |---[file1.txt]
)

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

Test Exception Bad Ostream in DumpVisitor
[Test OK] Result: (Expected: ERROR: bad output stream ==
    ↪ Result: ERROR: bad output stream)


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


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

Test Exception nullptr in Visit File
[Test OK] Result: (Expected: ERROR Nullptr == Result: ERROR
    ↪ Nullptr)

Test Exception nullptr in Visit Folder
[Test OK] Result: (Expected: ERROR Nullptr == Result: ERROR
    ↪ Nullptr)

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

```


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


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

Test Exception nullptr in Visit File
[Test OK] Result: (Expected: ERROR Nullptr == Result: ERROR
   ↪ Nullptr)

Test Exception nullptr in Visit Folder
[Test OK] Result: (Expected: ERROR Nullptr == Result: ERROR
   ↪ Nullptr)

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


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


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

Test Exception nullptr in Visit File
[Test OK] Result: (Expected: ERROR Nullptr == Result: ERROR
   ↪ Nullptr)

Test Exception nullptr in Visit Folder
[Test OK] Result: (Expected: ERROR Nullptr == Result: ERROR
   ↪ Nullptr)

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


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

```

```

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

FilterLinkVisitor Test filtered amount
[Test OK] Result: (Expected: 1 == Result: 1)

FilterLinkVisitor Test filtered obj
[Test OK] Result: (Expected: file1.txt == Result: file1.txt)

Filter Link Visitor Test Dump
[Test OK] Result: (Expected: \root\sub_folder\sub_sub_folder\LinkToFile1 -> file1.txt
 == Result: \root\sub_folder\sub_sub_folder\LinkToFile1 -> 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)


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


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

FilterFileVisitor Test filtered amount
[Test OK] Result: (Expected: 2 == Result: 2)

FilterFileVisitor Test for filtered file
[Test OK] Result: (Expected: file3.txt == Result: file3.txt)

FilterFileVisitor Test for filtered file
[Test OK] Result: (Expected: file1.txt == Result: file1.txt)

Filter File Visitor Test Dump
[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)


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

```

```

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

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

Test normal CTOR Link
[Test OK] Result: (Expected: LinkToMyFolder == Result: LinkToMyFolder)

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

Test Copy CTOR of Link
[Test OK] Result: (Expected: 0000026FB3EB1148 == Result: 0000026FB3EB1148)

Test for shallow Copy of Link
[Test OK] Result: (Expected: 0000026FB3EB1148 == Result: 0000026FB3EB1148)

Test for parent of Copied Link
[Test OK] Result: (Expected: Modified == Result: Modified)

Test normal COPY CTOR Link - error buffer
[Test OK] Result: (Expected: true == Result: true)

Test Assign Op of Link
[Test OK] Result: (Expected: 0000026FB3EB1148 == Result: 0000026FB3EB1148)

Test Assign Op for Parent of Link
[Test OK] Result: (Expected: Modified == Result: Modified)

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

Test Self Assing Op Link - error buffer
[Test OK] Result: (Expected: true == Result: true)

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: 0000026FB3EC69E0 == Result:
    ↪ 0000026FB3EC69E0)

Get next Child from folder
[Test OK] Result: (Expected: 0000026FB3EC75E0 == Result:
    ↪ 0000026FB3EC75E0)

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)
```

```

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

Test Parent of Copied Folder
[Test OK] Result: (Expected: 0000026FB3EC6F18 == Result:
    ↪ 0000026FB3EC6F18)

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

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

Test Assign Op Folder Parent - Child 0
[Test OK] Result: (Expected: MyFolder == Result: MyFolder)

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

Test Self Assign Folder - Child 0
[Test OK] Result: (Expected: 0000026FB3EC6CE0 == Result:
    ↪ 0000026FB3EC6CE0)

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

Test Remove Child from Folder
[Test OK] Result: (Expected: 0000026FB3EC6FE0 == Result:
    ↪ 0000026FB3EC6FE0)

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

Test Folder - add nullptr
[Test OK] Result: (Expected: ERROR Nullptr == Result: ERROR
    ↪ Nullptr)

Test Folder - CTOR with empty string
[Test OK] Result: (Expected: ERROR String Empty == Result:
    ↪ ERROR String Empty)

Test nested folders - root has sub1
```

```
[Test OK] Result: (Expected: 0000026FB3EBB060 == Result:
    ↪ 0000026FB3EBB060)

Test nested folders - sub1 has sub2
[Test OK] Result: (Expected: 0000026FB3EBB130 == Result:
    ↪ 0000026FB3EBB130)

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

Test parent pointer set on Add
[Test OK] Result: (Expected: parent == Result: parent)

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

Test remove non-existent child
[Test OK] Result: (Expected: 0000026FB3EC6CE0 == Result:
    ↪ 0000026FB3EC6CE0)

Test remove non-existent - error buffer
[Test OK] Result: (Expected: true == Result: true)

Test mixed children - file
[Test OK] Result: (Expected: 0000026FB3EC76A0 == Result:
    ↪ 0000026FB3EC76A0)

Test mixed children - folder
[Test OK] Result: (Expected: 0000026FB3EBB068 == Result:
    ↪ 0000026FB3EBB068)

Test mixed children - link
[Test OK] Result: (Expected: 0000026FB3EC7160 == Result:
    ↪ 0000026FB3EC7160)

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

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

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

```

```
Test Accept visits children
[Test OK] Result: (Expected: true == Result: true)

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

Test Folder SetName
[Test OK] Result: (Expected: renamed == Result: renamed)

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


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


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

Test normal CTOR File
[Test OK] Result: (Expected: file1.txt == Result: file1.txt)

Test normal CTOR File - size
[Test OK] Result: (Expected: 0 == Result: 0)

Test normal - write file size
[Test OK] Result: (Expected: 4096 == Result: 4096)

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

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

Test Copy Ctor Parent of file
[Test OK] Result: (Expected: ParentFolder == Result:
   ↪ ParentFolder)

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

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

```
/******************************************************************//**
 * \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

```
/*****************************************************************//**
 * \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

```
/*******************************************************************//**
 * \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

```
/*****************************************************************//**
 * \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));
        sub_folder_ptr->Add(m_Factory->CreateFile("file6.txt", BLOCKSIZE_LARGE));
        sub_folder_ptr->Add(sub_sub_folder);
        sub_sub_folder_ptr->Add(m_Factory->CreateFile("file7.txt", BLOCKSIZE_CUSTOM));
        sub_sub_folder_ptr->Add(m_Factory->CreateLink("LinkToRoot", root_folder));

        m_Root = move(root_folder);
}
```

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
```

```
	 * \param parent Shared pointer to parent FSObject
	 */
	void SetParent(Sptr parent);

protected:
	/** \brief Construct an FSObject with optional name
	 * \param name Name of the FSObject
	 */
	FSObject(std::string_view name = "");


private:
	std::string m_Name;
	FSObj_Wptr m_Parent;
};

#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::shared_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

```
/*****************************************************************//**
 * \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
{
    return std::make_shared<File>(File::File( *this ));
}
```

6.10 IFolder.hpp

```
/*******************************************************************//**
 * \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

```cpp
/******************************************************************//**
 * \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
         */
        void operator=(const Folder& fold);

protected:
        /**
         * \brief Copy Constructor of a Folder .
         * This makes a deep copy of the folder and its children.
         * This is protected to prevent direct usage, use Clone() instead
         * \param fold
         */
        Folder(const Folder& fold);

        // DTOR is defaulted because no special action is needed!

private:
        Folder::Cont m_Children;
};

#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(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)
        {
                child->Accept(visit);
        }
}

FSObj_Sptr Folder::Clone() const
{
        // Create a shared_ptr-owned copy so we can set parent pointers correctly
        // Use explicit new here so protected copy ctor is accessible in this class context
        auto newFolder = std::shared_ptr<Folder>(new Folder(*this));

        // Set parent of each cloned child to the new folder
        for (auto & child : newFolder->m_Children)
        {
                if (child)
                {
                        child->SetParent(newFolder);
                }
        }

        return newFolder;
}


void Folder::operator=(const Folder& fold)
{
        // prevent self-assignment
        if (this != &fold)
        {
                // call base class assignment
                FSObject::operator=(fold);

                // clear current children
                m_Children.clear();

                // deep copy of children
                m_Children.reserve(fold.m_Children.size());

                for (const auto& child : fold.m_Children)
                {
                        Add(child->Clone());
                }
        }
}
```

6.13 ILink.hpp

```
/******************************************************************//**
 * \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

```
/*****************************************************************//**
 * \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

