

Name: Simon Offenberger / Simon Vogelhuber

Aufwand in h: siehe Doku.

Mat.Nr: S2410306027 / S2410306014

Punkte:

Übungsgruppe: 1

korrigiert:

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

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

Eine Datei hat zusätzlich folgende Eigenschaften:

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

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

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

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

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

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

Name des Elementes auf das verwiesen wird ausgegeben werden.

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

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

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



HSD

FH-HAGENBERG

Systemdokumentation Projekt Filesystem

Version 1.0

S. Offenberger, S. Vogelhuber

Hagenberg, 1. Dezember 2025

Inhaltsverzeichnis

1 Organisatorisches	6
1.1 Team	6
1.2 Aufteilung der Verantwortlichkeitsbereiche	6
1.3 Aufwand	7
2 Anforderungsdefinition (Systemspezifikation)	8
2.1 Systemüberblick	8
2.2 Funktionale Anforderungen	8
2.2.1 Dateien	8
2.2.2 Ordner	9
2.2.3 Verweise	9
2.3 Erzeugung der Elemente	9
2.4 Besucher (Visitor) Anforderungen	9
2.4.1 Visitor: Dump	9
2.4.2 Visitor: FilterFiles	10
3 Systementwurf	11
3.1 Klassendiagramm	11
3.2 Designentscheidungen	12
4 Dokumentation der Komponenten (Klassen)	12
5 Testprotokollierung	13
6 Quellcode	17
6.1 Object.hpp	17
6.2 FSObjectFactory.hpp	18
6.3 FSObjectFactory.cpp	19
6.4 Filesystem.hpp	20
6.5 Filesystem.cpp	21
6.6 FSObject.hpp	22
6.7 FSObject.cpp	24
6.8 File.hpp	25
6.9 File.cpp	26
6.10 IFolder.hpp	27

6.11	Folder.hpp	28
6.12	Folder.cpp	29
6.13	ILink.hpp	30
6.14	Link.hpp	31
6.15	Link.cpp	32
6.16	IVisitor.hpp	33
6.17	FilterVisitor.hpp	34
6.18	FilterVisitor.cpp	36
6.19	FilterFileVisitor.hpp	38
6.20	FilterFileVisitor.cpp	39
6.21	FilterLinkVisitor.hpp	40
6.22	FilterLinkVisitor.cpp	41
6.23	DumpVisitor.hpp	42
6.24	DumpVisitor.cpp	43
6.25	main.cpp	44
6.26	Test.hpp	52

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 7 Ph / tatsächlich 9 Ph
- Simon Vogelhuber: geschätzt 8 Ph / tatsächlich 7 Ph

2 Anforderungsdefinition (Systemspezifikation)

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

2.1 Systemüberblick

Das System verwaltet drei Arten von Dateisystemelementen:

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

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

2.2 Funktionale Anforderungen

2.2.1 Dateien

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

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

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

- `Write(size_t const bytes)`

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

2.2.2 Ordner

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

2.2.3 Verweise

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

2.3 Erzeugung der Elemente

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

2.4 Besucher (Visitor) Anforderungen

2.4.1 Visitor: Dump

- Gibt die gesamte Dateisystemhierarchie aus.

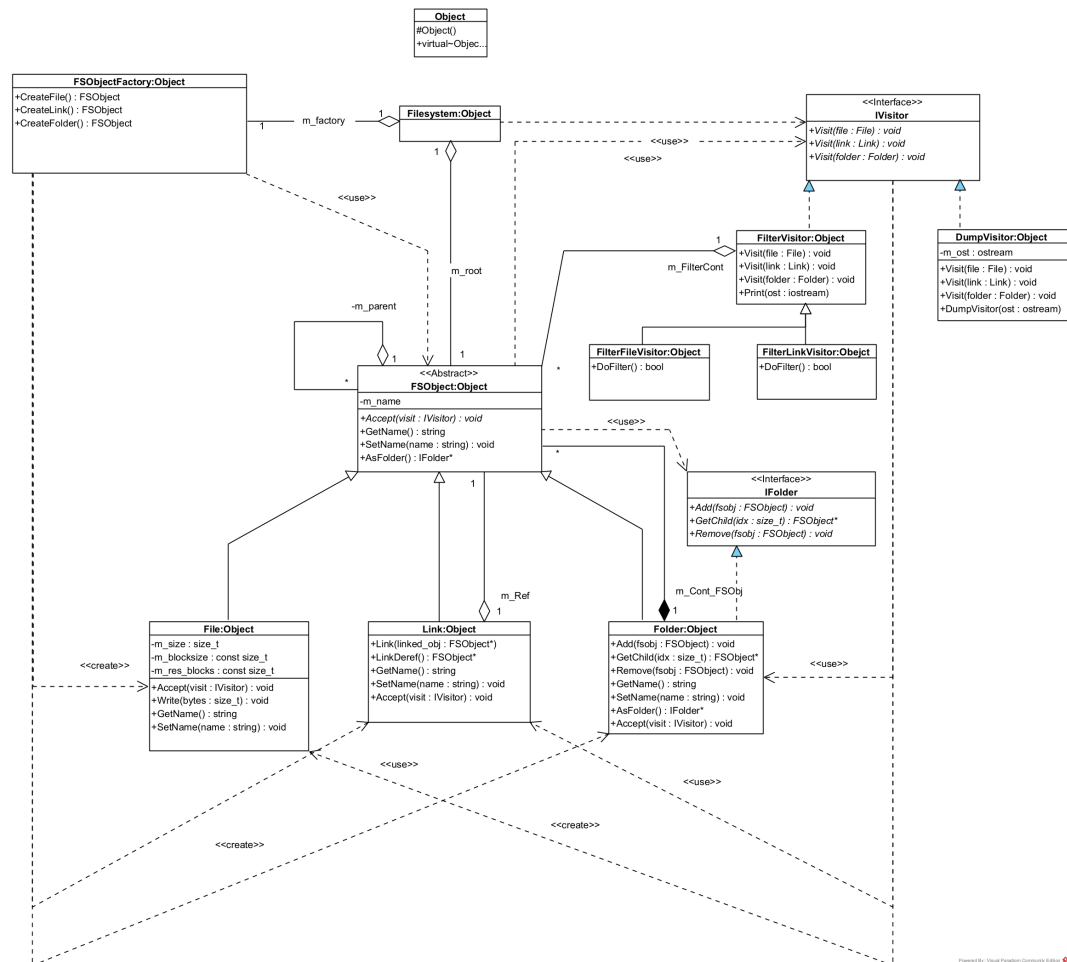
- Ausgabe sowohl auf der Standardausgabe als auch in einer Datei möglich.
- Muss Dateien, Ordner und Verweise in strukturierter Form darstellen.

