[1 Allgemeine Bäume 2](#_Toc408952697)

[1.1 Lösungsidee 2](#_Toc408952698)

[1.2 Implementierung 3](#_Toc408952699)

[1.2.1 Tree.h 3](#_Toc408952700)

[1.2.2 Tree.cpp 6](#_Toc408952701)

[1.2.3 Node.h 10](#_Toc408952702)

[1.2.4 Node.cpp 12](#_Toc408952703)

[1.2.5 IntNode.h 13](#_Toc408952704)

[1.2.6 IntNode.cpp 15](#_Toc408952705)

[1.2.7 Shell Script – Tests 16](#_Toc408952706)

[1.3 Tests 16](#_Toc408952707)

[2 Hierarchisches Dateisystem 17](#_Toc408952708)

[2.1 Lösungsidee 17](#_Toc408952709)

[2.2 Implementierung 18](#_Toc408952710)

[2.2.1 FsNode.h 18](#_Toc408952711)

[2.2.2 FsNode.cpp 19](#_Toc408952712)

[2.2.3 Directory.h 20](#_Toc408952713)

[2.2.4 Directory.cpp 21](#_Toc408952714)

[2.2.5 File.h 21](#_Toc408952715)

[2.2.6 File.cpp 23](#_Toc408952716)

[2.2.7 FileSystem.h 24](#_Toc408952717)

[2.2.8 FileSystem.cpp 27](#_Toc408952718)

[2.2.9 FsErrorHandler.h 31](#_Toc408952719)

# Allgemeine Bäume

Folgend ist die Dokumentation für die Aufgabe Allgemeine Bäume angeführt.

## Lösungsidee

Für diesen Teil der Aufgabe sollen gemäß der vorgegebenen Spezifikation die Klassen implementiert werden. Da man bereits versucht hat mit inline Methoden zu arbeiten, diese aber es erschweren sich in der Implementierung zurecht zu finden, da immer in zwei Dateien \*.h und \*.cpp nachgeschaut werden muss, soll dieses Mal die Implementierung vollständig von der API getrennt werden. Diese erfordert mehr Schreibaufwand erhöht aber die Leserlichkeit der Implementierung.

Beide Klassen sollen um Methoden erweitert werden, die es erleichtern in dieser Baumstruktur zu navigieren. Auf jeden Fall soll darauf geachtet werden, dass diese nicht public sichtbar sind sondern höchstens protected, wenn man davon ausgehen kann das diese Methoden in einer abgeleiteten Klasse benötigt werden können und nicht die Implementierung (Bsp.: Tree) negativ beeinflussen. Die Member root und size sollen auf jeden Fall nur über Getter Methoden nach außen zugreifbar sein.

Alle Klassen sollen von der Klasse Object der MiniLib erben und über Register registriert werden, damit während der Implementierung getestet werden kann ob Instanzen dieser Klassen auch wieder freigegeben werden.

Die Klasse Node soll so implementiert werden dass ihre jeweiligen first child und next sibling Pointer Referenzen kaskadierend gelöscht sowie auch im Copy Konstruktor kaskadierend kopiert werden. Dies ist möglich da diese Art der Implementierung eines Baumes erlaubt. Eine besondere Aufmerksamkeit soll der Validierung des Baumes zukommen, da zyklische Referenzen hier zu einem Programmabsturz führen werden.

Da ein Baum immer einen Root Knoten besitzen muss aber in dieser Implementierung auch möglich sein soll einen eigenen Root zu definieren soll automatisch nur ein Knoten angelegt werden, wenn beim Konstruktor keiner mitgeliefert wird oder dieser root null ist. Wenn ein Tree kopiert wird, dann soll dieser Tree vollständig kopiert werden und keine besonderen Prüfungen vorgenommen werden (Bsp.: Kein Anlegen eines root Knoten wenn der kopierte Tree keinen besitzt)

Die zwei Methoden Clear sowie DeleteElements sollen sich wie folgt unterscheiden.

Clear:

Hier soll lediglich die Referenz auf den root Knoten gelöscht und ein neuer root Knoten angelegt werden. Hierbei ist der Aufrufer für die Instanzverwaltung verantwortlich.

DeleteElements:

Hier sollen alle Knoten gelöscht und ein neuer root Knoten angelegt werden.