```
/*****************************************************************//**
 * \file  IVisitor.hpp
 * \brief  Interface for visitor pattern in filesystem objects
 *
 * \author Simon
 * \date   November 2025
 *********************************************************************/
#ifndef IVISITOR_HPP
#define IVISITOR_HPP

 // Forward declarations to avoid circular dependencies
class Folder;
class File;
class Link;

#include <memory>

class IVisitor
{
public:

        /** \brief Visit a folder
         * \param folder Shared pointer to the folder to visit
         */
        virtual void Visit(const std::shared_ptr<const Folder> folder)=0;

        /** \brief Visit a file
         * \param file Shared pointer to the file to visit
         */
        virtual void Visit(const std::shared_ptr<const File> file)=0;

        /** \brief Visit a link
         * \param link Shared pointer to the link to visit
         */
        virtual void Visit(const std::shared_ptr<const Link> link)=0;

        /** \brief Virtual destructor for visitor implementations */
        virtual ~IVisitor() = default;

private:
};

#endif
```

6.17 FilterVisitor.hpp

```
/*********************************************************************//**
 * \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;

private:

        /** \brief Dump a single FSObject path to the output stream
         * \param fsobj Weak pointer to object
         * \param ost Output stream
         */
        void DumpPath(const std::weak_ptr<const FSObject> & fsobj, std::ostream& ost) const;

        TContFSobj m_FilterCont;
};

#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))
	{
```

```
                m_FilterCont.emplace_back(link);
        }
}

/** \brief Dump all filtered objects to given ostream */
void FilterVisitor::DumpFiltered(std::ostream& ost) const
{
        if (!ost.good()) throw std::invalid_argument(FilterVisitor::ERROR_BAD_OSTREAM);

        for (const auto & obj : m_FilterCont) {
                DumpPath(obj, ost);
                ost << '\n';
        }
}

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

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

```
/*********************************************************************//**
 * \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

