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

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

Mat.Nr: S2410306027 / S2410306014 — Punkte:

Übungsgruppe: 1 — korrigiert:

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

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

Eine Datei hat zusätzlich folgende Eigenschaften:

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

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

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

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

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

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

Name des Elementes auf das verwiesen wird ausgegeben werden.

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

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

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

Systemdokumentation
Projekt Filesystem

Version 1.0

S. Offenberger, S. Vogelhuber

Hagenberg, 6. Dezember 2025

Inhaltsverzeichnis

1 Organisatorisches

1.1 Team

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

1.2 Aufteilung der Verantwortlichkeitsbereiche

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

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

1.3 Aufwand

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

2 Anforderungsdefinition (Systemspezifikation)

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

2.1 Systemüberblick

Das System verwaltet drei Arten von Dateisystemelementen:

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

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

2.2 Funktionale Anforderungen

2.2.1 Dateien

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

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

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

- `Write(size_t const bytes)`

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

2.2.2 Ordner

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

2.2.3 Verweise

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

2.3 Erzeugung der Elemente

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

2.4 Besucher (Visitor) Anforderungen

2.4.1 Visitor: Dump

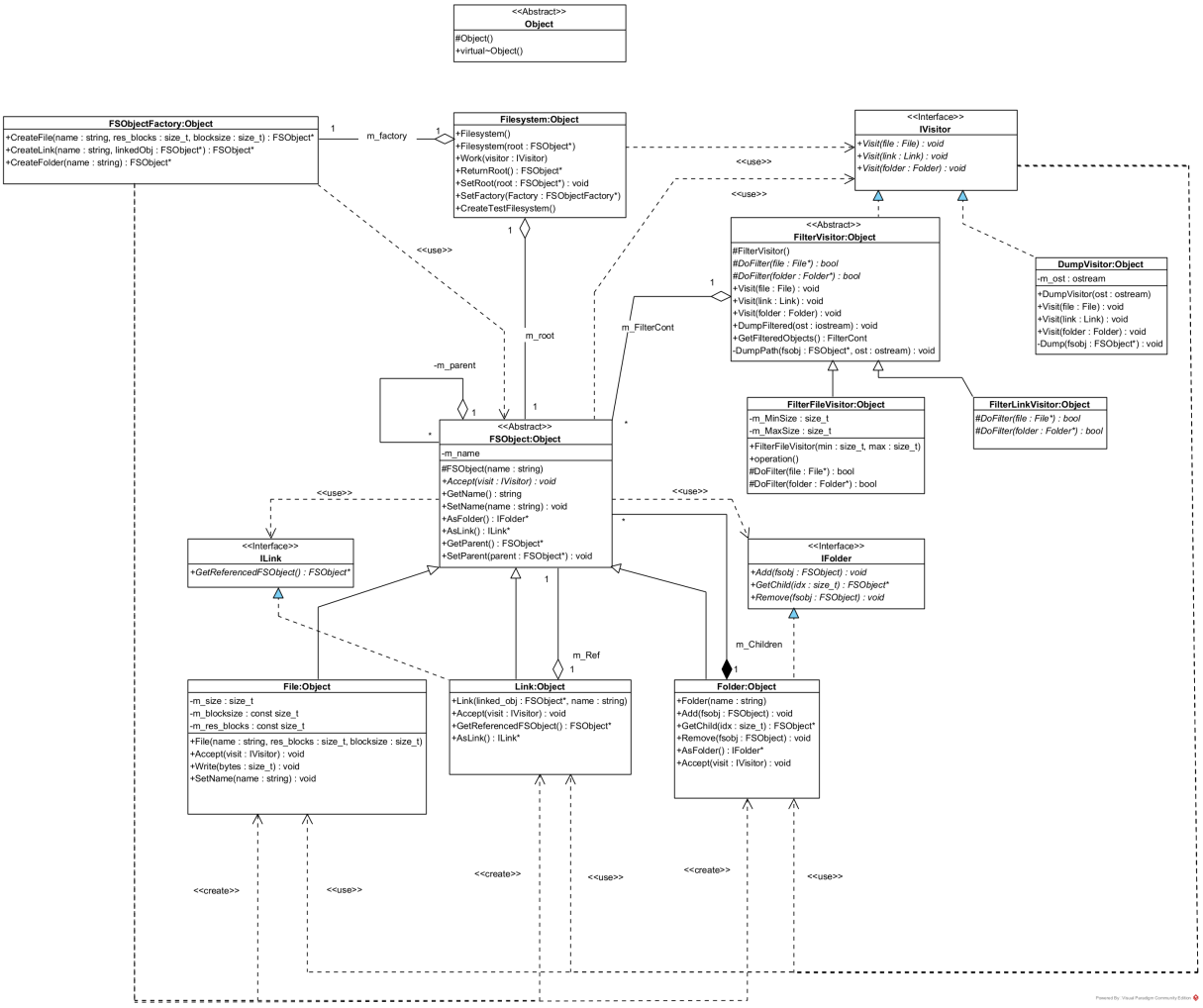
- Gibt die gesamte Dateisystemhierarchie aus.
- Ausgabe sowohl auf der Standardausgabe als auch in einer Datei möglich.
- Muss Dateien, Ordner und Verweise in strukturierter Form darstellen.