Alle Referenzen die außerhalb gehalten werden zeigen dann auf eine ungültige Pointer Adresse. Es liegt in der Verantwortung des Aufrufers diese ungültigen Adressen nicht mehr zu dereferenzieren.

Die Konstruktoren der abgeleiteten Klassen sollen zu den Basisklassen delegieren und die bereits implementierte Funktionalität wieder verwenden und nicht sie erneut implementieren.

Alle Methoden die public oder protected sind sollen mit den Schlüsselwort virtual versehen werden um den abgeleiteten Klassen die Möglichkeit zu geben diese zu überschreiben.

Des Weiteren sollen alle Methoden ausführlich dokumentiert werden.

## Implementierung

Folgend sind die Sources der ersten Aufgabenstellung angeführt.

### Tree.h

Folgend ist die Spezifikation von Tree angeführt.

/\*

\* Tree.h

\*

\* Created on: Dec 23, 2014

\* Author: cchet

\*/

#ifndef TREE\_H\_

#define TREE\_H\_

#include <ostream>

#include "Node.h"

#include "MLObject.h"

#define TREE\_CLASS "Tree"

/\*\*

\* This class represents the tree which is able to handle all derivations of the Node class.

\*/

class Tree**:** public ML**::**Object **{**

private**:**

Node**\*** root**;**

int size**;**

////////////////////////////////////////////////////////////

// Private utils method //

////////////////////////////////////////////////////////////

/\*\*

\* Answers the question if the parent node is handled by this tree instance.

\* This is done by comparing the pointer addresses.

\*

\* **@param**

\* parent: the node to be checked if managed by this tree.

\* **@return**

\* true if the node is managed, false otherwise

\*/

bool isManagedNode**(**const Node**\*** node**)** const**;**

/\*\*

\* Counts the nodes of this subtree including the given parent node.

\*

\* **@param**

\* node: the node to count hold nodes

\* **@return**

\* the count of nodes in the subtree including the root.

\*/

int countNodes**(**const Node**\*** node**)** const**;**

/\*\*

\* Gets the parent node of the given node.

\* The given subTreeRoot represents the tree or subtree to search for the node and its parent

\*

\* **@param**

\* subTreeRoot: the subtree to search the neighbor of the given node

\* **@param**

\* node: the node to search parent for

\*/

Node**\*** getParentNode**(**Node**\*** subTreeRoot**,** const Node**\*** node**)** const**;**

/\*\*

\* Gets the former neighbor if the parent has the node referenced by its next sibling field.

\*

\* **@param**

\* parent: the subtree to search the neighbor of the given node

\* **@param**

\* node: the node to search former neighbor for

\*/

Node**\*** getFormerNeighbour**(**Node**\*** parent**,** const Node**\*** node**)** const**;**

public**:**

////////////////////////////////////////////////////////////

// Constructor and Destructor //

////////////////////////////////////////////////////////////

/\*\*

\* Default constructor which creates a new root node for this tree.

\*/

Tree**();**

/\*\*

\* Copy constructor for this class

\*/

Tree**(**Node **&** root**);**

/\*\*

\* Creates a tree with the given node as root.

\* If the root node is null then a root must be set via setRoot otherwise no children can be inserted.

\*/

Tree**(**Node**\*** root**);**

/\*\*

\* Copies the whole tree held by the root node by copying the root node.

\* The held nodes copy their held referenced nodes and so the whole tree is copied.

\*

\* **@param**

\* other: the tree to be copied.

\*/

Tree**(**const Tree **&** other**);**

/\*\*

\* The deconstructor of this class.

\* The nodes are deleted via cascade which starts at the root node.

\*/

virtual **~**Tree**();**

////////////////////////////////////////////////////////////

// Getter and Setter //

////////////////////////////////////////////////////////////

/\*\*

\* Getter method for the root node.

\*

\* **@return** the root node.

\*/

virtual Node**\*** getRoot**()** const**;**

/\*\*

\* Setter for root node.

\*

\* **@param**

\* node: the node acting as the root node.

\*/

virtual void setRoot**(**Node**\*** node**);**

/\*\*

\* The getter for the current size of the tree.