2.4.2 Visitor: FilterFiles

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

3 Systementwurf

3.1 Klassendiagramm



3.2 Designentscheidungen

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

```
42
43
44 *****
45 TESTCASE START
46 *****
47
48 Test Exception nullptr in Visit File
49 [Test OK] Result: (Expected: ERROR nullptr == Result: ERROR nullptr)
50
51 Test Exception nullptr in Visit Folder
52 [Test OK] Result: (Expected: ERROR nullptr == Result: ERROR nullptr)
53
54 Test Exception nullptr in Visit Link
55 [Test OK] Result: (Expected: ERROR nullptr == Result: ERROR nullptr)
56
57
58 *****
59
60
61 *****
62 TESTCASE START
63 *****
64
65 Test Exception nullptr in Visit File
66 [Test OK] Result: (Expected: ERROR nullptr == Result: ERROR nullptr)
67
68 Test Exception nullptr in Visit Folder
69 [Test OK] Result: (Expected: ERROR nullptr == Result: ERROR nullptr)
70
71 Test Exception nullptr in Visit Link
72 [Test OK] Result: (Expected: ERROR nullptr == Result: ERROR nullptr)
73
74
75 *****
76
77
78 *****
79 TESTCASE START
80 *****
81
82 FilterLinkVisitor Test filtered amount
83 [Test OK] Result: (Expected: 1 == Result: 1)
84
85 FilterLinkVisitor Test filtered obj
```

```
86 [Test OK] Result: (Expected: file1.txt == Result: file1.txt)
87
88 Filter Link Visitor Test Dump
89 [Test OK] Result: (Expected: root\sub_folder\sub_sub_folder\LinkToFile1 ->
    ↪ file1.txt
90 == Result: root\sub_folder\sub_sub_folder\LinkToFile1 -> file1.txt
91 )
92
93 Test for Exception in Testcase
94 [Test OK] Result: (Expected: true == Result: true)
95
96 Test for Exception in Dump with bad Ostream
97 [Test OK] Result: (Expected: ERROR: bad output stream == Result: ERROR: bad
    ↪ output stream)
98
99
100 *****
101
102
103 *****
104 TESTCASE START
105 *****
106
107 FilterFileVisitor Test filtered amount
108 [Test OK] Result: (Expected: 2 == Result: 2)
109
110 FilterFileVisitor Test for filtered file
111 [Test OK] Result: (Expected: file3.txt == Result: file3.txt)
112
113 FilterFileVisitor Test for filtered file
114 [Test OK] Result: (Expected: file1.txt == Result: file1.txt)
115
116 Filter File Visitor Test Dump
117 [Test OK] Result: (Expected: root\file3.txt
118 root\sub_folder\sub_sub_folder\file1.txt
119 == Result: root\file3.txt
120 root\sub_folder\sub_sub_folder\file1.txt
121 )
122
123 Test for Exception in Testcase
124 [Test OK] Result: (Expected: true == Result: true)
125
126 Test for Exception in Dump with bad Ostream
```

```
127 [Test OK] Result: (Expected: ERROR: bad output stream == Result: ERROR: bad
    ↳ output stream)
128
129 Test for Exception in Filter File Visitor CTOR
130 [Test OK] Result: (Expected: Invalid size range: minimum size must be less
    ↳ than maximum size == Result: Invalid size range: minimum size must be
    ↳ less than maximum size)
131
132
133 *****
134
135
136 *****
137 TESTCASE START
138 *****
139
140 Test if file was constructed
141 [Test OK] Result: (Expected: true == Result: true)
142
143 Test if Link was constructed
144 [Test OK] Result: (Expected: true == Result: true)
145
146 Test if Folder was constructed
147 [Test OK] Result: (Expected: true == Result: true)
148
149 Test for Exception in Testcase
150 [Test OK] Result: (Expected: true == Result: true)
151
152 Test Exception nullptr CTOR Link
153 [Test OK] Result: (Expected: ERROR Nullptr == Result: ERROR Nullptr)
154
155
156 *****
157
158 TEST OK!!
```


6 Quellcode

6.1 Object.hpp

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

6.2 FSObjectFactory.hpp

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

6.3 FSObjectFactory.cpp

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

6.4 Filesystem.hpp

```
1  /*****  
2  * \file Filesystem.hpp  
3  * \brief Filesystem class representing the root of a filesystem  
4  *  
5  * \author Simon  
6  * \date   November 2025  
7  *****/  
8  #ifndef FILE_SYSTEM_HPP  
9  #define FILE_SYSTEM_HPP  
10  
11 #include "FSObject.hpp"  
12 #include "IVisitor.hpp"  
13  
14 class Filesystem : public Object  
15 {  
16 public:  
17  
18     // Public Error Messages  
19     inline static const std::string ERROR_NULLPTR = "ERROR_Nullptr";  
20  
21     Filesystem() = default;  
22  
23     /** \brief Construct a Filesystem with a root FSObject  
24      * \param root Root FSObject shared pointer  
25      */  
26     Filesystem(FSObject::Sptr root);  
27  
28     /** \brief Walk the filesystem with a visitor  
29      * \param visitor Visitor to apply  
30      * \return Reference to visitor  
31      */  
32     IVisitor& Work(IVisitor& visitor);  
33  
34     /** \brief Returns the root FSObject  
35      * \return Shared pointer to root  
36      */  
37     FSObject::Sptr ReturnRoot();  
38  
39     /** \brief Set the filesystem root  
40      * \param root Shared pointer to new root  
41      */  
42     void SetRoot(FSObject::Sptr root);  
43  
44 private:  
45     FSObject::Sptr m_Root;  
46 };  
47 #endif
```