2.4.2 Visitor: FilterFiles

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

3 Systementwurf

3.1 Klassendiagramm



3.2 Designentscheidungen

Aus der Aufgabenstellung lassen sich folgenden Designpattern ableiten:

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

3.3 Composite Pattern

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

3.4 Factory Pattern

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

3.5 Visitor Pattern

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

3.6 Template Methode Pattern

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

4 Dokumentation der Komponenten (Klassen)

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

5 Testprotokollierung

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

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

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

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

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

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

Test for Exception in Filter File Visiter CTOR
[Test OK] Result: (Expected: Invalid size range: minimum size
    ↪ must be less than maximum size == Result: Invalid size
    ↪ range: minimum size must be less than maximum size)


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


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

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

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

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

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

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


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

```

```
158
159 ****************************************
160               TESTCASE START
161 ****************************************
162
163 Test normal CTOR Link
164 [Test OK] Result: (Expected: MyFolder == Result: MyFolder)
165
166 Test normal CTOR Link
167 [Test OK] Result: (Expected: LinkToMyFolder == Result:
    ↪ LinkToMyFolder)
168
169 Test normal CTOR Link - error buffer
170 [Test OK] Result: (Expected: true == Result: true)
171
172 Test Exception nullptr CTOR Link
173 [Test OK] Result: (Expected: ERROR Nullptr == Result: ERROR
    ↪ Nullptr)
174
175 Test Exception empty string CTOR Link
176 [Test OK] Result: (Expected: ERROR String Empty == Result:
    ↪ ERROR String Empty)
177
178 Test GetReferencedFSObject
179 [Test OK] Result: (Expected: file1.txt == Result: file1.txt)
180
181 Empty error buffer
182 [Test OK] Result: (Expected: true == Result: true)
183
184
185 ****************************************
186               TESTCASE START
187 ****************************************
188
189 Test normal CTOR Folder
190 [Test OK] Result: (Expected: MyFolder == Result: MyFolder)
191
192 Get Child from folder
193 [Test OK] Result: (Expected: 0000027F833A1150 == Result:
    ↪ 0000027F833A1150)
194
195 Get next Child from folder
196 [Test OK] Result: (Expected: 0000027F833A7580 == Result:
    ↪ 0000027F833A7580)
```

