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, 3. 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 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

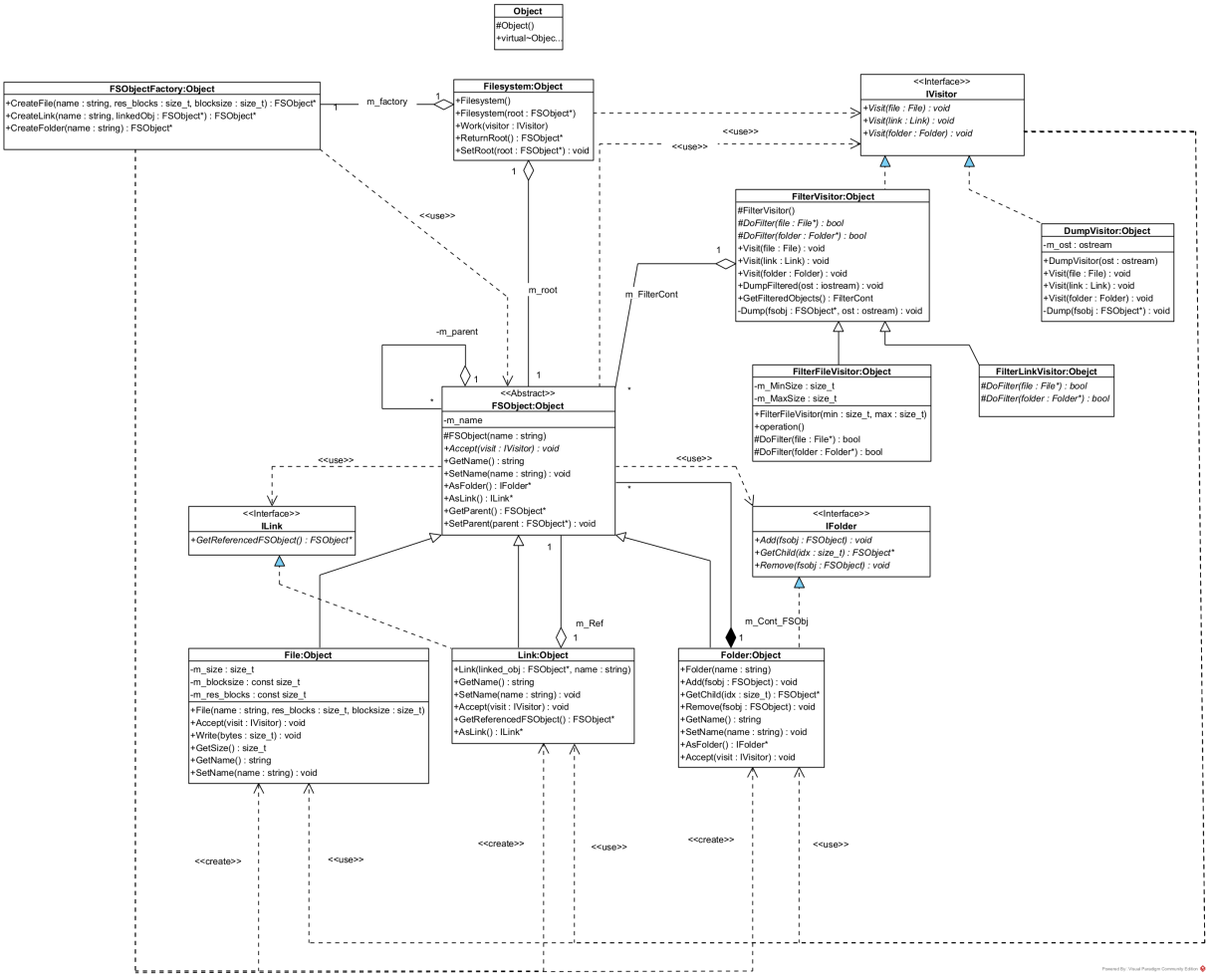
- Gibt die gesamte Dateisystemhierarchie aus.
- Ausgabe sowohl auf der Standardausgabe als auch in einer Datei möglich.
- Muss Dateien, Ordner und Verweise in strukturierter Form darstellen.