6.5 Filesystem.cpp

```
1 #include "Filesystem.hpp"
2
3 Filesystem::Filesystem(FSObject::Sptr root)
4 {
5     if (root == nullptr) throw ERROR_NULLPTR;
6
7     m_Root = move(root);
8 }
9
10 IVisitor& Filesystem::Work(IVisitor& visitor)
11 {
12     if (m_Root == nullptr) throw ERROR_NULLPTR;
13
14     m_Root->Accept(visitor);
15
16     return visitor;
17 }
18
19 FSObject::Sptr Filesystem::ReturnRoot()
20 {
21     return move(m_Root);
22 }
23
24 void Filesystem::SetRoot(FSObject::Sptr root)
25 {
26     if (root == nullptr) throw ERROR_NULLPTR;
27
28     m_Root = move(root);
29 }
```

6.6 FSObject.hpp

```

1  /*****
2  * \file FSObject.hpp
3  * \brief Base class for filesystem objects
4  *
5  * \author Simon
6  * \date   November 2025
7  *****/
8  #ifndef FS_OBJECT_HPP
9  #define FS_OBJECT_HPP
10
11 #include "Object.h"
12 #include "IVisitor.hpp"
13 #include "IFolder.hpp"
14 #include "ILink.hpp"
15
16 #include <memory>
17 #include <vector>
18
19 class FSObject : public Object
20 {
21 public:
22     // Public Error Messages
23     inline static const std::string ERROR_NULLPTR = "ERROR_Nullptr";
24
25     // Smart pointer types
26     using Sptr = std::shared_ptr<FSObject>;
27     using Uptr = std::unique_ptr<FSObject>;
28     using Wptr = std::weak_ptr<FSObject>;
29
30     /** \brief Accept a visitor (pure virtual)
31     * \param visit Visitor to accept
32     */
33     virtual void Accept(IVisitor& visit) =0;
34
35     /** \brief Try to "cast" this FSObject to a folder
36     * \return Shared pointer to IFolder or nullptr
37     */
38     virtual IFolder::Sptr AsFolder();
39
40     /** \brief Try to cast this FSObject to a link
41     * \return Shared pointer to ILink or nullptr
42     */
43     virtual std::shared_ptr<const ILink> AsLink() const;
44
45     /** \brief Construct an FSObject with optional name
46     * \param name Name of the FSObject
47     */
48     FSObject(std::string_view name = "") : m_Name{ name } {}
49
50     /** \brief Get the name of the object
51     * \return Name as std::string_view
52     */
53     std::string_view GetName() const;
54
55     /** \brief Set the name of the object
56     * \param name New name
57     */
58     void SetName(std::string_view name);
59
60     /** \brief Set parent of this FSObject
61     * \param parent Shared pointer to parent FSObject
62     */
63     void SetParent(Sptr parent);
64
65     /** \brief Get parent as weak pointer
66     * \return Weak pointer to parent
67     */
68     FSObj_Wptr GetParent() const;
69
70 private:
71     std::string m_Name;
72     FSObj_Wptr m_Parent;

```

```
73 | ;  
74 |  
75 | #endif
```

6.7 FSObject.cpp

```
1 #include "FSObject.hpp"
2 #include <string>
3
4 IFolder::Sptr FSObject::AsFolder()
5 {
6     return nullptr;
7 }
8
9 std::shared_ptr<const ILink> FSObject::AsLink() const
10 {
11     return nullptr;
12 }
13
14 std::string_view FSObject::GetName() const
15 {
16     return std::string_view(m_Name);
17 }
18
19 void FSObject::SetName(std::string_view name)
20 {
21     m_Name = name;
22 }
23
24 void FSObject::SetParant(Sptr parent)
25 {
26     if (parent == nullptr) throw ERROR_NULLPTR;
27     m_Parent = move(parent);
28 }
29
30 FObj_Wptr FSObject::GetParent() const
31 {
32     return m_Parent;
33 }
```


6.8 File.hpp

```
1  /*****  
2  * \file File.hpp  
3  * \brief File class representing a file in the filesystem  
4  *  
5  * \author Simon  
6  * \date   November 2025  
7  *****/  
8  #ifndef FILE_HPP  
9  #define FILE_HPP  
10  
11 #include "FSObject.hpp"  
12  
13 class File : public FSObject, public std::enable_shared_from_this<File>  
14 {  
15 public:  
16     // Public Error Messages  
17     inline static const std::string ERROR_NULLPTR = "ERROR_Nullptr";  
18     static inline const std::string ERR_OUT_OF_SPACE = "Not_enough_space_to_write_data";  
19  
20     // Smart pointer types  
21     using Uptr = std::unique_ptr<File>;  
22     using Sptr = std::shared_ptr<File>;  
23     using Wptr = std::weak_ptr<File>;  
24  
25     /** \brief Construct a file  
26     * \param name File name  
27     * \param res_blocks Reserved blocks  
28     * \param blocksize Block size (default 4096)  
29     */  
30     File(std::string_view name, const size_t res_blocks, const size_t blocksize = 4096)  
31         : m_size(0), m_blocksize(blocksize), FSObject{ name },  
32         m_res_blocks(res_blocks)  
33     {}  
34  
35     /** \brief Accept a visitor  
36     * \param visit Visitor to accept  
37     */  
38     virtual void Accept(IVisitor& visit) override;  
39  
40     /** \brief Write bytes to the file (increases size)  
41     * \param bytes Number of bytes to write  
42     * Call by Value is intentional because it is faster than by reference for built-in types  
43     */  
44     void Write(const size_t bytes);  
45  
46     /** \brief Get current size of the file  
47     * \return Size in bytes  
48     */  
49     size_t GetSize() const;  
50  
51 private:  
52     size_t m_size;  
53     const size_t m_blocksize;  
54     const size_t m_res_blocks;  
55 };  
56 #endif
```

6.9 File.cpp

```
1 #include "File.hpp"
2
3 /** \brief Accept a visitor for this file */
4 void File::Accept(IVisitor& visit)
5 {
6     visit.Visit(shared_from_this());
7 }
8
9 /** \brief Write bytes to the file, throws on out of space */
10 void File::Write(const size_t bytes)
11 {
12     if ((bytes + m_size) > m_blocksize * m_res_blocks)
13         throw ERR_OUT_OF_SPACE;
14
15     m_size += bytes;
16 }
17
18 /** \brief Return current size */
19 size_t File::GetSize() const
20 {
21     return m_size;
22 }
```