\*/

virtual int getSize**()** const**;**

////////////////////////////////////////////////////////////

// Tree Manipulation //

////////////////////////////////////////////////////////////

/\*\*

\* Inserts a child node for the given parent if the parent is a managed node and the child is not null.

\*

\* **@param**

\* parent: the parent node to append child on

\* **@param**

\* child:t he child node to be appended to the given parent node.

\*/

virtual void insertChild**(**Node**\*** parent**,** Node**\*** child**);**

/\*\*

\* Deletes the subtree held by teh given node.

\*

\* **@param**

\* node: the node to be deleted along with its held nodes.

\*/

virtual void deleteSubTree**(**Node**\*** node**);**

/\*\*

\* Clears the tree by removing the root references and creating a new root.

\* Be aware that the caller would be responsible for handling the lifecycle of the formerly managed tree nodes.

\*/

virtual void clear**();**

/\*\*

\* Deletes all elements including the root.

\* All nodes will not be available and all held references will become invalid.

\*/

virtual void deleteElements**();**

////////////////////////////////////////////////////////////

// Utils //

////////////////////////////////////////////////////////////

/\*\*

\* Prints this tree and all of its held nodes.

\*

\* **@param**

\* os: the ostram isntance ot print the result.

\*/

virtual void print**(**std**::**ostream **&** os**)** const**;**

////////////////////////////////////////////////////////////

// Operators //

////////////////////////////////////////////////////////////

/\*\*

\* The friend method which handles the operator <<.

\*

\* **@param**

\* os: the ostream where to put result on

\* **@param**

\* tree: the tree part of this operation

\*/

friend std**::**ostream**&** **operator<<(**std**::**ostream **&** os**,** const Tree **&** tree**);**

/\*\*

\* Assigns the tree to the current Tree instance by performing a deep copy.

\*

\* **@param**

\* other: the tree to be assigned

\*/

Tree **&** **operator=(**const Tree **&** other**);**

**};**

#endif /\* TREE\_H\_ \*/

### Tree.cpp

Folgend ist die Implementierung der Spezifikation Tree.h angeführt.

/\*

\* Tree.cpp

\*

\* Created on: Dec 23, 2014

\* Author: cchet

\*/

#include <string>

#include <iostream>

#include <vector>

#include <map>

#include "Tree.h"

#include "Node.h"

**using** **namespace** std**;**

**using** **namespace** ML**;**

////////////////////////////////////////////////////////////

// Private Utils //

////////////////////////////////////////////////////////////

static void buildLeveledMap**(**Node**\*** node**,** int level**,**

map**<**int**,** vector**<**Node**\*>>** **&** resultMap**)** **{**

/\* Only if the node is null or has first child \*/

**if** **((**node **!=** nullptr**)** **&&** **(**node**->**getFirstChild**()))** **{**

vector**<**Node**\*>** parents**;**

Node**\*** next **=** node**->**getFirstChild**();**

/\* get all nodes with first child set \*/

**while** **(**next **!=** nullptr**)** **{**

**if** **(**next**->**getFirstChild**()** **!=** nullptr**)** **{**

parents**.**push\_back**(**next**);**

**}** /\* if \*/

next **=** next**->**getNextSibling**();**

**}** /\* while \*/

/\* crawl deeper into tree if nodes were found \*/

**if** **(**parents**.**size**()** **>** 0**)** **{**

resultMap**.**insert**(**pair**<**int**,** vector**<**Node**\*>>(**level**,** parents**));**