2.4.2 Visitor: FilterFiles

- Filtert Dateien anhand eines minimalen und maximalen 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 TEST OK!!
```

6 Quellcode

6.1 Object.hpp

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

#include <string>

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

#endif // OBJECT_H
```

6.2 FSObjectFactory.hpp

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

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


class FSObjectFactory : public Object
{
public:
        inline static const std::string ERROR_NULLPTR = "ERROR_Nullptr";

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

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

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

private:
};
#endif
```

6.3 FSObjectFactory.cpp

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

#include "FSObjectFactory.hpp"


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

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

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

6.4 Filesystem.hpp

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

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

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
         */
        IVisitor& 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);

private:
        FSObject::Sptr m_Root;
};
#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

```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";

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


```

```
73 private:
74         std::string m_Name;
75         FSObj_Wptr m_Parent;
76 };
77
78 #endif
```

6.7 FSObject.cpp

```cpp
#include "FSObject.hpp"
#include <string>

IFolder::Sptr FSObject::AsFolder()
{
    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)
{
    m_Name = name;
}

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

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

6.8 File.hpp

```
/*****************************************************************//**
 * \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";
    static inline 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
#include "File.hpp"

/** \brief Accept a visitor for this file */
void File::Accept(IVisitor& visit)
{
    visit.Visit(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 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) =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) 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 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
#include "Folder.hpp"

/** \brief Add child to folder, sets parent pointer on child */
void Folder::Add(FSObj_Sptr fsobj)
{
 if (fsobj == nullptr) throw 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)
{
 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 */
IFolder::Sptr Folder::AsFolder()
{
 return shared_from_this();
}

/** \brief Accept a visitor and forward to children */
void Folder::Accept(IVisitor& visit)
{
 visit.Visit(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
#include "Link.hpp"

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

    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(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<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

```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<FSObj_Wptr>;

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

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

        /** \brief Visit a link and apply filter
         * \param link Link to visit
         */
        virtual void Visit(const std::shared_ptr<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<File>& file)=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<Link>& link)=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 FSObj_Wptr& fsobj, std::ostream& ost) const;
76
77         TContFSobj m_FilterCont;
78 };
79
80 #endif
```

6.18 FilterVisitor.cpp

```
#include "FilterVisitor.hpp"
#include "Folder.hpp"
#include "File.hpp"
#include "Link.hpp"

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

using namespace std;

void FilterVisitor::DumpPath(const FSObj_Wptr & fsobj, std::ostream& ost) const
{
        if (fsobj.expired()) return;

        FSObject::Sptr obj = fsobj.lock();

        DumpPath(obj->GetParent(), ost);

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

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

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

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

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


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

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

```
73
74          for_each(m_FilterCont.cbegin(), m_FilterCont.cend(), [&](const auto & obj) {
75                  DumpPath(obj, ost);
76                  ost << "\n";
77          });
78  }
79
80  /** \brief Return the filtered objects container */
81  const FilterVisitor::TContFSobj& FilterVisitor::GetFilteredObjects() const
82  {
83          return m_FilterCont;
84  }
```

6.19 FilterFileVisitor.hpp

```cpp
/*****************************************************************//**
 * \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<File>& file) override;

        /** \brief Links are not accepted by this filter
         * \param link Link to check
         * \return false always
         */
        virtual bool DoFilter(const std::shared_ptr<Link>& link) 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
#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 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<File>& file)
{
	if (file == nullptr) throw 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<Link>& link)
{
	if (link == nullptr) throw 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<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

```
/*****************************************************************//**
 * \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 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<Folder>& folder) override;

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

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

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

#endif
```

6.24 DumpVisitor.cpp

```cpp
#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<Folder>& folder)
{
        if (m_ost.fail()) throw ERROR_BAD_OSTREAM;
        if (folder == nullptr) throw ERROR_NULLPTR;

        Dump(folder);
}

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

        Dump(file);
}

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

        Dump(Link);
}

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

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

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

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

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

6.25 main.cpp

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

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

using namespace std;