6.10 IFolder.hpp

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

6.11 Folder.hpp

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

6.12 Folder.cpp

```
1 #include "Folder.hpp"
2
3 /** \brief Add child to folder, sets parent pointer on child */
4 void Folder::Add(FSObj_Sptr fsobj)
5 {
6     if (fsobj == nullptr) throw FSOBJ::ERROR_NULLPTR;
7     fsobj->SetParent(std::move(shared_from_this()));
8     m_Children.emplace_back(move(fsobj));
9 }
10
11 /** \brief Get child by index */
12 FSObj_Sptr Folder::GetChild(const size_t idx)
13 {
14     if (idx < m_Children.size())
15     {
16         return m_Children.at(idx);
17     }
18     return nullptr;
19 }
20
21
22 /** \brief Remove a child from container */
23 void Folder::Remove(FSObj_Sptr fsobj)
24 {
25     m_Children.erase(
26         std::remove(m_Children.begin(), m_Children.end(), fsobj), m_Children.end()
27     );
28 }
29
30 /** \brief Return this as IFolder shared pointer */
31 IFolder_Sptr Folder::AsFolder()
32 {
33     return shared_from_this();
34 }
35
36 /** \brief Accept a visitor and forward to children */
37 void Folder::Accept(IVisitor& visit)
38 {
39     visit.Visit(shared_from_this());
40
41     for(auto& child : m_Children)
42     {
43         child->Accept(visit);
44     }
45 }
```

6.13 ILink.hpp

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

6.14 Link.hpp

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

6.15 Link.cpp

```
1 #include "Link.hpp"
2
3 /** \brief Construct a link to another FSObject */
4 Link::Link(FSObj_Sptr linked_obj, std::string_view name) : FSObject(name)
5 {
6     if (linked_obj == nullptr) throw Link::ERROR_NULLPTR;
7     m_Ref = move(linked_obj);
8 }
9
10
11 /** \brief Cast to ILink */
12 std::shared_ptr<const ILink> Link::AsLink() const
13 {
14     return move(shared_from_this());
15 }
16
17 /** \brief Get referenced FSObject (shared_ptr) or nullptr */
18 FSObj_Sptr Link::GetReferncedFSObject() const
19 {
20     return m_Ref.lock();
21 }
22
23 /** \brief Accept a visitor */
24 void Link::Accept(IVisitor& visit)
25 {
26     visit.Visit(shared_from_this());
27 }
```


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<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<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<Link>& link)=0;  
36  
37     /** \brief Virtual destructor for visitor implementations */  
38     virtual ~IVisitor() = default;  
39  
40 private:  
41 };  
42  
43 #endif
```

6.17 FilterVisitor.hpp

```

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

```

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

6.18 FilterVisitor.cpp

```

1 #include "FilterVisitor.hpp"
2 #include "Folder.hpp"
3 #include "File.hpp"
4 #include "Link.hpp"
5
6 #include <vector>
7 #include <algorithm>
8 #include <iostream>
9 #include <cassert>
10
11 using namespace std;
12
13 void FilterVisitor::Dump(const FSOBJ_Wptr & fsobj, std::ostream& ost) const
14 {
15     std::vector<FSObject::Sptr> path_components;
16
17     // get shared pointer from weak pointer
18     const FSOBJ_Sptr fsobj_ptr = fsobj.lock();
19
20     if (fsobj_ptr == 0) throw FilterVisitor::ERROR_NULLPTR;
21
22     FSOBJ_Sptr parent = fsobj_ptr->GetParent().lock();
23
24     while (parent != nullptr) {
25         path_components.push_back(parent);
26         parent = parent->GetParent().lock();
27     }
28
29     for_each(path_components.crbegin(), path_components.crend(),
30             [&](const FSOBJ_Sptr& obj) {
31                 ost << obj->GetName() << "\\";
32             });
33
34     ost << fsobj_ptr->GetName();
35
36     std::shared_ptr<const ILink> link_ptr = fsobj_ptr->AsLink();
37
38     if (link_ptr) {
39         const FSOBJ_Sptr linked_obj = link_ptr->GetReferncedFSObject();
40         if (linked_obj) {
41             ost << "_->_" << link_ptr->GetReferncedFSObject()->GetName();
42         }
43     }
44
45     ost << "\n";
46 }
47
48
49 /** \brief Default visit for folder (no-op) */
50 void FilterVisitor::Visit(const std::shared_ptr<Folder>& folder)
51 {
52     if (folder == nullptr) throw ERROR_NULLPTR;
53 }
54
55 /** \brief Visit a file and if it matches add to filtered container */
56 void FilterVisitor::Visit(const std::shared_ptr<File>& file)
57 {
58     if (file == nullptr) throw ERROR_NULLPTR;
59
60     // if file matches filter add to container
61     if(DoFilter(file))
62     {
63         m_FilterCont.emplace_back(file);
64     }
65 }
66
67 /** \brief Visit a link and if it matches add to filtered container */
68 void FilterVisitor::Visit(const std::shared_ptr<Link>& link)
69 {
70     if (link == nullptr) throw ERROR_NULLPTR;
71
72     // if link matches filter add to container

```

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

6.19 FilterFileVisitor.hpp

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

6.20 FilterFileVisitor.cpp

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

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<File>& file) 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<Link>& link) override;  
30  
31 private:  
32 };  
33  
34 #endif
```


6.22 FilterLinkVisitor.cpp

```
1 #include "FilterLinkVisitor.hpp"
2 #include <cassert>
3
4 /** \brief Files are not accepted by link filter */
5 bool FilterLinkVisitor::DoFilter(const std::shared_ptr<File>& file)
6 {
7     assert(file != nullptr);
8     return false;
9 }
10
11 /** \brief Links are accepted by link filter */
12 bool FilterLinkVisitor::DoFilter(const std::shared_ptr<Link>& link)
13 {
14     assert(link != nullptr);
15     return true;
16 }
```