level**++;**

**for** **(**unsigned int i **=** 0**;** i **<** parents**.**size**();** i**++)** **{**

buildLeveledMap**(**parents**[**i**],** level**,** resultMap**);**

**}** /\* for \*/

**}** /\* if \*/

**}** /\* if \*/

**}** /\* Tree::buildLeveledMap \*/

bool Tree**::**isManagedNode**(**const Node**\*** node**)** const **{**

bool childResult **=** **false;**

bool siblingResult **=** **false;**

Node**\*** child**;**

Node**\*** next**;**

**if** **((**node **!=** nullptr**)** **&&** **(!(**childResult **=** **(**root **==** node**))))** **{**

child **=** root**->**getFirstChild**();**

**while** **((**child **!=** nullptr**)** **&&** **(!**siblingResult**)**

**&&** **(!(**childResult **=** **(**child **==** node**))))** **{**

siblingResult **=** **false;**

next **=** child**->**getNextSibling**();**

**while** **((**next **!=** nullptr**)** **&&** **((!(**siblingResult **=** **(**next **==** node**)))))** **{**

next **=** next**->**getNextSibling**();**

**}** /\* while \*/

child **=** child**->**getFirstChild**();**

**}** /\* if \*/

**}** /\* if \*/

**return** childResult **||** siblingResult**;**

**}** /\* Tree::isManagedNode \*/

int Tree**::**countNodes**(**const Node**\*** node**)** const **{**

int result **=** 0**;**

**if** **(**node **!=** nullptr**)** **{**

result**++;**

Node**\*** tmp **=** node**->**getNextSibling**();**

**while** **(**tmp **!=** nullptr**)** **{**

tmp **=** tmp**->**getNextSibling**();**

result**++;**

**}** /\* while \*/

**return** result **+** countNodes**(**node**->**getFirstChild**());**

**}** /\* if \*/

**return** result**;**

**}** /\* Tree::countNodes \*/

Node**\*** Tree**::**getParentNode**(**Node**\*** subTreeRoot**,** const Node**\*** node**)** const **{**

Node**\*** result **=** nullptr**;**

Node**\*** next**;**

**if** **((**subTreeRoot **!=** nullptr**)** **&&** **(**node **!=** nullptr**))** **{**

result **=** **(**subTreeRoot**->**getFirstChild**()** **==** node**)** **?** subTreeRoot **:** nullptr**;**

next **=** subTreeRoot**->**getNextSibling**();**

**while** **((**next **!=** nullptr**)** **&&** **(**result **==** nullptr**))** **{**

result **=** getParentNode**(**next**->**getFirstChild**(),** node**);**

next **=** next**->**getNextSibling**();**

**}** /\* while \*/

**if** **(**result **==** nullptr**)** **{**

result **=** getParentNode**(**subTreeRoot**->**getFirstChild**(),** node**);**

**}**

**}** /\* if \*/

**return** result**;**

**}** /\* Tree::getParentNode \*/

Node**\*** Tree**::**getFormerNeighbour**(**Node**\*** parent**,** const Node**\*** node**)** const **{**