```
/*******************************************************************//**
 * \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

```
/*******************************************************************//**
 * \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);


        bool TestOK = true;

        DumpVisitor dumper{ cout };
        FilterLinkVisitor filter_link;
        FilterFileVisitor filter_file(0, 1024);

        TestOK = TestOK && TestDumpVisitor(cout);
        TestOK = TestOK && TestVisitor(cout, dumper);
        TestOK = TestOK && TestVisitor(cout, filter_link);
        TestOK = TestOK && TestVisitor(cout, filter_file);
        TestOK = TestOK && TestFilterLinkVisitor(cout);
        TestOK = TestOK && TestFilterFileVisitor(cout);
        TestOK = TestOK && TestFactory(cout);
        TestOK = TestOK && TestLink(cout);
        TestOK = TestOK && TestFolder(cout);
        TestOK = TestOK && TestFile(cout);
        TestOK = TestOK && TestFileSystem(cout);

        if (WriteOutputFile) {

            TestOK = TestOK && TestDumpVisitor(output);
            TestOK = TestOK && TestVisitor(output, dumper);
            TestOK = TestOK && TestVisitor(output, filter_link);
            TestOK = TestOK && TestVisitor(output, filter_file);
            TestOK = TestOK && TestFilterLinkVisitor(output);
            TestOK = TestOK && TestFilterFileVisitor(output);
            TestOK = TestOK && TestFactory(output);
            TestOK = TestOK && TestLink(output);
            TestOK = TestOK && TestFolder(output);
            TestOK = TestOK && TestFile(output);
            TestOK = TestOK && TestFileSystem(output);

            if (TestOK) {
                output << TestCaseOK;
            }
            else {
                output << TestCaseFail;
            }

            output.close();
        }

        if (TestOK) {
            cout << TestCaseOK;
        }
        else {
            cout << TestCaseFail;
        }
    }
    catch (const string& err) {
        cerr << err << TestCaseFail;
    }
    catch (bad_alloc const& error) {
        cerr << error.what() << TestCaseFail;
    }
    catch (const exception& err) {
        cerr << err.what() << TestCaseFail;
    }
    catch (...) {
        cerr << "Unhandelt␣Exception" << TestCaseFail;
    }

    if (output.is_open()) output.close();

        return 0;
};

bool TestDumpVisitor(ostream & ost)
{
    assert(ost.good());
    ost << TestStart;

    bool TestOK = true;
    string error_msg;

    try {
                FSObjectFactory factory;
                FSObject::Sptr root_folder = factory.CreateFolder("root");
                FSObject::Sptr sub_folder = factory.CreateFolder("sub_folder");
                FSObject::Sptr sub_sub_folder = factory.CreateFolder("sub_sub_folder");
                sub_sub_folder->AsFolder()->Add(File::Sptr{make_shared<File>("file1.txt", 2048)});
                sub_folder->AsFolder()->Add(sub_sub_folder);
                root_folder->AsFolder()->Add(sub_folder);

                stringstream result;
                stringstream expected;

                DumpVisitor dumper(result);

                root_folder->Accept(dumper);

                expected  << "|---[root/]\n"
                  << "|␣␣|---[sub_folder/]\n"
                  << "|␣␣|␣␣|---[sub_sub_folder/]\n"
                  << "|␣␣|␣␣|␣␣|---[file1.txt]\n";

                TestOK = TestOK && check_dump(ost, "DumpVisitor␣Test", expected.str(), result.str());

    }
    catch (const string& err) {
        error_msg = err;
    }
    catch (bad_alloc const& error) {
        error_msg = error.what();
    }
    catch (const exception& err) {
        error_msg = err.what();
    }
    catch (...) {
        error_msg = "Unhandelt␣Exception";
    }

    TestOK = TestOK && check_dump(ost, "Test␣Exception␣in␣TestCase", true, error_msg.empty());
    error_msg.clear();

    try {

                FSObjectFactory factory;
                FSObject::Sptr root_folder = factory.CreateFolder("root");

                stringstream result;

                result.setstate(ios::badbit);

                DumpVisitor dumper(result);

                root_folder->Accept(dumper); // <= sould throw Exception bad Ostream

    }
    catch (const string& err) {
        error_msg = err;
    }
    catch (bad_alloc const& error) {
        error_msg = error.what();
    }
    catch (const exception& err) {
        error_msg = err.what();
    }
    catch (...) {
        error_msg = "Unhandelt␣Exception";
    }

    TestOK = TestOK && check_dump(ost, "Test␣Exception␣Bad␣Ostream␣in␣DumpVisitor", DumpVisitor::
         ERROR_BAD_OSTREAM, error_msg);
    error_msg.clear();

    ost << TestEnd;

    return TestOK;
}

bool TestFilterLinkVisitor(ostream& ost)
{
    assert(ost.good());

    ost << TestStart;

    bool TestOK = true;
    string error_msg;


    try {
        FSObjectFactory factory;
        FSObject::Sptr root_folder = factory.CreateFolder("root");
        FSObject::Sptr sub_folder = factory.CreateFolder("sub_folder");
        FSObject::Sptr sub_sub_folder = factory.CreateFolder("sub_sub_folder");
        File::Sptr file = make_shared<File>("file1.txt", 2048);
        Link::Sptr link = make_shared<Link>(file,"LinkToFile1");
        sub_sub_folder->AsFolder()->Add(file );
        sub_sub_folder->AsFolder()->Add(link);
        sub_folder->AsFolder()->Add(sub_sub_folder);
        root_folder->AsFolder()->Add(sub_folder);

                FilterLinkVisitor link_filter;

                root_folder->Accept(link_filter);

                TestOK = TestOK && check_dump(ost, "FilterLinkVisitor␣Test␣filtered␣amount",
                     static_cast<size_t>(1), link_filter.GetFilteredObjects().size());
                TestOK = TestOK && check_dump(ost, "FilterLinkVisitor␣Test␣filtered␣obj", link->
                     GetReferncedFSObject()->GetName(), link_filter.GetFilteredObjects().cbegin()->lock
                     ()->AsLink()->GetReferncedFSObject()->GetName());

        stringstream result;
                stringstream expected;

                link_filter.DumpFiltered(result);

                expected << "\\root\\sub_folder\\sub_sub_folder\\LinkToFile1␣->␣file1.txt" << std::endl
                     ;

                TestOK = TestOK && check_dump(ost, "Filter␣Link␣Visitor␣Test␣Dump␣", expected.str(),
                     result.str());

    }
    catch (const string& err) {
        error_msg = err;
    }
    catch (bad_alloc const& error) {
        error_msg = error.what();
    }
    catch (const exception& err) {
        error_msg = err.what();
    }
    catch (...) {
        error_msg = "Unhandelt␣Exception";
    }

    TestOK = TestOK && check_dump(ost, "Test␣for␣Exception␣in␣Testcase", true, error_msg.empty());
    error_msg.clear();

    try {

        FilterLinkVisitor link_filter{};

        stringstream result;
                result.setstate(ios::badbit);

                link_filter.DumpFiltered(result);
    }
    catch (const string& err) {
        error_msg = err;
    }
    catch (bad_alloc const& error) {
        error_msg = error.what();
    }
    catch (const exception& err) {
        error_msg = err.what();
    }
    catch (...) {
        error_msg = "Unhandelt␣Exception";
    }

    TestOK = TestOK && check_dump(ost, "Test␣for␣Exception␣in␣Dump␣with␣bad␣Ostream", error_msg,
         FilterLinkVisitor::ERROR_BAD_OSTREAM);
    error_msg.clear();


    ost << TestEnd;

    return TestOK;
}

bool TestFilterFileVisitor(ostream& ost)
{
    assert(ost.good());

    ost << TestStart;

    bool TestOK = true;
    string error_msg;


    try {
        FSObjectFactory factory;
        FSObject::Sptr root_folder = factory.CreateFolder("root");
        FSObject::Sptr sub_folder = factory.CreateFolder("sub_folder");
        FSObject::Sptr sub_sub_folder = factory.CreateFolder("sub_sub_folder");
        File::Sptr file = make_shared<File>("file1.txt", 10);
        File::Sptr file1 = make_shared<File>("file2.txt", 10);
        File::Sptr file2 = make_shared<File>("file3.txt", 10);
        File::Sptr file3 = make_shared<File>("file4.txt", 10);
        Link::Sptr link = make_shared<Link>(file, "LinkToFile1");

                file->Write(8192);
                file1->Write(4096);
                file2->Write(6000);
                file3->Write(1000);

        sub_sub_folder->AsFolder()->Add(file);
        root_folder->AsFolder()->Add(file2);
        sub_sub_folder->AsFolder()->Add(link);
        sub_folder->AsFolder()->Add(sub_sub_folder);
        sub_folder->AsFolder()->Add(file3);
        root_folder->AsFolder()->Add(sub_folder);
        root_folder->AsFolder()->Add(file1);

        FilterFileVisitor file_filter(5000,9000);

        root_folder->Accept(file_filter);

        TestOK = TestOK && check_dump(ost, "FilterFileVisitor␣Test␣filtered␣amount", static_cast<size_t
             >(2), file_filter.GetFilteredObjects().size());
        TestOK = TestOK && check_dump(ost, "FilterFileVisitor␣Test␣for␣filtered␣file", file2->GetName()
             , file_filter.GetFilteredObjects().cbegin()->lock()->GetName());
        TestOK = TestOK && check_dump(ost, "FilterFileVisitor␣Test␣for␣filtered␣file", file->GetName(),
              file_filter.GetFilteredObjects().crbegin()->lock()->GetName());

        stringstream result;
        stringstream expected;

        file_filter.DumpFiltered(result);

        expected << "\\root\\file3.txt" << std::endl
                 << "\\root\\sub_folder\\sub_sub_folder\\file1.txt" << std::endl;

        TestOK = TestOK && check_dump(ost, "Filter␣File␣Visitor␣Test␣Dump␣", expected.str(), result.str
             ());

    }
    catch (const string& err) {
        error_msg = err;
    }
    catch (bad_alloc const& error) {
        error_msg = error.what();
    }
    catch (const exception& err) {
        error_msg = err.what();
    }
    catch (...) {
        error_msg = "Unhandelt␣Exception";
    }

    TestOK = TestOK && check_dump(ost, "Test␣for␣Exception␣in␣Testcase", true, error_msg.empty());
    error_msg.clear();

    try {

        FilterFileVisitor file_filter{1,2};

        stringstream result;