6.23 DumpVisitor.hpp

```
1  /**
2   * \file DumpVisitor.hpp
3   * \brief Visitor that dumps filesystem object paths to an output stream
4   *
5   * \author Simon
6   * \date   November 2025
7   */
8  #ifndef DUMP_VISITOR_HPP
9  #define DUMP_VISITOR_HPP
10
11 #include <iostream>
12 #include "IVisitor.hpp"
13 #include "FSObject.hpp"
14
15 class DumpVisitor : public IVisitor
16 {
17 public:
18
19     // Public Error Messages
20     inline static const std::string ERROR_NULLPTR = "ERROR_Nullptr";
21     inline static const std::string ERROR_BAD_OSTREAM = "ERROR:_bad_output_stream";
22
23     /** \brief Construct a dumper that writes to given ostream
24      * \param ost Output stream reference
25      */
26     DumpVisitor(std::ostream& ost) : m_ost{ ost } {}
27
28     /** \brief Visit folder
29      * \param folder Folder to visit
30      */
31     virtual void Visit(const std::shared_ptr<Folder>& folder) override;
32
33     /** \brief Visit file
34      * \param file File to visit
35      */
36     virtual void Visit(const std::shared_ptr<File>& file) override;
37
38     /** \brief Visit link
39      * \param Link Link to visit
40      */
41     virtual void Visit(const std::shared_ptr<Link>& Link) override;
42
43 private:
44     /** \brief Dump a single FSObject path to the output stream
45      * \param fsobj Shared pointer to object
46      */
47     void Dump(const std::shared_ptr<FSObject>& fsobj);
48
49     // Output stream reference
50     std::ostream & m_ost;
51 };
52
53 #endif
```

6.24 DumpVisitor.cpp

```
1 #include "DumpVisitor.hpp"
2
3 #include "Folder.hpp"
4 #include "File.hpp"
5 #include "Link.hpp"
6
7 #include <vector>
8 #include <algorithm>
9 #include <cassert>
10
11 /** \brief Visit folder and dump its path */
12 void DumpVisitor::Visit(const std::shared_ptr<Folder>& folder)
13 {
14     if (m_ost.fail()) throw ERROR_BAD_OSTREAM;
15     if (folder == nullptr) throw ERROR_NULLPTR;
16
17     Dump(folder);
18 }
19
20 /** \brief Visit file and dump its path */
21 void DumpVisitor::Visit(const std::shared_ptr<File>& file)
22 {
23     if (m_ost.fail()) throw ERROR_BAD_OSTREAM;
24     if (file == nullptr) throw ERROR_NULLPTR;
25
26     Dump(file);
27 }
28
29 /** \brief Visit link and dump its path */
30 void DumpVisitor::Visit(const std::shared_ptr<Link>& Link)
31 {
32     if (m_ost.fail()) throw ERROR_BAD_OSTREAM;
33     if (Link == nullptr) throw ERROR_NULLPTR;
34
35     Dump(Link);
36 }
37
38 /** \brief Dump full path for a FSObject to the internal ostream */
39 void DumpVisitor::Dump(const std::shared_ptr<FSObject>& fsobj)
40 {
41     assert(fsobj != nullptr);
42
43     std::vector<FSObject::Sptr> path_components;
44     path_components.reserve(10); // reserve some space to avoid multiple allocations
45
46     FSObject::Sptr parent = fsobj->GetParent().lock();
47     FSObject::Sptr next_parent;
48
49     while (parent != nullptr) {
50         next_parent = parent->GetParent().lock();
51         path_components.emplace_back(parent);
52         parent = move(next_parent);
53     }
54
55     for_each(path_components.rbegin(), path_components.rend(),
56             [&](const FSObject::Sptr & obj) {
57                 m_ost << obj->GetName() << "\\\";
58             });
59
60     m_ost << fsobj->GetName() << "\n";
61
62 }
```

6.25 main.cpp

```
1  /*****  
2  * \file   main.cpp  
3  * \brief  Testdriver for the filesystem  
4  *  
5  * \author Simon  
6  * \date   December 2025  
7  *****/  
8  
9  #include <iostream>  
10 #include <string>  
11 #include <memory>  
12 #include "FSObject.hpp"  
13 #include "IFolder.hpp"  
14 #include "ILink.hpp"  
15 #include "FSObjectFactory.hpp"  
16 #include "DumpVisitor.hpp"  
17 #include "FilterFileVisitor.hpp"  
18 #include "FilterLinkVisitor.hpp"  
19 #include "Filesystem.hpp"  
20 #include <cassert>  
21 #include <sstream>  
22 #include "Test.hpp"  
23 #include "fstream"  
24 #include "vld.h"  
25  
26 using namespace std;  
27  
28 #define WriteOutputFile ON  
29  
30 static bool TestDumpVisitor(ostream& ost);  
31 static bool TestFilterLinkVisitor(ostream& ost);  
32 static bool TestFilterFileVisitor(ostream& ost);  
33 static bool TestVisitor(ostream& ost, IVisitor & visit);  
34 static bool TestFactory(ostream& ost);  
35  
36 int main()  
37 {  
38     DumpVisitor visitor(std::cout);  
39  
40     FilterLinkVisitor filter_link_visitor;  
41  
42     FilterFileVisitor filter_file_visitor(4096, 16384);  
43  
44     FSObjectFactory factory;  
45  
46     FSObject::Sptr root_folder = factory.CreateFolder();  
47     IFolder::Sptr root_folder_ptr = root_folder->AsFolder();  
48     FSObject::Sptr sub_folder = factory.CreateFolder();  
49     IFolder::Sptr sub_folder_ptr = sub_folder->AsFolder();  
50     FSObject::Sptr sub_sub_folder = factory.CreateFolder();  
51     IFolder::Sptr sub_sub_folder_ptr = sub_sub_folder->AsFolder();  
52  
53     sub_folder_ptr->SetName("sub_folder");  
54     sub_sub_folder_ptr->SetName("sub_sub_folder");  
55  
56     root_folder_ptr->SetName("root");  
57     root_folder_ptr->Add(factory.CreateFile("file1.txt", 2048));  
58     root_folder_ptr->Add(factory.CreateFile("file2.txt", 2048));  
59     root_folder_ptr->Add(factory.CreateFile("file3.txt", 2048));  
60     root_folder_ptr->Add(factory.CreateFile("file4.txt", 2048));  
61     root_folder_ptr->Add(sub_folder);  
62     sub_folder_ptr->Add(factory.CreateFile("file5.txt", 8192));  
63     sub_folder_ptr->Add(factory.CreateFile("file6.txt", 32768));  
64     sub_folder_ptr->Add(sub_sub_folder);  
65     sub_sub_folder_ptr->Add(factory.CreateFile("file7.txt", 12288));  
66     sub_sub_folder_ptr->Add(factory.CreateLink("LinkToRoot", root_folder));  
67  
68  
69     Filesystem homework(move(root_folder));  
70  
71     homework.Work(visitor);  
72
```