```
197
198 Get Child for invalid index
199 [Test OK] Result: (Expected: 0000000000000000 == Result:
    ↪ 0000000000000000)
200
201 Test Folder - error buffer
202 [Test OK] Result: (Expected: true == Result: true)
203
204 Test Remove Child from Folder
205 [Test OK] Result: (Expected: 0000027F833A7580 == Result:
    ↪ 0000027F833A7580)
206
207 Test Folder - error buffer
208 [Test OK] Result: (Expected: true == Result: true)
209
210 Test Folder - add nullptr
211 [Test OK] Result: (Expected: ERROR Nullptr == Result: ERROR
    ↪ Nullptr)
212
213 Test Folder - CTOR with empty string
214 [Test OK] Result: (Expected: ERROR String Empty == Result:
    ↪ ERROR String Empty)
215
216
217 ****************************************
218
219
220 ****************************************
221               TESTCASE START
222 ****************************************
223
224 Test normal CTOR File
225 [Test OK] Result: (Expected: file1.txt == Result: file1.txt)
226
227 Test normal CTOR File - size
228 [Test OK] Result: (Expected: 0 == Result: 0)
229
230 Test normal - write file size
231 [Test OK] Result: (Expected: 4096 == Result: 4096)
232
233 Test normal - error buffer empty
234 [Test OK] Result: (Expected: true == Result: true)
235
236 Test CTOR Empty string - error buffer empty
```

```
237 [Test OK] Result: (Expected: ERROR String Empty == Result:
        ↪ ERROR String Empty)
238
239
240 ****************************************
241
242 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

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

6.3 FSObjectFactory.cpp

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

#include "FSObjectFactory.hpp"


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

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

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

6.4 Filesystem.hpp

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

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

class FileSystem : public Object
{
public:

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

        FileSystem() = default;

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

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

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

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

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

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

private:

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

6.5 Filesystem.cpp

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

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

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", 2048));
	root_folder_ptr->Add(m_Factory->CreateFile("file2.txt", 2048));
	root_folder_ptr->Add(m_Factory->CreateFile("file3.txt", 2048));
	root_folder_ptr->Add(m_Factory->CreateFile("file4.txt", 2048));
	root_folder_ptr->Add(sub_folder);
	sub_folder_ptr->Add(m_Factory->CreateFile("file5.txt", 8192));
	sub_folder_ptr->Add(m_Factory->CreateFile("file6.txt", 32768));
	sub_folder_ptr->Add(sub_sub_folder);
	sub_sub_folder_ptr->Add(m_Factory->CreateFile("file7.txt", 12288));
	sub_sub_folder_ptr->Add(m_Factory->CreateLink("LinkToRoot", root_folder));

	m_Root = move(root_folder);
}
```

6.6 FSObject.hpp

```cpp
/*****************************************************************//**
 * \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 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 SetParant(Sptr parent);

protected:
```

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

6.7 FSObject.cpp

```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::SetParant(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;

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

6.9 File.cpp

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

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

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

    m_size += bytes;
}

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

6.10 IFolder.hpp

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

// fwd declaration
class FSObject;

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

class IFolder
{
public:

        using Sptr = std::shared_ptr<IFolder>;

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

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

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

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

private:
};

#endif
```

6.11 Folder.hpp

```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::shared_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;

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>
/** \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->SetParant(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);
 }
}
```

6.13 ILink.hpp

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

 // fwd declaration
class FSObject;

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

class ILink
{
public:

        using Sptr = std::shared_ptr<ILink>;

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

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

private:
};

#endif
```

6.14 Link.hpp

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

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

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

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

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

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

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

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

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

#endif
```

6.15 Link.cpp

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

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

    m_Ref = move(linked_obj);
}

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

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

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

6.16 IVisitor.hpp

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

6.17 FilterVisitor.hpp

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

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

#include <vector>
#include <ostream>

class FilterVisitor : public Object, public IVisitor
{
public:

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

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

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

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

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

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

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

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
```

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

6.18 FilterVisitor.cpp

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

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


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

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

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

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

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

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

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

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

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

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

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

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

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

6.19 FilterFileVisitor.hpp

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

#include "FilterVisitor.hpp"

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

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

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

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

6.22 FilterLinkVisitor.cpp

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

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

/** \brief Links are accepted by link filter */
bool FilterLinkVisitor::DoFilter(const std::shared_ptr<const Link>& link) const
{
        assert(link != nullptr);
        return true;
}
```

6.23 DumpVisitor.hpp

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

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

class DumpVisitor : public Object, public IVisitor
{
public:

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

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

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

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

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

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

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

#endif
```