        result.setstate(ios::badbit);

        file_filter.DumpFiltered(result);
    }
    catch (const string& err) {
        error_msg = err;
    }
    catch (bad_alloc const& error) {
        error_msg = error.what();
    }
    catch (const exception& err) {
        error_msg = err.what();
    }
    catch (...) {
        error_msg = "Unhandelt␣Exception";
    }

    TestOK = TestOK && check_dump(ost, "Test␣for␣Exception␣in␣Dump␣with␣bad␣Ostream", error_msg,
         FilterLinkVisitor::ERROR_BAD_OSTREAM);
    error_msg.clear();

    try {

                FilterFileVisitor file_filter{ 2,1 }; // <= should throw invalid size range
    }
    catch (const string& err) {
        error_msg = err;
    }
    catch (bad_alloc const& error) {
        error_msg = error.what();
    }
    catch (const exception& err) {
        error_msg = err.what();
    }
    catch (...) {
        error_msg = "Unhandelt␣Exception";
    }

    TestOK = TestOK && check_dump(ost, "Test␣for␣Exception␣in␣Filter␣File␣Visiter␣CTOR", error_msg,
         FilterFileVisitor::ERROR_INVALID_SIZE_RANGE);
    error_msg.clear();


    ost << TestEnd;

    return TestOK;
}

bool TestVisitor(ostream& ost, IVisitor& visit)
{
    assert(ost.good());

    ost << TestStart;

    bool TestOK = true;
    string error_msg;


    try {

        stringstream result;

        File::Sptr file = nullptr;

        visit.Visit(file); // <= sould throw Exception Nullptr

    }
    catch (const string& err) {
        error_msg = err;
    }
    catch (bad_alloc const& error) {
        error_msg = error.what();
    }
    catch (const exception& err) {
        error_msg = err.what();
    }
    catch (...) {
        error_msg = "Unhandelt␣Exception";
    }

    TestOK = TestOK && check_dump(ost, "Test␣Exception␣nullptr␣in␣Visit␣File", DumpVisitor::
         ERROR_NULLPTR, error_msg);
    error_msg.clear();

    try {

        stringstream result;

        Folder::Sptr folder = nullptr;

        visit.Visit(folder); // <= sould throw Exception Nullptr

    }
    catch (const string& err) {
        error_msg = err;
    }
    catch (bad_alloc const& error) {
        error_msg = error.what();
    }
    catch (const exception& err) {
        error_msg = err.what();
    }
    catch (...) {
        error_msg = "Unhandelt␣Exception";
    }

    TestOK = TestOK && check_dump(ost, "Test␣Exception␣nullptr␣in␣Visit␣Folder", DumpVisitor::
         ERROR_NULLPTR, error_msg);
    error_msg.clear();

    try {

        stringstream result;

        Link::Sptr lnk = nullptr;

        visit.Visit(lnk); // <= sould throw Exception Nullptr

    }
    catch (const string& err) {
        error_msg = err;
    }
    catch (bad_alloc const& error) {
        error_msg = error.what();
    }
    catch (const exception& err) {
        error_msg = err.what();
    }
    catch (...) {
        error_msg = "Unhandelt␣Exception";
    }

    TestOK = TestOK && check_dump(ost, "Test␣Exception␣nullptr␣in␣Visit␣Link", DumpVisitor::
         ERROR_NULLPTR, error_msg);
    error_msg.clear();

    ost << TestEnd;

    return TestOK;
}

bool TestFactory(ostream& ost)
{
    assert(ost.good());

    ost << TestStart;

    bool TestOK = true;
    string error_msg;


    try {
        FSObjectFactory fact;
        FSObj_Sptr file = fact.CreateFile("file1.txt",10);
        FSObj_Sptr folder = fact.CreateFolder("root");
        FSObj_Sptr lnk = fact.CreateLink("link␣to␣file",file);


        TestOK = TestOK && check_dump(ost, "Test␣if␣file␣was␣constructed", true, file != nullptr);
        TestOK = TestOK && check_dump(ost, "Test␣if␣Link␣was␣constructed", true, lnk->AsLink() !=
             nullptr);
        TestOK = TestOK && check_dump(ost, "Test␣if␣Folder␣was␣constructed", true, folder->AsFolder()
             != nullptr);

    }
    catch (const string& err) {
        error_msg = err;
    }
    catch (bad_alloc const& error) {
        error_msg = error.what();
    }
    catch (const exception& err) {
        error_msg = err.what();
    }
    catch (...) {
        error_msg = "Unhandelt␣Exception";
    }

    TestOK = TestOK && check_dump(ost, "Test␣for␣Execption␣in␣Tesstcase", true, error_msg.empty());
    error_msg.clear();

    try {
        FSObjectFactory fact;
        File::Sptr file= nullptr;
        FSObj_Sptr Lnk = fact.CreateLink("Link␣to␣File", file);

    }
    catch (const string& err) {
        error_msg = err;
    }
    catch (bad_alloc const& error) {
        error_msg = error.what();
    }
    catch (const exception& err) {
        error_msg = err.what();
    }
    catch (...) {
        error_msg = "Unhandelt␣Exception";
    }

    TestOK = TestOK && check_dump(ost, "Test␣Exception␣nullptr␣CTOR␣Link", Link::ERROR_NULLPTR,
         error_msg);
    error_msg.clear();

    ost << TestEnd;

    return TestOK;
}

bool TestLink(ostream& ost)
{
    assert(ost.good());

    ost << TestStart;

    bool TestOK = true;
    string error_msg;

    // test normal operation
    try
    {
        std::string_view folder_name = "MyFolder";
        std::string_view link_name = "LinkToMyFolder";
        Folder::Sptr folder = make_shared<Folder>(folder_name);
        Link::Sptr link = make_shared<Link>(folder, link_name);

        TestOK = TestOK && check_dump(ost, "Test␣normal␣CTOR␣Link", folder_name, link->
             GetReferncedFSObject()->GetName());
        TestOK = TestOK && check_dump(ost, "Test␣normal␣CTOR␣Link", link_name, link->GetName());

    }
    catch (const string& err) {
        error_msg = err;
    }
    catch (bad_alloc const& error) {
        error_msg = error.what();
    }
    catch (const exception& err) {
        error_msg = err.what();
    }
    catch (...) {
        error_msg = "Unhandelt␣Exception";
    }

    TestOK = TestOK && check_dump(ost, "Test␣normal␣CTOR␣Link␣-␣error␣buffer", true, error_msg.empty())
         ;
    error_msg.clear();

    // test Copy CTOR of Link
    try
    {
        std::string_view folder_name = "MyFolder";
        std::string_view link_name = "LinkToMyFolder";
        Folder::Sptr folder = make_shared<Folder>(folder_name);
        Link::Sptr link = make_shared<Link>(folder, link_name);
                Folder::Sptr parent_folder = make_shared<Folder>("AnotherFolder");
                parent_folder->Add(link); // set parent of link

                // Call Copy CTOR
        Link::Sptr link_copy = make_shared<Link>( *link );

        TestOK = TestOK && check_dump(ost, "Test␣Copy␣CTOR␣of␣Link", link->GetReferncedFSObject(),
             link_copy->GetReferncedFSObject());

                // modify copied link referenced FSObject name
                link_copy->GetReferncedFSObject()->SetName("NewFolderName");

        TestOK = TestOK && check_dump(ost, "Test␣for␣shallow␣Copy␣of␣Link", link->GetReferncedFSObject
             (), link_copy->GetReferncedFSObject());

                parent_folder->SetName("Modified");

        TestOK = TestOK && check_dump(ost, "Test␣for␣parent␣of␣Copied␣Link", parent_folder->GetName(),
             link_copy->GetParent().lock()->GetName());

    }
    catch (const string& err) {
        error_msg = err;
    }
    catch (bad_alloc const& error) {
        error_msg = error.what();
    }
    catch (const exception& err) {
        error_msg = err.what();
    }
    catch (...) {
        error_msg = "Unhandelt␣Exception";
    }

    TestOK = TestOK && check_dump(ost, "Test␣normal␣COPY␣CTOR␣Link␣-␣error␣buffer", true, error_msg.
        empty());
    error_msg.clear();

    // test Assign Op of Link
    try
    {
        std::string_view folder_name = "MyFolder";
        std::string_view link_name = "LinkToMyFolder";
        Folder::Sptr folder = make_shared<Folder>(folder_name);
        Link::Sptr link = make_shared<Link>(folder, link_name);
                Folder::Sptr another_folder = make_shared<Folder>("AnotherFolder");
                another_folder->Add(link); // set parent of link

        Link::Sptr link_ass = make_shared<Link>(folder, "Ass␣Link");

        *link_ass = *link;

        TestOK = TestOK && check_dump(ost, "Test␣Assign␣Op␣of␣Link", link->GetReferncedFSObject(),
             link_ass->GetReferncedFSObject());

        another_folder->SetName("Modified");

        TestOK = TestOK && check_dump(ost, "Test␣Assign␣Op␣for␣Parent␣of␣Link", another_folder->GetName
             (), link_ass->GetParent().lock()->GetName());

    }
    catch (const string& err) {
        error_msg = err;
    }
    catch (bad_alloc const& error) {
        error_msg = error.what();
    }
    catch (const exception& err) {
        error_msg = err.what();
    }
    catch (...) {
        error_msg = "Unhandelt␣Exception";
    }

    TestOK = TestOK && check_dump(ost, "Test␣Assing␣Op␣Link␣-␣error␣buffer", true, error_msg.empty());
    error_msg.clear();

    // test Self Assign of Link
    try
    {
        std::string_view folder_name = "MyFolder";
        std::string_view link_name = "LinkToMyFolder";
        Folder::Sptr folder = make_shared<Folder>(folder_name);
        Link::Sptr link = make_shared<Link>(folder, link_name);

        *link = *link; // <= could throw
    }
    catch (const string& err) {
        error_msg = err;
    }
    catch (bad_alloc const& error) {
        error_msg = error.what();
    }
    catch (const exception& err) {
        error_msg = err.what();
    }
    catch (...) {
        error_msg = "Unhandelt␣Exception";
    }

    TestOK = TestOK && check_dump(ost, "Test␣Self␣Assing␣Op␣Link␣-␣error␣buffer", true, error_msg.empty
        ());
    error_msg.clear();

    // test link to nullptr
    try
    {
        Link::Sptr link = make_shared<Link>(nullptr, "LinkToNothing");
    }
    catch (const string& err) {
        error_msg = err;
    }
    catch (bad_alloc const& error) {
        error_msg = error.what();
    }
    catch (const exception& err) {
        error_msg = err.what();
    }
    catch (...) {
        error_msg = "Unhandelt␣Exception";
    }

    TestOK = TestOK && check_dump(ost, "Test␣Exception␣nullptr␣CTOR␣Link", Link::ERROR_NULLPTR,
         error_msg);
    error_msg.clear();

    // test Link with empty string
    try
    {