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

```
148
149 bool TestDumpVisitor(ostream & ost)
150 {
151     assert(ost.good());
152     ost << TestStart;
153
154     bool TestOK = true;
155     string error_msg;
156
157     try {
158         FSObjectFactory factory;
159         FSObject::Sptr root_folder = factory.CreateFolder("root");
160         FSObject::Sptr sub_folder = factory.CreateFolder("sub_folder");
161         FSObject::Sptr sub_sub_folder = factory.CreateFolder("sub_sub_folder");
162         sub_sub_folder->AsFolder()->Add(File::Sptr(make_shared<File>("file1.txt", 2048)));
163         sub_folder->AsFolder()->Add(sub_sub_folder);
164         root_folder->AsFolder()->Add(sub_folder);
165
166         stringstream result;
167         stringstream expected;
168
169         DumpVisitor dumper(result);
170
171         root_folder->Accept(dumper);
172
173         expected << "root" << std::endl
174                 << "root\\sub_folder" << std::endl
175                 << "root\\sub_folder\\sub_sub_folder" << std::endl
176                 << "root\\sub_folder\\sub_sub_folder\\file1.txt" << std::endl;
177
178         TestOK = TestOK && check_dump(ost, "DumpVisitor_Test", expected.str(), result.str());
179     }
180     catch (const string& err) {
181         error_msg = err;
182     }
183     catch (bad_alloc const& error) {
184         error_msg = error.what();
185     }
186     catch (const exception& err) {
187         error_msg = err.what();
188     }
189     catch (...) {
190         error_msg = "Unhandelt_Exception";
191     }
192
193     TestOK = TestOK && check_dump(ost, "Test_Exception_in_TestCase", true, error_msg.empty());
194     error_msg.clear();
195
196     try {
197         FSObjectFactory factory;
198         FSObject::Sptr root_folder = factory.CreateFolder("root");
199
200         stringstream result;
201
202         result.setstate(ios::badbit);
203
204         DumpVisitor dumper(result);
205
206         root_folder->Accept(dumper); // <= sould throw Exception bad Ostream
207
208     }
209     catch (const string& err) {
210         error_msg = err;
211     }
212     catch (bad_alloc const& error) {
213         error_msg = error.what();
214     }
215     catch (const exception& err) {
216         error_msg = err.what();
217     }
218     catch (...) {
219         error_msg = "Unhandelt_Exception";
220     }
221
222 }
```

```
223
224     TestOK = TestOK && check_dump(ost, "Test_Exception_Bad_Ostream_in_DumpVisitor", DumpVisitor::ERROR_BAD_OSTREAM, error_msg);
225     error_msg.clear();
226
227     ost << TestEnd;
228
229     return TestOK;
230 }
231
232 bool TestFilterLinkVisitor(ostream& ost)
233 {
234     assert(ost.good());
235
236     ost << TestStart;
237
238     bool TestOK = true;
239     string error_msg;
240
241
242     try {
243         FSObjectFactory factory;
244         FSObject::Sptr root_folder = factory.CreateFolder("root");
245         FSObject::Sptr sub_folder = factory.CreateFolder("sub_folder");
246         FSObject::Sptr sub_sub_folder = factory.CreateFolder("sub_sub_folder");
247         File::Sptr file = make_shared<File>("file1.txt", 2048);
248         Link::Sptr link = make_shared<Link>(file, "LinkToFile1");
249         sub_sub_folder->AsFolder()->Add(file);
250         sub_sub_folder->AsFolder()->Add(link);
251         sub_folder->AsFolder()->Add(sub_sub_folder);
252         root_folder->AsFolder()->Add(sub_folder);
253
254         FilterLinkVisitor link_filter;
255
256         root_folder->Accept(link_filter);
257
258         TestOK = TestOK && check_dump(ost, "FilterLinkVisitor_Test_filtered_amount", static_cast<size_t>(1), link_filter.GetFilteredObjectCount());
259         TestOK = TestOK && check_dump(ost, "FilterLinkVisitor_Test_filtered_obj", link->GetReferencedFSObject()->GetName(), link_filter.GetFilteredObjectCount());
260
261         stringstream result;
262         stringstream expected;
263
264         link_filter.DumpFiltered(result);
265
266         expected << "root\\sub_folder\\sub_sub_folder\\LinkToFile1_>_file1.txt" << std::endl;
267
268         TestOK = TestOK && check_dump(ost, "Filter_Link_Visitor_Test_Dump_", expected.str(), result.str());
269
270     }
271     catch (const string& err) {
272         error_msg = err;
273     }
274     catch (bad_alloc const& error) {
275         error_msg = error.what();
276     }
277     catch (const exception& err) {
278         error_msg = err.what();
279     }
280     catch (...) {
281         error_msg = "Unhandelt_Exception";
282     }
283
284     TestOK = TestOK && check_dump(ost, "Test_for_Exception_in_Testcase", true, error_msg.empty());
285     error_msg.clear();
286
287     try {
288
289         FilterLinkVisitor link_filter{};
290
291         stringstream result;
292         result.setstate(ios::badbit);
293
294         link_filter.DumpFiltered(result);
295     }
296     catch (const string& err) {
297         error_msg = err;
```

```

298     }
299     catch (bad_alloc const& error) {
300         error_msg = error.what();
301     }
302     catch (const exception& err) {
303         error_msg = err.what();
304     }
305     catch (...) {
306         error_msg = "Unhandelt_Exception";
307     }
308
309     TestOK = TestOK && check_dump(ost, "Test_for_Exception_in_Dump_with_bad_Ostream", error_msg, FilterLinkVisitor::ERROR_BAD_OSTREAM);
310     error_msg.clear();
311
312     ost << TestEnd;
313
314     return TestOK;
315 }
316
317 bool TestFilterFileVisitor(ostream& ost)
318 {
319     assert(ost.good());
320
321     ost << TestStart;
322
323     bool TestOK = true;
324     string error_msg;
325
326     try {
327         FSObjectFactory factory;
328         FSObject::Sptr root_folder = factory.CreateFolder("root");
329         FSObject::Sptr sub_folder = factory.CreateFolder("sub_folder");
330         FSObject::Sptr sub_sub_folder = factory.CreateFolder("sub_sub_folder");
331         File::Sptr file = make_shared<File>("file1.txt", 10);
332         File::Sptr file1 = make_shared<File>("file2.txt", 10);
333         File::Sptr file2 = make_shared<File>("file3.txt", 10);
334         File::Sptr file3 = make_shared<File>("file4.txt", 10);
335         Link::Sptr link = make_shared<Link>(file, "LinkToFile1");
336
337         file->Write(8192);
338         file1->Write(4096);
339         file2->Write(6000);
340         file3->Write(1000);
341
342         sub_sub_folder->AsFolder()->Add(file);
343         root_folder->AsFolder()->Add(file2);
344         sub_sub_folder->AsFolder()->Add(link);
345         sub_folder->AsFolder()->Add(sub_sub_folder);
346         sub_folder->AsFolder()->Add(file3);
347         root_folder->AsFolder()->Add(sub_folder);
348         root_folder->AsFolder()->Add(file1);
349
350         FilterFileVisitor file_filter(5000,9000);
351
352         root_folder->Accept(file_filter);
353
354         TestOK = TestOK && check_dump(ost, "FilterFileVisitor_Test_filtered_amount", static_cast<size_t>(2), file_filter.GetFilteredObjects());
355         TestOK = TestOK && check_dump(ost, "FilterFileVisitor_Test_for_filtered_file", file2->GetName(), file_filter.GetFilteredObjects());
356         TestOK = TestOK && check_dump(ost, "FilterFileVisitor_Test_for_filtered_file", file->GetName(), file_filter.GetFilteredObjects());
357
358         stringstream result;
359         stringstream expected;
360
361         file_filter.DumpFiltered(result);
362
363         expected << "root\\file3.txt" << std::endl
364                 << "root\\sub_folder\\sub_sub_folder\\file1.txt" << std::endl;
365
366         TestOK = TestOK && check_dump(ost, "Filter_File_Visitor_Test_Dump_", expected.str(), result.str());
367
368     }
369     catch (const string& err) {
370         error_msg = err;
371     }
372