6.24 DumpVisitor.cpp

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

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



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

	Dump(folder);
}

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

	Dump(file);
}

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

	Dump(Link);
}

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

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

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

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

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

6.25 main.cpp

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

int main()
{
        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;

    ofstream output{ "Testoutput.txt" };

    try {

```

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

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

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

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

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

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

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

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

```
650     catch (const exception& err) {
651         error_msg = err.what();
652     }
653     catch (...) {
654         error_msg = "Unhandelt Exception";
655     }
656 
657     TestOK = TestOK && check_dump(ost, "Test Exception empty string CTOR Link", Link::
            ERROR_STRING_EMPTY, error_msg);
658     error_msg.clear();
659 
660 
661     // test Link GetReferencedFSObject
662     try
663     {
664         File::Sptr file = make_shared<File>("file1.txt", 2048);
665         Link::Sptr link = make_shared<Link>(file, file->GetName());
666 
667         FSObj_Sptr ref = link->GetReferncedFSObject();// <= should be File not Folder
668 
669         TestOK = TestOK && check_dump(ost, "Test GetReferencedFSObject", file->GetName(), ref->GetName
                ());
670     }
671     catch (const string& err) {
672         error_msg = err;
673     }
674     catch (bad_alloc const& error) {
675         error_msg = error.what();
676     }
677     catch (const exception& err) {
678         error_msg = err.what();
679     }
680     catch (...) {
681         error_msg = "Unhandelt Exception";
682     }
683 
684     TestOK = TestOK && check_dump(ost, "Empty error buffer", true, error_msg.empty());
685     error_msg.clear();
686 
687     return TestOK;
688 }
689 bool TestFolder(ostream& ost)
690 {
691     assert(ost.good());
692 
693     ost << TestStart;
694 
695     bool TestOK = true;
696     string error_msg;
697 
698     // test folder as intended
699     try
700     {
701         string_view folder_name = "MyFolder";
702         Folder::Sptr folder = make_shared<Folder>(folder_name);
703         TestOK = TestOK && check_dump(ost, "Test normal CTOR Folder", folder_name, folder->GetName());
704 
705         File::Sptr file1 = make_shared<File>("file1.txt", 2048);
706         File::Sptr file2 = make_shared<File>("file2.txt", 4096);
707 
708         folder->Add(file1);
709         folder->Add(file2);
710 
711         FSObject::Sptr err_file = folder->GetChild(2); // <= should be nullptr
712         FSObject::Sptr shared_null = nullptr;
713 
714         TestOK = TestOK && check_dump(ost, "Get Child from folder", static_pointer_cast<FSObject>(file1
                ), folder->GetChild(0));
715         TestOK = TestOK && check_dump(ost, "Get next Child from folder", static_pointer_cast<FSObject>(
                file2), folder->GetChild(1));
716         TestOK = TestOK && check_dump(ost, "Get Child for invalid index", err_file, shared_null);
717     }
718     catch (const string& err) {
719         error_msg = err;
720     }
```