#define WriteOutputFile ON

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

int main()
{
        DumpVisitor visitor(std::cout);

        FilterLinkVisitor filter_link_visitor;

        FilterFileVisitor filter_file_visitor(4096, 16384);

        FSObjectFactory factory;

        FSObject::Sptr root_folder = factory.CreateFolder();
        IFolder::Sptr root_folder_ptr = root_folder->AsFolder();
        FSObject::Sptr sub_folder = factory.CreateFolder();
        IFolder::Sptr sub_folder_ptr = sub_folder->AsFolder();
        FSObject::Sptr sub_sub_folder = factory.CreateFolder();
        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(factory.CreateFile("file1.txt", 2048));
        root_folder_ptr->Add(factory.CreateFile("file2.txt", 2048));
        root_folder_ptr->Add(factory.CreateFile("file3.txt", 2048));
        root_folder_ptr->Add(factory.CreateFile("file4.txt", 2048));
        root_folder_ptr->Add(sub_folder);
        sub_folder_ptr->Add(factory.CreateFile("file5.txt", 8192));
        sub_folder_ptr->Add(factory.CreateFile("file6.txt", 32768));
        sub_folder_ptr->Add(sub_sub_folder);
        sub_sub_folder_ptr->Add(factory.CreateFile("file7.txt", 12288));
        sub_sub_folder_ptr->Add(factory.CreateLink("LinkToRoot", root_folder));


        FileSystem homework(move(root_folder));

        homework.Work(visitor);

```

```
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/]\n"
174                   << "|␣␣|---[sub_folder/]\n"
175                   << "|␣␣|␣␣|---[sub_sub_folder/]\n"
176                   << "|␣␣|␣␣|␣␣|---[file1.txt]\n";
177
178                 TestOK = TestOK && check_dump(ost, "DumpVisitor␣Test", expected.str(), result.str());
179
180     }
181     catch (const string& err) {
182         error_msg = err;
183     }
184     catch (bad_alloc const& error) {
185         error_msg = error.what();
186     }
187     catch (const exception& err) {
188         error_msg = err.what();
189     }
190     catch (...) {
191         error_msg = "Unhandelt␣Exception";
192     }
193
194     TestOK = TestOK && check_dump(ost, "Test␣Exception␣in␣TestCase", true, error_msg.empty());
195     error_msg.clear();
196
197     try {
198
199                 FSObjectFactory factory;
200                 FSObject::Sptr root_folder = factory.CreateFolder("root");
201
202                 stringstream result;
203
204                 result.setstate(ios::badbit);
205
206                 DumpVisitor dumper(result);
207
208                 root_folder->Accept(dumper); // <= sould throw Exception bad Ostream
209
210     }
211     catch (const string& err) {
212         error_msg = err;
213     }
214     catch (bad_alloc const& error) {
215         error_msg = error.what();
216     }
217     catch (const exception& err) {
218         error_msg = err.what();
219     }
220     catch (...) {
221         error_msg = "Unhandelt␣Exception";
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.GetFilteredObjects().size());
259                 TestOK = TestOK && check_dump(ost, "FilterLinkVisitor␣Test␣filtered␣obj", link->
                        GetReferncedFSObject()->GetName(), link_filter.GetFilteredObjects().cbegin()->lock
                        ()->AsLink()->GetReferncedFSObject()->GetName());
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
313     ost << TestEnd;
314
315     return TestOK;
316 }
317
318 bool TestFilterFileVisitor(ostream& ost)
319 {
320     assert(ost.good());
321
322     ost << TestStart;
323
324     bool TestOK = true;
325     string error_msg;
326
327
328     try {
329         FSObjectFactory factory;
330         FSObject::Sptr root_folder = factory.CreateFolder("root");
331         FSObject::Sptr sub_folder = factory.CreateFolder("sub_folder");
332         FSObject::Sptr sub_sub_folder = factory.CreateFolder("sub_sub_folder");
333         File::Sptr file = make_shared<File>("file1.txt", 10);
334         File::Sptr file1 = make_shared<File>("file2.txt", 10);
335         File::Sptr file2 = make_shared<File>("file3.txt", 10);
336         File::Sptr file3 = make_shared<File>("file4.txt", 10);
337         Link::Sptr link = make_shared<Link>(file, "LinkToFile1");
338
339                 file->Write(8192);
340                 file1->Write(4096);
341                 file2->Write(6000);
342                 file3->Write(1000);
343
344         sub_sub_folder->AsFolder()->Add(file);
345         root_folder->AsFolder()->Add(file2);
346         sub_sub_folder->AsFolder()->Add(link);
347         sub_folder->AsFolder()->Add(sub_sub_folder);
348         sub_folder->AsFolder()->Add(file3);
349         root_folder->AsFolder()->Add(sub_folder);
350         root_folder->AsFolder()->Add(file1);
351
352         FilterFileVisitor file_filter(5000,9000);
353
354         root_folder->Accept(file_filter);
355
356         TestOK = TestOK && check_dump(ost, "FilterFileVisitor Test filtered amount", static_cast<size_t
                >(2), file_filter.GetFilteredObjects().size());
357         TestOK = TestOK && check_dump(ost, "FilterFileVisitor Test for filtered file", file2->GetName()
                , file_filter.GetFilteredObjects().cbegin()->lock()->GetName());
358         TestOK = TestOK && check_dump(ost, "FilterFileVisitor Test for filtered file", file->GetName(),
                 file_filter.GetFilteredObjects().crbegin()->lock()->GetName());
359
360         stringstream result;
361         stringstream expected;
362
```