```



```
373     }
374     catch (bad_alloc const& error) {
375         error_msg = error.what();
376     }
377     catch (const exception& err) {
378         error_msg = err.what();
379     }
380     catch (...) {
381         error_msg = "Unhandelt_Exception";
382     }
383
384     TestOK = TestOK && check_dump(ost, "Test_for_Exception_in_Testcase", true, error_msg.empty());
385     error_msg.clear();
386
387     try {
388
389         FilterFileVisitor file_filter{1,2};
390
391         stringstream result;
392         result.setstate(ios::badbit);
393
394         file_filter.DumpFiltered(result);
395     }
396     catch (const string& err) {
397         error_msg = err;
398     }
399     catch (bad_alloc const& error) {
400         error_msg = error.what();
401     }
402     catch (const exception& err) {
403         error_msg = err.what();
404     }
405     catch (...) {
406         error_msg = "Unhandelt_Exception";
407     }
408
409     TestOK = TestOK && check_dump(ost, "Test_for_Exception_in_Dump_with_bad_Ostream", error_msg, FilterLinkVisitor::ERROR_BAD_OSTREAM);
410     error_msg.clear();
411
412     try {
413
414         FilterFileVisitor file_filter{ 2,1 }; // <= should throw invalid size range
415     }
416     catch (const string& err) {
417         error_msg = err;
418     }
419     catch (bad_alloc const& error) {
420         error_msg = error.what();
421     }
422     catch (const exception& err) {
423         error_msg = err.what();
424     }
425     catch (...) {
426         error_msg = "Unhandelt_Exception";
427     }
428
429     TestOK = TestOK && check_dump(ost, "Test_for_Exception_in_Filter_File_Visiter_CTOR", error_msg, FilterFileVisitor::ERROR_INVALID_SI
430     error_msg.clear();
431
432
433     ost << TestEnd;
434
435     return TestOK;
436 }
437
438 bool TestVisitor(ostream& ost, IVisitor& visit)
439 {
440     assert(ost.good());
441
442     ost << TestStart;
443
444     bool TestOK = true;
445     string error_msg;
446
447
```

```
448     try {
449
450         stringstream result;
451
452         File::Sptr file = nullptr;
453
454         visit.Visit(file); // <= sould throw Exception Nullptr
455     }
456     catch (const string& err) {
457         error_msg = err;
458     }
459     catch (bad_alloc const& error) {
460         error_msg = error.what();
461     }
462     catch (const exception& err) {
463         error_msg = err.what();
464     }
465     catch (...) {
466         error_msg = "Unhandelt_Exception";
467     }
468
469     TestOK = TestOK && check_dump(ost, "Test_Exception_nullptr_in_Visit_File", DumpVisitor::ERROR_NULLPTR, error_msg);
470     error_msg.clear();
471
472     try {
473
474         stringstream result;
475
476         Folder::Sptr folder = nullptr;
477
478         visit.Visit(folder); // <= sould throw Exception Nullptr
479     }
480     catch (const string& err) {
481         error_msg = err;
482     }
483     catch (bad_alloc const& error) {
484         error_msg = error.what();
485     }
486     catch (const exception& err) {
487         error_msg = err.what();
488     }
489     catch (...) {
490         error_msg = "Unhandelt_Exception";
491     }
492
493     TestOK = TestOK && check_dump(ost, "Test_Exception_nullptr_in_Visit_Folder", DumpVisitor::ERROR_NULLPTR, error_msg);
494     error_msg.clear();
495
496     try {
497
498         stringstream result;
499
500         Link::Sptr lnk = nullptr;
501
502         visit.Visit(lnk); // <= sould throw Exception Nullptr
503     }
504     catch (const string& err) {
505         error_msg = err;
506     }
507     catch (bad_alloc const& error) {
508         error_msg = error.what();
509     }
510     catch (const exception& err) {
511         error_msg = err.what();
512     }
513     catch (...) {
514         error_msg = "Unhandelt_Exception";
515     }
516
517     TestOK = TestOK && check_dump(ost, "Test_Exception_nullptr_in_Visit_Link", DumpVisitor::ERROR_NULLPTR, error_msg);
518     error_msg.clear();
519
520
521
522
```

```
523     ost << TestEnd;
524
525     return TestOK;
526 }
527
528 bool TestFactory(ostream& ost)
529 {
530     assert(ost.good());
531
532     ost << TestStart;
533
534     bool TestOK = true;
535     string error_msg;
536
537     try {
538         FSObjectFactory fact;
539         FSObj_Sptr file = fact.CreateFile("file1.txt",10);
540         FSObj_Sptr folder = fact.CreateFolder();
541         FSObj_Sptr lnk = fact.CreateLink("link_to_file",file);
542
543
544         TestOK = TestOK && check_dump(ost, "Test_if_file_was_constructed", true, file != nullptr);
545         TestOK = TestOK && check_dump(ost, "Test_if_Link_was_constructed", true, lnk->AsLink() != nullptr);
546         TestOK = TestOK && check_dump(ost, "Test_if_Folder_was_constructed", true, folder->AsFolder() != nullptr);
547
548     }
549     catch (const string& err) {
550         error_msg = err;
551     }
552     catch (bad_alloc const& error) {
553         error_msg = error.what();
554     }
555     catch (const exception& err) {
556         error_msg = err.what();
557     }
558     catch (...) {
559         error_msg = "Unhandelt_Exception";
560     }
561
562     TestOK = TestOK && check_dump(ost, "Test_for_Exception_in_Tesstcase", true, error_msg.empty());
563     error_msg.clear();
564
565     try {
566         FSObjectFactory fact;
567         File::Sptr file= nullptr;
568         FSObj_Sptr Lnk = fact.CreateLink("Link_to_File", file);
569
570     }
571     catch (const string& err) {
572         error_msg = err;
573     }
574     catch (bad_alloc const& error) {
575         error_msg = error.what();
576     }
577     catch (const exception& err) {
578         error_msg = err.what();
579     }
580     catch (...) {
581         error_msg = "Unhandelt_Exception";
582     }
583
584     TestOK = TestOK && check_dump(ost, "Test_Exception_nullptr_CTOR_Link", Link::ERROR_NULLPTR, error_msg);
585     error_msg.clear();
586
587     ost << TestEnd;
588
589     return TestOK;
590 }
591 }
```