        File::Sptr file = make_shared<File>("file1.txt", 2048);
        Link::Sptr link = make_shared<Link>(file, "");
    }
    catch (const string& err) {
        error_msg = err;
    }
    catch (bad_alloc const& error) {
        error_msg = error.what();
    }
    catch (const exception& err) {
        error_msg = err.what();
    }
    catch (...) {
        error_msg = "Unhandelt␣Exception";
    }

    TestOK = TestOK && check_dump(ost, "Test␣Exception␣empty␣string␣CTOR␣Link", Link::
         ERROR_STRING_EMPTY, error_msg);
    error_msg.clear();


    // test Link GetReferencedFSObject
    try
    {
        File::Sptr file = make_shared<File>("file1.txt", 2048);
        Link::Sptr link = make_shared<Link>(file, file->GetName());

        FSObj_Sptr ref = link->GetReferncedFSObject();// <= should be File not Folder

        TestOK = TestOK && check_dump(ost, "Test␣GetReferencedFSObject", file->GetName(), ref->GetName
            ());
    }
    catch (const string& err) {
        error_msg = err;
    }
    catch (bad_alloc const& error) {
        error_msg = error.what();
    }
    catch (const exception& err) {
        error_msg = err.what();
    }
    catch (...) {
        error_msg = "Unhandelt␣Exception";
    }

    TestOK = TestOK && check_dump(ost, "Empty␣error␣buffer", true, error_msg.empty());
    error_msg.clear();

    // Link to a Link (chained links)
    try
    {
        File::Sptr file = make_shared<File>("original.txt", 2048);
        Link::Sptr link1 = make_shared<Link>(file, "Link1");
        Link::Sptr link2 = make_shared<Link>(link1, "Link2");

        TestOK = TestOK && check_dump(ost, "Test␣chained␣links",
            link1->GetName(), link2->GetReferncedFSObject()->GetName());
    }
    catch (const exception& err) {
        error_msg = err.what();
    }
    TestOK = TestOK && check_dump(ost, "Test␣chained␣links␣-␣error␣buffer", true, error_msg.empty());
    error_msg.clear();

    //Link when referenced object is destroyed (weak_ptr expiration)
    try
    {
        Link::Sptr link;
        {
            File::Sptr file = make_shared<File>("temp.txt", 2048);
            link = make_shared<Link>(file, "LinkToTemp");
            TestOK = TestOK && check_dump(ost, "Test␣link␣before␣destruction",
                true, link->GetReferncedFSObject() != nullptr);
        } // file goes out of scope here

        FSObj_Sptr expired_ref = link->GetReferncedFSObject();
        TestOK = TestOK && check_dump(ost, "Test␣link␣after␣object␣destruction",
            true, expired_ref == nullptr);
    }
    catch (const exception& err) {
        error_msg = err.what();
    }
    TestOK = TestOK && check_dump(ost, "Test␣weak_ptr␣expiration␣-␣error␣buffer", true, error_msg.empty
        ());
    error_msg.clear();

    //AsLink() method returns valid pointer
    try
    {
        File::Sptr file = make_shared<File>("file.txt", 2048);
        Link::Sptr link = make_shared<Link>(file, "TestLink");

        std::shared_ptr<const ILink> ilink = link->AsLink();
        TestOK = TestOK && check_dump(ost, "Test␣AsLink()␣returns␣valid␣pointer",
            true, ilink != nullptr);
        TestOK = TestOK && check_dump(ost, "Test␣AsLink()␣reference␣matches",
            file->GetName(), ilink->GetReferncedFSObject()->GetName());
    }
    catch (const exception& err) {
        error_msg = err.what();
    }
    TestOK = TestOK && check_dump(ost, "Test␣AsLink()␣-␣error␣buffer", true, error_msg.empty());
    error_msg.clear();

    //Link SetName functionality
    try
    {
        File::Sptr file = make_shared<File>("file.txt", 2048);
        Link::Sptr link = make_shared<Link>(file, "OriginalName");

        link->SetName("NewName");
        TestOK = TestOK && check_dump(ost, "Test␣Link␣SetName",
            string_view("NewName"), link->GetName());
    }
    catch (const exception& err) {
        error_msg = err.what();
    }
    TestOK = TestOK && check_dump(ost, "Test␣SetName␣-␣error␣buffer", true, error_msg.empty());
    error_msg.clear();

    //Link SetName with empty string (should throw)
    try
    {
        File::Sptr file = make_shared<File>("file.txt", 2048);
        Link::Sptr link = make_shared<Link>(file, "OriginalName");
        link->SetName(""); // should throw
    }
    catch (const exception& err) {
        error_msg = err.what();
    }
    TestOK = TestOK && check_dump(ost, "Test␣Link␣SetName␣empty␣string",
        FSObject::ERROR_STRING_EMPTY, error_msg);
    error_msg.clear();

    // Link Accept visitor
    try
    {
        File::Sptr file = make_shared<File>("file.txt", 2048);
        Link::Sptr link = make_shared<Link>(file, "TestLink");
        stringstream result;
        DumpVisitor visitor(result);

        link->Accept(visitor);
        TestOK = TestOK && check_dump(ost, "Test␣Link␣Accept␣visitor␣-␣not␣empty",
            false, result.str().empty());
    }
    catch (const exception& err) {
        error_msg = err.what();
    }
    TestOK = TestOK && check_dump(ost, "Test␣Link␣Accept␣-␣error␣buffer", true, error_msg.empty());
    error_msg.clear();

    ost << TestEnd;
    return TestOK;
}
bool TestFolder(ostream& ost)
{
    assert(ost.good());

    ost << TestStart;

    bool TestOK = true;
    string error_msg;

    // test folder as intended
    try
    {
        string_view folder_name = "MyFolder";
        Folder::Sptr folder = make_shared<Folder>(folder_name);
        TestOK = TestOK && check_dump(ost, "Test␣normal␣CTOR␣Folder", folder_name, folder->GetName());

        File::Sptr file1 = make_shared<File>("file1.txt", 2048);
        File::Sptr file2 = make_shared<File>("file2.txt", 4096);

        folder->Add(file1);
        folder->Add(file2);

        FSObject::Sptr err_file = folder->GetChild(2); // <= should be nullptr
        FSObject::Sptr shared_null = nullptr;

        TestOK = TestOK && check_dump(ost, "Get␣Child␣from␣folder", static_pointer_cast<FSObject>(file1
             ), folder->GetChild(0));
        TestOK = TestOK && check_dump(ost, "Get␣next␣Child␣from␣folder", static_pointer_cast<FSObject>(
             file2), folder->GetChild(1));
        TestOK = TestOK && check_dump(ost, "Get␣Child␣for␣invalid␣index", err_file, shared_null);
    }
    catch (const string& err) {
        error_msg = err;
    }
    catch (bad_alloc const& error) {
        error_msg = error.what();
    }
    catch (const exception& err) {
        error_msg = err.what();
    }
    catch (...) {
        error_msg = "Unhandelt␣Exception";
    }

    TestOK = TestOK && check_dump(ost, "Test␣Folder␣-␣error␣buffer", error_msg.empty(), true);
    error_msg.clear();

    // test Copy Ctor of Folder
    try
    {
        string_view folder_name = "MyFolder";
        Folder::Sptr folder = make_shared<Folder>( folder_name );
        File::Sptr file1 = make_shared<File>("file1.txt", 2048);
        File::Sptr file2 = make_shared<File>("file2.txt", 4096);
                Folder::Sptr sub_folder = make_shared<Folder>("SubFolder");
                File::Sptr sub_file = make_shared<File>("sub_file.txt", 1024);

        folder->Add(file1);
        folder->Add(file2);
                folder->Add(sub_folder);
                sub_folder->Add(sub_file);

                // Call Copy Ctor
        FSObject::Sptr folder_copy = folder->Clone();

                TestOK = TestOK && check_dump(ost, "Test␣Copy␣Ctor␣Folder␣-␣Child␣0", file1->GetName(),
                       folder_copy->AsFolder()->GetChild(0)->GetName());
                TestOK = TestOK && check_dump(ost, "Test␣Copy␣Ctor␣Folder␣-␣Sub␣Folder␣File", sub_file
                      ->GetName(), folder_copy->AsFolder()->GetChild(2)->AsFolder()->GetChild(0)->
                      GetName());

                file1->SetName("modified_file1.txt");
                sub_file->SetName("modified_sub.txt");

                TestOK = TestOK && check_dump(ost, "Test␣Copy␣Ctor␣Folder␣test␣for␣Deep␣Copy", true,
                      file1->GetName() !=folder_copy->AsFolder()->GetChild(0)->GetName());
                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());

                TestOK = TestOK && check_dump(ost, "Test␣Parent␣of␣Copied␣Folder", static_pointer_cast<
                      FSObject>(folder_copy), folder_copy->AsFolder()->GetChild(0)->GetParent().lock());

    }
    catch (const string& err) {
        error_msg = err;
    }
    catch (bad_alloc const& error) {
        error_msg = error.what();
    }
    catch (const exception& err) {
        error_msg = err.what();
    }
    catch (...) {
        error_msg = "Unhandelt␣Exception";
    }

    TestOK = TestOK && check_dump(ost, "Test␣Folder␣-␣error␣buffer", error_msg.empty(), true);
    error_msg.clear();

        // test Assign Opterator of Folder
    try
    {
        string_view folder_name = "MyFolder";
        Folder::Sptr folder = make_shared<Folder>(folder_name);
        File::Sptr file1 = make_shared<File>("file1.txt", 2048);
        File::Sptr file2 = make_shared<File>("file2.txt", 4096);

        folder->Add(file1);
        folder->Add(file2);

        Folder::Sptr folder_ass = make_shared<Folder>( "Ass␣folder");
        *folder_ass = *folder;
```

```

                TestOK = TestOK && check_dump(ost, "Test␣Assign␣Op␣Folder␣-␣Child␣0", file1->GetName(),
                       folder_ass->GetChild(0)->GetName());

        folder->SetName("Modified␣Name");

        TestOK = TestOK && check_dump(ost, "Test␣Assign␣Op␣Folder␣Parent␣-␣Child␣0", folder_ass->
              GetName(), folder_ass->GetChild(0)->GetParent().lock()->GetName());

    }
    catch (const string& err) {
        error_msg = err;
    }
    catch (bad_alloc const& error) {
        error_msg = error.what();
    }
    catch (const exception& err) {
        error_msg = err.what();
    }
    catch (...) {
        error_msg = "Unhandelt␣Exception";
    }

    TestOK = TestOK && check_dump(ost, "Test␣Folder␣-␣error␣buffer", error_msg.empty(), true);
    error_msg.clear();

        // test Assign Opterator of Folder Self Assign
    try
    {
        string_view folder_name = "MyFolder";
        Folder::Sptr folder = make_shared<Folder>(folder_name);