```
721     catch (bad_alloc const& error) {
722         error_msg = error.what();
723     }
724     catch (const exception& err) {
725         error_msg = err.what();
726     }
727     catch (...) {
728         error_msg = "Unhandelt Exception";
729     }
730
731     TestOK = TestOK && check_dump(ost, "Test Folder - error buffer", error_msg.empty(), true);
732     error_msg.clear();
733
734     // test remove child
735     try
736     {
737         Folder::Sptr folder = make_shared<Folder>("MyFolder");
738         File::Sptr file1 = make_shared<File>("file1.txt", 2048);
739         File::Sptr file2 = make_shared<File>("file2.txt", 4096);
740         folder->Add(file1);
741         folder->Add(file2);
742         folder->Remove(file1);
743         TestOK = TestOK && check_dump(ost, "Test Remove Child from Folder", static_pointer_cast<
                FSObject>(file2), folder->GetChild(0));
744     }
745     catch (const string& err) {
746         error_msg = err;
747     }
748     catch (bad_alloc const& error) {
749         error_msg = error.what();
750     }
751     catch (const exception& err) {
752         error_msg = err.what();
753     }
754     catch (...) {
755         error_msg = "Unhandelt Exception";
756     }
757
758     TestOK = TestOK && check_dump(ost, "Test Folder - error buffer", error_msg.empty(), true);
759     error_msg.clear();
760
761     // test add nullptr
762     try
763     {
764         Folder::Sptr folder = make_shared<Folder>("MyFolder");
765         FSObject::Sptr null_ptr = nullptr;
766         folder->Add(null_ptr); // <= should throw Exception
767     }
768     catch (const string& err) {
769         error_msg = err;
770     }
771     catch (bad_alloc const& error) {
772         error_msg = error.what();
773     }
774     catch (const exception& err) {
775         error_msg = err.what();
776     }
777     catch (...) {
778         error_msg = "Unhandelt Exception";
779     }
780
781     TestOK = TestOK && check_dump(ost, "Test Folder - add nullptr", Folder::ERROR_NULLPTR, error_msg);
782     error_msg.clear();
783
784     // test Folder with empty string
785     try
786     {
787         Folder::Sptr folder = make_shared<Folder>("");
788     }
789     catch (const string& err) {
790         error_msg = err;
791     }
792     catch (bad_alloc const& error) {
793         error_msg = error.what();
794     }
```

```
795     catch (const exception& err) {
796         error_msg = err.what();
797     }
798     catch (...) {
799         error_msg = "Unhandelt Exception";
800     }
801
802     TestOK = TestOK && check_dump(ost, "Test Folder - CTOR with empty string", FSObject::
            ERROR_STRING_EMPTY, error_msg);
803     error_msg.clear();
804
805     ost << TestEnd;
806     return TestOK;
807 }
808 bool TestFile(ostream& ost)
809 {
810     assert(ost.good());
811
812     ost << TestStart;
813
814     bool TestOK = true;
815     string error_msg;
816
817     // File as intended
818     try
819     {
820         string_view file_name = "file1.txt";
821         size_t block_size = 2048;
822         size_t res_blocks = 20;
823         File::Sptr file = make_shared<File>(file_name, res_blocks, block_size);
824
825         TestOK = TestOK && check_dump(ost, "Test normal CTOR File", file_name, file->GetName());
826         TestOK = TestOK && check_dump(
827             ost, "Test normal CTOR File - size",
828             static_cast<size_t>(0), file->GetSize());
829
830         // Write to file
831         size_t write_size = 4096;
832         file->Write(write_size);
833         TestOK = TestOK && check_dump(ost, "Test normal - write file size", write_size, file->GetSize()
                );
834     }
835     catch (const string& err) {
836         error_msg = err;
837     }
838     catch (bad_alloc const& error) {
839         error_msg = error.what();
840     }
841     catch (const exception& err) {
842         error_msg = err.what();
843     }
844     catch (...) {
845         error_msg = "Unhandelt Exception";
846     }
847
848     TestOK = TestOK && check_dump(ost, "Test normal - error buffer empty", error_msg.empty(), true);
849     error_msg.clear();
850
851     // File with empty string
852     try
853     {
854         File::Sptr file = make_shared<File>("", 20, 2048);
855     }
856     catch (const string& err) {
857         error_msg = err;
858     }
859     catch (bad_alloc const& error) {
860         error_msg = error.what();
861     }
862     catch (const exception& err) {
863         error_msg = err.what();
864     }
865     catch (...) {
866         error_msg = "Unhandelt Exception";
867     }
```

```
868
869     TestOK = TestOK && check_dump(ost, "Test␣CTOR␣Empty␣string␣-␣error␣buffer␣empty", error_msg, File::
            ERROR_STRING_EMPTY);
870     error_msg.clear();
871
872     ost << TestEnd;
873     return TestOK;
874 }
```

6.26 Test.hpp

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

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

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

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

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

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

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

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

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

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

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