6.26 Test.hpp

```
1  /*****  
2  * \file   Test.hpp  
3  * \brief  File that provides a Test Function with a formatted output  
4  *  
5  * \author Simon  
6  * \date   April 2025  
7  *****/  
8  #ifndef TEST_HPP  
9  #define TEST_HPP  
10  
11 #include <string>  
12 #include <iostream>  
13 #include <vector>  
14 #include <list>  
15 #include <queue>  
16 #include <forward_list>  
17  
18 #define ON 1  
19 #define OFF 0  
20 #define COLOR_OUTPUT OFF  
21  
22 // Definitions of colors in order to change the color of the output stream.  
23 const std::string colorRed = "\x1B[31m";  
24 const std::string colorGreen = "\x1B[32m";  
25 const std::string colorWhite = "\x1B[37m";  
26  
27 inline std::ostream& RED(std::ostream& ost) {  
28     if (ost.good()) {  
29         ost << colorRed;  
30     }  
31     return ost;  
32 }  
33 inline std::ostream& GREEN(std::ostream& ost) {  
34     if (ost.good()) {  
35         ost << colorGreen;  
36     }  
37     return ost;  
38 }  
39 inline std::ostream& WHITE(std::ostream& ost) {  
40     if (ost.good()) {  
41         ost << colorWhite;  
42     }  
43     return ost;  
44 }  
45  
46 inline std::ostream& TestStart(std::ostream& ost) {  
47     if (ost.good()) {  
48         ost << std::endl;  
49         ost << "*****" << std::endl;  
50         ost << "TESTCASE_START" << std::endl;  
51         ost << "*****" << std::endl;  
52         ost << std::endl;  
53     }  
54     return ost;  
55 }  
56  
57 inline std::ostream& TestEnd(std::ostream& ost) {  
58     if (ost.good()) {  
59         ost << std::endl;  
60         ost << "*****" << std::endl;  
61         ost << std::endl;  
62     }  
63     return ost;  
64 }  
65  
66 inline std::ostream& TestCaseOK(std::ostream& ost) {  
67  
68     #if COLOR_OUTPUT  
69         if (ost.good()) {  
70             ost << colorGreen << "TEST_OK!!" << colorWhite << std::endl;  
71         }  
72     #else
```

```

73     if (ost.good()) {
74         ost << "TEST_OK!!" << std::endl;
75     }
76 #endif // COLOR_OUTPUT
77
78     return ost;
79 }
80
81 inline std::ostream& TestCaseFail(std::ostream& ost) {
82
83 #if COLOR_OUTPUT
84     if (ost.good()) {
85         ost << colorRed << "TEST_FAILED!!" << colorWhite << std::endl;
86     }
87 #else
88     if (ost.good()) {
89         ost << "TEST_FAILED!!" << std::endl;
90     }
91 #endif // COLOR_OUTPUT
92
93     return ost;
94 }
95
96 /**
97  * \brief function that reports if the testcase was successful.
98  *
99  * \param testcase String that indicates the testcase
100  * \param successful true -> reports to cout test OK
101  * \param successful false -> reports test failed
102  */
103 template <typename T>
104 bool check_dump(std::ostream& ostr, const std::string& testcase, const T& expected, const T& result) {
105     if (ostr.good()) {
106 #if COLOR_OUTPUT
107         if (expected == result) {
108             ostr << testcase << std::endl << colorGreen << "[Test_OK]" << colorWhite << "Result:_(Expected:_" << std::boolalpha << expected << std::noboolalpha << std::endl << std::endl;
109         }
110         else {
111             ostr << testcase << std::endl << colorRed << "[Test_FAILED]" << colorWhite << "Result:_(Expected:_" << std::boolalpha << expected << std::noboolalpha << std::endl << std::endl;
112         }
113 #else
114         if (expected == result) {
115             ostr << testcase << std::endl << "[Test_OK]" << "Result:_(Expected:_" << std::boolalpha << expected << "_" << "==" << "Result:_" << result << std::endl << std::endl;
116         }
117         else {
118             ostr << testcase << std::endl << "[Test_FAILED]" << "Result:_(Expected:_" << std::boolalpha << expected << "_" << "!=" << "Result:_" << result << std::endl << std::endl;
119         }
120 #endif
121     }
122     if (ostr.fail()) {
123         std::cerr << "Error:_Write_Ostream" << std::endl;
124     }
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::exception( "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::exception( "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::exception{ "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::exception{ "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::exception{ "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
```