        File::Sptr file1 = make_shared<File>("file1.txt", 2048);
        File::Sptr file2 = make_shared<File>("file2.txt", 4096);

        folder->Add(file1);
        folder->Add(file2);
        *folder = *folder;

                TestOK = TestOK && check_dump(ost, "Test␣Self␣Assign␣Folder␣-␣Child␣0",
                      static_pointer_cast<FSObject>(file1), folder->GetChild(0));

    }
    catch (const string& err) {
        error_msg = err;
    }
    catch (bad_alloc const& error) {
        error_msg = error.what();
    }
    catch (const exception& err) {
        error_msg = err.what();
    }
    catch (...) {
        error_msg = "Unhandelt␣Exception";
    }

    TestOK = TestOK && check_dump(ost, "Test␣Folder␣-␣error␣buffer", error_msg.empty(), true);
    error_msg.clear();

    // test remove child
    try
    {
        Folder::Sptr folder = make_shared<Folder>("MyFolder");
        File::Sptr file1 = make_shared<File>("file1.txt", 2048);
        File::Sptr file2 = make_shared<File>("file2.txt", 4096);
        folder->Add(file1);
        folder->Add(file2);
        folder->Remove(file1);
        TestOK = TestOK && check_dump(ost, "Test␣Remove␣Child␣from␣Folder", static_pointer_cast<
              FSObject>(file2), folder->GetChild(0));
    }
    catch (const string& err) {
        error_msg = err;
    }
    catch (bad_alloc const& error) {
        error_msg = error.what();
    }
    catch (const exception& err) {
        error_msg = err.what();
    }
    catch (...) {
        error_msg = "Unhandelt␣Exception";
    }

    TestOK = TestOK && check_dump(ost, "Test␣Folder␣-␣error␣buffer", error_msg.empty(), true);
    error_msg.clear();

    // test add nullptr
    try
    {
        Folder::Sptr folder = make_shared<Folder>("MyFolder");
        FSObject::Sptr null_ptr = nullptr;
        folder->Add(null_ptr); // <= should throw Exception
    }
    catch (const string& err) {
        error_msg = err;
    }
    catch (bad_alloc const& error) {
        error_msg = error.what();
    }
    catch (const exception& err) {
        error_msg = err.what();
    }
    catch (...) {
        error_msg = "Unhandelt␣Exception";
    }

    TestOK = TestOK && check_dump(ost, "Test␣Folder␣-␣add␣nullptr", Folder::ERROR_NULLPTR, error_msg);
    error_msg.clear();

    // test Folder with empty string
    try
    {
        Folder::Sptr folder = make_shared<Folder>("");
    }
    catch (const string& err) {
        error_msg = err;
    }
    catch (bad_alloc const& error) {
        error_msg = error.what();
    }
    catch (const exception& err) {
        error_msg = err.what();
    }
    catch (...) {
        error_msg = "Unhandelt␣Exception";
    }

    TestOK = TestOK && check_dump(ost, "Test␣Folder␣-␣CTOR␣with␣empty␣string", FSObject::
        ERROR_STRING_EMPTY, error_msg);
    error_msg.clear();

    //Nested folder structure
        try
    {
        Folder::Sptr root = make_shared<Folder>("root");
        Folder::Sptr sub1 = make_shared<Folder>("sub1");
        Folder::Sptr sub2 = make_shared<Folder>("sub2");

        root->Add(sub1);
        sub1->Add(sub2);

        TestOK = TestOK && check_dump(ost, "Test␣nested␣folders␣-␣root␣has␣sub1",
            sub1, static_pointer_cast<Folder>(root->GetChild(0)));
        TestOK = TestOK && check_dump(ost, "Test␣nested␣folders␣-␣sub1␣has␣sub2",
            sub2, static_pointer_cast<Folder>(sub1->GetChild(0)));
    }
    catch (const exception& err) {
        error_msg = err.what();
    }
    TestOK = TestOK && check_dump(ost, "Test␣nested␣folders␣-␣error␣buffer", true, error_msg.empty());
    error_msg.clear();

    //Parent pointer is set correctly when adding child
    try
    {
        Folder::Sptr parent = make_shared<Folder>("parent");
        File::Sptr child = make_shared<File>("child.txt", 2048);

        parent->Add(child);
        FSObj_Wptr parent_wptr = child->GetParent();
        FSObj_Sptr parent_sptr = parent_wptr.lock();

        TestOK = TestOK && check_dump(ost, "Test␣parent␣pointer␣set␣on␣Add",
            parent->GetName(), parent_sptr->GetName());
    }
    catch (const exception& err) {
        error_msg = err.what();
    }
    TestOK = TestOK && check_dump(ost, "Test␣parent␣pointer␣-␣error␣buffer", true, error_msg.empty());
    error_msg.clear();

    //Remove non-existent child (should not crash)
    try
    {
        Folder::Sptr folder = make_shared<Folder>("folder");
        File::Sptr file1 = make_shared<File>("file1.txt", 2048);
        File::Sptr file2 = make_shared<File>("file2.txt", 2048);

        folder->Add(file1);
        folder->Remove(file2); // file2 was never added

        TestOK = TestOK && check_dump(ost, "Test␣remove␣non-existent␣child",
            static_pointer_cast<FSObject>(file1), folder->GetChild(0));
    }
    catch (const exception& err) {
        error_msg = err.what();
    }
    TestOK = TestOK && check_dump(ost, "Test␣remove␣non-existent␣-␣error␣buffer", true, error_msg.empty
        ());
    error_msg.clear();

    //Multiple children of different types
    try
    {
        Folder::Sptr folder = make_shared<Folder>("mixed");
        File::Sptr file = make_shared<File>("file.txt", 2048);
        Folder::Sptr subfolder = make_shared<Folder>("subfolder");
        Link::Sptr link = make_shared<Link>(file, "link");

        folder->Add(file);
        folder->Add(subfolder);
        folder->Add(link);

        TestOK = TestOK && check_dump(ost, "Test␣mixed␣children␣-␣file",
            static_pointer_cast<FSObject>(file), folder->GetChild(0));
        TestOK = TestOK && check_dump(ost, "Test␣mixed␣children␣-␣folder",
            static_pointer_cast<FSObject>(subfolder), folder->GetChild(1));
        TestOK = TestOK && check_dump(ost, "Test␣mixed␣children␣-␣link",
            static_pointer_cast<FSObject>(link), folder->GetChild(2));
    }
    catch (const exception& err) {
        error_msg = err.what();
    }
    TestOK = TestOK && check_dump(ost, "Test␣mixed␣children␣-␣error␣buffer", true, error_msg.empty());
    error_msg.clear();

    //AsFolder() returns valid pointer
    try
    {
        Folder::Sptr folder = make_shared<Folder>("test");
        IFolder::Sptr ifolder = folder->AsFolder();

        TestOK = TestOK && check_dump(ost, "Test␣AsFolder()␣returns␣valid␣pointer",
            true, ifolder != nullptr);
    }
    catch (const exception& err) {
        error_msg = err.what();
    }
    TestOK = TestOK && check_dump(ost, "Test␣AsFolder()␣-␣error␣buffer", true, error_msg.empty());
    error_msg.clear();

    //Accept visitor with children
    try
    {
        Folder::Sptr folder = make_shared<Folder>("root");
        File::Sptr file = make_shared<File>("file.txt", 2048);
        folder->Add(file);

        stringstream result;
        DumpVisitor visitor(result);
        folder->Accept(visitor);

        // Should visit both folder and file
        TestOK = TestOK && check_dump(ost, "Test␣Accept␣visits␣children",
            true, result.str().find("root") != string::npos &&
            result.str().find("file.txt") != string::npos);
    }
    catch (const exception& err) {
        error_msg = err.what();
    }
    TestOK = TestOK && check_dump(ost, "Test␣Accept␣visitor␣-␣error␣buffer", true, error_msg.empty());
    error_msg.clear();

    //SetName on folder
    try
    {
        Folder::Sptr folder = make_shared<Folder>("original");
        folder->SetName("renamed");

        TestOK = TestOK && check_dump(ost, "Test␣Folder␣SetName",
            string_view("renamed"), folder->GetName());
    }
    catch (const exception& err) {
        error_msg = err.what();
    }
    TestOK = TestOK && check_dump(ost, "Test␣Folder␣SetName␣-␣error␣buffer", true, error_msg.empty());
    error_msg.clear();

    ost << TestEnd;
    return TestOK;
}
bool TestFile(ostream& ost)
{
    assert(ost.good());

    ost << TestStart;

    bool TestOK = true;
    string error_msg;

    // File as intended
    try
    {
        string_view file_name = "file1.txt";
        size_t block_size = 2048;
        size_t res_blocks = 20;
        File::Sptr file = make_shared<File>(file_name, res_blocks, block_size);

        TestOK = TestOK && check_dump(ost, "Test␣normal␣CTOR␣File", file_name, file->GetName());
        TestOK = TestOK && check_dump(
            ost, "Test␣normal␣CTOR␣File␣-␣size",
            static_cast<size_t>(0), file->GetSize());

        // Write to file
        size_t write_size = 4096;
        file->Write(write_size);
        TestOK = TestOK && check_dump(ost, "Test␣normal␣-␣write␣file␣size", write_size, file->GetSize()
             );
    }
    catch (const string& err) {
        error_msg = err;
    }
    catch (bad_alloc const& error) {
        error_msg = error.what();
    }
    catch (const exception& err) {
        error_msg = err.what();
    }
    catch (...) {
        error_msg = "Unhandelt␣Exception";
    }

    TestOK = TestOK && check_dump(ost, "Test␣normal␣-␣error␣buffer␣empty", error_msg.empty(), true);
    error_msg.clear();

    // File Copy Ctor
    try
    {
        string_view file_name = "file1.txt";
        size_t block_size = 2048;
        size_t res_blocks = 20;
        File::Sptr file = make_shared<File>( file_name, res_blocks, block_size );
                Folder::Sptr parent_folder = make_shared<Folder>("ParentFolder");
        parent_folder->Add(file);

                File file_copy = *file; // Copy ctor

        // Write to file
        size_t write_size = 4096;

        file->Write(write_size);

        TestOK = TestOK && check_dump(ost, "Test␣Copy␣Ctor␣",file->GetName(), file_copy.GetName());

        TestOK = TestOK && check_dump(ost, "Test␣Copy␣Ctor␣Parent␣of␣file", file->GetParent().lock()->
             GetName(), file_copy.GetParent().lock()->GetName());
    }
    catch (const string& err) {
        error_msg = err;
    }
    catch (bad_alloc const& error) {
        error_msg = error.what();
    }
    catch (const exception& err) {
        error_msg = err.what();
    }
    catch (...) {
        error_msg = "Unhandelt␣Exception";
    }

    TestOK = TestOK && check_dump(ost, "Test␣normal␣-␣error␣buffer␣empty", error_msg.empty(), true);