```
363         file_filter.DumpFiltered(result);
364 
365         expected << "\\root\\file3.txt" << std::endl
366                  << "\\root\\sub_folder\\sub_sub_folder\\file1.txt" << std::endl;
367 
368         TestOK = TestOK && check_dump(ost, "Filter␣File␣Visitor␣Test␣Dump␣", expected.str(), result.str
                ());
369 
370     }
371     catch (const string& err) {
372         error_msg = err;
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_SIZE_RANGE);
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     }
457     catch (const string& err) {
458         error_msg = err;
459     }
460     catch (bad_alloc const& error) {
461         error_msg = error.what();
462     }
463     catch (const exception& err) {
464         error_msg = err.what();
465     }
466     catch (...) {
467         error_msg = "Unhandelt Exception";
468     }
469
470     TestOK = TestOK && check_dump(ost, "Test Exception nullptr in Visit File", DumpVisitor::
          ERROR_NULLPTR, error_msg);
471     error_msg.clear();
472
473     try {
474
475         stringstream result;
476
477         Folder::Sptr folder = nullptr;
478
479         visit.Visit(folder); // <= sould throw Exception Nullptr
480
481     }
482     catch (const string& err) {
483         error_msg = err;
484     }
485     catch (bad_alloc const& error) {
486         error_msg = error.what();
487     }
488     catch (const exception& err) {
489         error_msg = err.what();
490     }
491     catch (...) {
492         error_msg = "Unhandelt Exception";
493     }
494
495     TestOK = TestOK && check_dump(ost, "Test Exception nullptr in Visit Folder", DumpVisitor::
          ERROR_NULLPTR, error_msg);
496     error_msg.clear();
497
498     try {
499
500         stringstream result;
501
502         Link::Sptr lnk = nullptr;
503
504         visit.Visit(lnk); // <= sould throw Exception Nullptr
505
506     }
507     catch (const string& err) {
```

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

```
580     }
581     catch (...) {
582         error_msg = "Unhandelt Exception";
583     }
584
585     TestOK = TestOK && check_dump(ost, "Test Exception nullptr CTOR Link", Link::ERROR_NULLPTR,
            error_msg);
586     error_msg.clear();
587
588     ost << TestEnd;
589
590     return TestOK;
591 }
```

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.
const std::string colorRed = "\x1B[31m";
const std::string colorGreen = "\x1B[32m";
const std::string colorWhite = "\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::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
```