    error_msg.clear();

        // Assign Operator is deletet because of const members!

    // File with empty string
    try
    {
        File::Sptr file = make_shared<File>("", 20, 2048);
    }
    catch (const string& err) {
        error_msg = err;
    }
    catch (bad_alloc const& error) {
        error_msg = error.what();
    }
    catch (const exception& err) {
        error_msg = err.what();
    }
    catch (...) {
        error_msg = "Unhandelt␣Exception";
    }

    TestOK = TestOK && check_dump(ost, "Test␣CTOR␣Empty␣string␣-␣error␣buffer␣empty", error_msg, File::
         ERROR_STRING_EMPTY);
    error_msg.clear();

    // Write multiple times
    try
    {
        File::Sptr file = make_shared<File>("multi.txt", 10, 2048);

        file->Write(1000);
        file->Write(2000);
        file->Write(3000);

        TestOK = TestOK && check_dump(ost, "Test␣multiple␣writes",
            static_cast<size_t>(6000), file->GetSize());
    }
    catch (const exception& err) {
        error_msg = err.what();
    }
    TestOK = TestOK && check_dump(ost, "Test␣multiple␣writes␣-␣error␣buffer", true, error_msg.empty());
    error_msg.clear();

    // Write exactly to capacity
    try
    {
        size_t blocks = 5;
        size_t blocksize = 1024;
        File::Sptr file = make_shared<File>("exact.txt", blocks, blocksize);

        file->Write(blocks * blocksize); // Write exactly to capacity

        TestOK = TestOK && check_dump(ost, "Test␣write␣to␣exact␣capacity",
            blocks * blocksize, file->GetSize());
    }
    catch (const exception& err) {
        error_msg = err.what();
    }
    TestOK = TestOK && check_dump(ost, "Test␣exact␣capacity␣-␣error␣buffer", true, error_msg.empty());
    error_msg.clear();

    // Write exceeds capacity (should throw)
    try
    {
        File::Sptr file = make_shared<File>("overflow.txt", 2, 1024);
        file->Write(3000); // Exceeds 2 * 1024 = 2048
    }
    catch (const exception& err) {
        error_msg = err.what();
    }
    TestOK = TestOK && check_dump(ost, "Test␣write␣exceeds␣capacity",
        File::ERR_OUT_OF_SPACE, error_msg);
    error_msg.clear();

    // Write zero bytes
    try
    {
        File::Sptr file = make_shared<File>("zero.txt", 10, 2048);
        file->Write(0);

        TestOK = TestOK && check_dump(ost, "Test␣write␣zero␣bytes",
            static_cast<size_t>(0), file->GetSize());
    }
    catch (const exception& err) {
        error_msg = err.what();
    }
    TestOK = TestOK && check_dump(ost, "Test␣write␣zero␣-␣error␣buffer", true, error_msg.empty());
    error_msg.clear();

    // Multiple writes approaching capacity
    try
    {
        File::Sptr file = make_shared<File>("approach.txt", 3, 1000);
        file->Write(1000);
        file->Write(1000);
        file->Write(1000); // Total = 3000, capacity = 3000

        TestOK = TestOK && check_dump(ost, "Test␣multiple␣writes␣to␣capacity",
            static_cast<size_t>(3000), file->GetSize());
    }
    catch (const exception& err) {
        error_msg = err.what();
    }
    TestOK = TestOK && check_dump(ost, "Test␣approach␣capacity␣-␣error␣buffer", true, error_msg.empty()
         );
    error_msg.clear();

    // Write after reaching capacity (should throw)
    try
    {
        File::Sptr file = make_shared<File>("full.txt", 2, 1024);
        file->Write(2048); // Fill to capacity
        file->Write(1);    // Should throw
    }
    catch (const exception& err) {
        error_msg = err.what();
    }
    TestOK = TestOK && check_dump(ost, "Test␣write␣when␣full",File::ERR_OUT_OF_SPACE, error_msg);
    error_msg.clear();

    // File with default blocksize (4096)
    try
    {
        File::Sptr file = make_shared<File>("default.txt", 5); // Default blocksize = 4096
        file->Write(10000);

        TestOK = TestOK && check_dump(ost, "Test␣default␣blocksize", static_cast<size_t>(10000), file->
             GetSize());
    }
    catch (const exception& err) {
        error_msg = err.what();
    }
    TestOK = TestOK && check_dump(ost, "Test␣default␣blocksize␣-␣error␣buffer", true, error_msg.empty()
         );
    error_msg.clear();

    // Accept visitor
    try
    {
        File::Sptr file = make_shared<File>("visitor.txt", 10, 2048);
        stringstream result;
        DumpVisitor visitor(result);

        file->Accept(visitor);

        TestOK = TestOK && check_dump(ost, "Test␣File␣Accept␣visitor", true, result.str().find("visitor
             .txt") != string::npos);
    }
    catch (const exception& err) {
        error_msg = err.what();
    }
    TestOK = TestOK && check_dump(ost, "Test␣File␣Accept␣-␣error␣buffer", true, error_msg.empty());
    error_msg.clear();

    // SetName on file
    try
    {
        File::Sptr file = make_shared<File>("old.txt", 10, 2048);
        file->SetName("new.txt");

        TestOK = TestOK && check_dump(ost, "Test␣File␣SetName",
            string_view("new.txt"), file->GetName());
    }
    catch (const exception& err) {
        error_msg = err.what();
    }
    TestOK = TestOK && check_dump(ost, "Test␣File␣SetName␣-␣error␣buffer", true, error_msg.empty());
    error_msg.clear();

    // File AsFolder should return nullptr
    try
    {
        File::Sptr file = make_shared<File>("notfolder.txt", 10, 2048);
        IFolder::Sptr folder_ptr = file->AsFolder();

        TestOK = TestOK && check_dump(ost, "Test␣File␣AsFolder␣returns␣nullptr", true, folder_ptr ==
             nullptr);
    }
    catch (const exception& err) {
        error_msg = err.what();
    }
    TestOK = TestOK && check_dump(ost, "Test␣File␣AsFolder␣-␣error␣buffer", true, error_msg.empty());
    error_msg.clear();

    ost << TestEnd;
    return TestOK;
}

bool TestFileSystem(ostream& ost)
{
    assert(ost.good());

    ost << TestStart;

    bool TestOK = true;
    string error_msg;

    try
    {
                FileSystem fsys;

                fsys.SetFactory(make_unique<FSObjectFactory>());

                // build a Test filesystem using the set Factory
        fsys.CreateTestFilesystem();

                DumpVisitor dumper(ost);

                ost << "Dump␣of␣Test␣Filesystem␣via␣Dump␣Visitor:\n\n";

                fsys.Work(dumper);

                ost << "\n\n";
    }
    catch (const string& err) {
        error_msg = err;
    }
    catch (bad_alloc const& error) {
        error_msg = error.what();
    }
    catch (const exception& err) {
        error_msg = err.what();
    }
    catch (...) {
        error_msg = "Unhandelt␣Exception";
    }

    TestOK = TestOK && check_dump(ost, "Test␣normal␣op␣Filesystem␣-␣error␣buffer␣empty", error_msg.
         empty(), true);
    error_msg.clear();

    try
    {
                FileSystem fsys;


                FSObjectFactory factory;

                FSObject::Sptr root = factory.CreateFolder("root");

        fsys.SetRoot(move(root));

        stringstream result;
        stringstream expected;


                DumpVisitor dumper(result);

                fsys.Work(dumper);

        root = move(fsys.ReturnRoot());

                DumpVisitor expected_dumper(expected);

                root->Accept(expected_dumper);

                TestOK = TestOK && check_dump(ost, "Test␣ReturnRoot␣matches",
                        expected.str(), result.str());

    }
    catch (const string& err) {
        error_msg = err;
    }
    catch (bad_alloc const& error) {
        error_msg = error.what();
    }
    catch (const exception& err) {
        error_msg = err.what();
    }
    catch (...) {
        error_msg = "Unhandelt␣Exception";
    }

    TestOK = TestOK && check_dump(ost, "Test␣normal␣op␣Filesystem␣-␣error␣buffer␣empty", error_msg.
         empty(), true);
    error_msg.clear();


    try
    {
                FileSystem fsys;
        FSObject::Sptr root = nullptr;

                fsys.SetRoot(move(root)); // <= should throw
    }
    catch (const string& err) {
        error_msg = err;
    }
    catch (bad_alloc const& error) {
        error_msg = error.what();
    }
    catch (const exception& err) {
        error_msg = err.what();
    }
    catch (...) {
        error_msg = "Unhandelt␣Exception";
    }

    TestOK = TestOK && check_dump(ost, "Test␣Exception␣Set␣Null␣Root", FileSystem::ERROR_NULLPTR ,
         error_msg);
    error_msg.clear();

    try
    {
                FileSystem fsys;
        FSObjectFactory::Uptr factory = nullptr;

        fsys.SetFactory(move(factory)); // <= should throw
    }
    catch (const string& err) {
        error_msg = err;
    }
    catch (bad_alloc const& error) {
        error_msg = error.what();
    }
    catch (const exception& err) {
        error_msg = err.what();
    }
    catch (...) {
        error_msg = "Unhandelt␣Exception";
    }

    TestOK = TestOK && check_dump(ost, "Test␣Exception␣Set␣Null␣Factory", FileSystem::ERROR_NULLPTR ,
         error_msg);
    error_msg.clear();

    try
    {
                FileSystem fsys;

                fsys.CreateTestFilesystem(); // <= should throw because no factory set
    }
    catch (const string& err) {
        error_msg = err;
    }
    catch (bad_alloc const& error) {
        error_msg = error.what();
    }
    catch (const exception& err) {
        error_msg = err.what();
    }
    catch (...) {
        error_msg = "Unhandelt␣Exception";
    }

    TestOK = TestOK && check_dump(ost, "Test␣Exception␣no␣Factory␣in␣Create␣Test␣FileSystem",
         FileSystem::ERROR_NULLPTR ,error_msg);
    error_msg.clear();

    try
    {
                FileSystem fsys;
                DumpVisitor dumper(ost);
                fsys.Work(dumper); // <= should throw because root is null
    }
    catch (const string& err) {
        error_msg = err;
    }
    catch (bad_alloc const& error) {
        error_msg = error.what();
    }
    catch (const exception& err) {
        error_msg = err.what();
    }
    catch (...) {
        error_msg = "Unhandelt␣Exception";
    }

    TestOK = TestOK && check_dump(ost, "Test␣Exception␣Work␣with␣no␣root␣set", FileSystem::
         ERROR_NULLPTR ,error_msg);
    error_msg.clear();


    ost << TestEnd;

    return TestOK;
}
```

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
```

```
        if (ost.good()) {
                ost << "TEST_OK!!" << std::endl;
        }
#endif // COLOR_OUTPUT

        return ost;
}

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

#if COLOR_OUTPUT
        if (ost.good()) {
                ost << colorRed() << "TEST␣FAILED␣!!" << colorWhite() << std::endl;

        }
#else
        if (ost.good()) {
                ost << "TEST␣FAILED␣!!" << std::endl;

        }
#endif // COLOR_OUTPUT

        return ost;
}

/**
        * \brief function that reports if the testcase was successful.
        *
        * \param testcase       String that indicates the testcase
        * \param succsessful true -> reports to cout test OK
        * \param succsessful false -> reports test failed
        */
template <typename T>
bool check_dump(std::ostream& ostr, const std::string& testcase, const T& expected, const T& result) {
        if (ostr.good()) {
#if COLOR_OUTPUT
                if (expected == result) {
                        ostr << testcase << std::endl << colorGreen() << "[Test␣OK]␣" << colorWhite() <<"Result:␣(Expected:␣" << std::boolalpha << expected << "␣==" << "␣Result:␣" << result << ")" << std::noboolalpha << std::endl << std::endl;
                }
                else {
                        ostr << testcase << std::endl << colorRed() << "[Test␣FAILED]␣" << colorWhite() << "Result:␣(Expected:␣" << std::boolalpha << expected << "␣!=" << "␣Result:␣" << result << ")" << std::noboolalpha << std::endl << std::endl;
                }
#else
                if (expected == result) {
                        ostr << testcase << std::endl << "[Test␣OK]␣" << "Result:␣(Expected:␣" << std::boolalpha << expected << "␣==" << "␣Result:␣" << result << ")" << std::noboolalpha << std::endl << std::endl;
                }
                else {
                        ostr << testcase << std::endl << "[Test␣FAILED]␣" << "Result:␣(Expected:␣" << std::boolalpha << expected << "␣!=" << "␣Result:␣" << result << ")" << std::noboolalpha << std::endl << std::endl;
                }
#endif
                if (ostr.fail()) {
                        std::cerr << "Error:␣Write␣Ostream" << std::endl;
                }
        }
        else {
                std::cerr << "Error:␣Bad␣Ostream" << std::endl;
        }
        return expected == result;
}

template <typename T1, typename T2>
std::ostream& operator<< (std::ostream& ost,const std::pair<T1,T2> & p) {
        if (!ost.good()) throw std::runtime_error("Error␣bad␣Ostream!");
        ost << "(" << p.first << "," << p.second << ")";
        return ost;
}

```

```
template <typename T>
std::ostream& operator<< (std::ostream& ost,const std::vector<T> & cont) {
        if (!ost.good()) throw std::runtime_error("Error␣bad␣Ostream!");
        std::copy(cont.cbegin(), cont.cend(), std::ostream_iterator<T>(ost, "␣"));
        return ost;
}

template <typename T>
std::ostream& operator<< (std::ostream& ost,const std::list<T> & cont) {
        if (!ost.good()) throw std::runtime_error("Error␣bad␣Ostream!");
        std::copy(cont.cbegin(), cont.cend(), std::ostream_iterator<T>(ost, "␣"));
        return ost;
}

template <typename T>
std::ostream& operator<< (std::ostream& ost,const std::deque<T> & cont) {
        if (!ost.good()) throw std::runtime_error("Error␣bad␣Ostream!");
        std::copy(cont.cbegin(), cont.cend(), std::ostream_iterator<T>(ost, "␣"));
        return ost;
}

template <typename T>
std::ostream& operator<< (std::ostream& ost,const std::forward_list<T> & cont) {
        if (!ost.good()) throw std::runtime_error("Error␣bad␣Ostream!");
        std::copy(cont.cbegin(), cont.cend(), std::ostream_iterator<T>(ost, "␣"));
        return ost;
}


#endif // !TEST_HPP
```