Node**\*** result **=** nullptr**;**

**if** **((**parent **!=** nullptr**)** **&&** **(**node **!=** nullptr**))** **{**

Node**\*** pre **=** parent**;**

Node**\*** cur **=** parent**->**getNextSibling**();**

**while** **((**cur **!=** nullptr**)** **&&** **(**cur **!=** node**))** **{**

pre **=** cur**;**

cur **=** cur**->**getNextSibling**();**

**}** /\* while \*/

result **=**

**(**cur **==** node**)** **?**

pre **:** getFormerNeighbour**(**parent**->**getFirstChild**(),** node**);**

**}** /\* if \*/

**return** result**;**

**}** /\* Tree::getFormerNeighbour \*/

////////////////////////////////////////////////////////////

// Constructor and Destructor //

////////////////////////////////////////////////////////////

Tree**::**Tree**()** **:**

Tree**(**nullptr**)** **{**

**}** /\* Tree::Tree \*/

Tree**::**Tree**(**Node **&** root**)** **:**

Tree**(&**root**)** **{**

**}** /\* Tree::Tree \*/

Tree**::**Tree**(**Node**\*** root**)** **:**

root**(**root**)** **{**

**if** **(**root **!=** nullptr**)** **{**

size **=** countNodes**(**root**);**

**}**

Register**(**TREE\_CLASS**,** OBJECT\_CLASS**);**

**}** /\* Tree::Tree \*/

Tree**::**Tree**(**const Tree **&** other**)** **:**

size**(**other**.**getSize**())** **{**

**if** **(**other**.**getRoot**()** **!=** nullptr**)** **{**

root **=** other**.**getRoot**()->**clone**();**

**}**

Register**(**TREE\_CLASS**,** OBJECT\_CLASS**);**

**}** /\* Tree::Tree \*/

Tree**::~**Tree**()** **{**

**if** **(**root **!=** nullptr**)** **{**

**delete** root**;**

**}**

**}** /\* Tree::~Tree \*/

////////////////////////////////////////////////////////////

// Getter and Setter //

////////////////////////////////////////////////////////////

Node**\*** Tree**::**getRoot**()** const **{**

**return** root**;**

**}** /\* Tree::getRoot \*/

void Tree**::**setRoot**(**Node**\*** node**)** **{**

/\* check for valid new root node \*/

**if** **((**node **==** nullptr**)** **||** **(**node**->**getNextSibling**()** **!=** nullptr**))** **{**

cout

**<<** "root node must not be null and must not have next sibling set !!!"

**<<** endl**;**

**return;**

**}**

/\* if current root node is not null delete it \*/

**else** **if** **(**root **!=** nullptr**)** **{**

deleteElements**();**

**}**

/\* set new root node \*/

clear**();**

**}** /\* Tree::setRoot \*/

int Tree**::**getSize**()** const **{**

**return** size**;**

**}** /\* Tree::getSize \*/

////////////////////////////////////////////////////////////

// Tree Manipulation //

////////////////////////////////////////////////////////////

void Tree**::**insertChild**(**Node**\*** parent**,** Node**\*** child**)** **{**

/\* Check if root is defined \*/

**if** **(**root **==** nullptr**)** **{**

cout

**<<** "Cannot add child on empty tree !!! At least the root node must be defined !!!"

**<<** endl **<<** flush**;**

**}**

/\* Check for nullptr nodes \*/

**if** **((**parent **==** nullptr**)** **||** **(**child **==** nullptr**))** **{**

cout **<<** "Parent and child not are not allowed to be null" **<<** endl

**<<** flush**;**

**return;**

**}** /\* if \*/

/\* Check for recursions \*/

**if** **(**parent **==** child**)** **{**

cout **<<** "Parent and child are not allowed to point to the same node !!!"

**<<** endl**;**

cout **<<** "parent: " **<<** parent**->**AsString**()** **<<** endl**;**

cout **<<** "child: " **<<** child**->**AsString**()** **<<** endl**;**

cout **<<** endl **<<** flush**;**

**return;**

**}** /\* if \*/

/\* Check if parent node is managed by this tree \*/

**if** **(!**isManagedNode**(**parent**))** **{**

cout **<<** "Parent is not managed by this tree" **<<** endl **<<** flush**;**

**return;**

**}** /\* if \*/

/\* set as first child if no first child present \*/

**if** **(**parent**->**getFirstChild**()** **==** nullptr**)** **{**

parent**->**setFirstChild**(**child**);**

**}**

/\* insert as new first child and move current first child to next sibling of new first child \*/

**else** **{**

child**->**setNextSibling**(**parent**->**getFirstChild**());**

parent**->**setFirstChild**(**child**);**

**}** /\* if \*/

size**++;**

**}** /\* Tree::insertChild \*/

void Tree**::**deleteSubTree**(**Node**\*** node**)** **{**

bool isRoot **=** **false;**

Node**\*** tmp **=** nullptr**;**

/\* Error if instance is not managed \*/

**if** **(!**isManagedNode**(**node**))** **{**

cout **<<** "Cannot delete unmanaged subtree !!! node: " **<<** node**->**AsString**()**

**<<** endl **<<** flush**;**

**}** **else** **{**

/\* check if root node \*/

**if** **(**root **==** node**)** **{**

isRoot **=** **true;**

**}**

/\* handle references to sub tree root \*/

**else** **{**

/\* cut from first child list if found here \*/

**if** **((**tmp **=** getParentNode**(**root**,** node**))** **!=** nullptr**)** **{**

tmp**->**setFirstChild**(**node**->**getNextSibling**());**

**}**

/\* Cut from sibling list if found there \*/

**else** **if** **((**tmp **=** getFormerNeighbour**(**root**,** node**))** **!=** nullptr**)** **{**

tmp**->**setNextSibling**(**tmp**->**getNextSibling**()->**getNextSibling**());**

**}** /\* if \*/

/\* need to remove reference because otherwise would be deleted along with the node \*/

node**->**setNextSibling**(**nullptr**);**

**}**/\* if \*/

/\* delete the sub tree \*/

**delete** node**;**

node **=** nullptr**;**

/\* re initialize if it is root \*/

**if** **(**isRoot**)** **{**

clear**();**

**}** **else** **{**

size **=** countNodes**(**root**);**

**}**/\* if \*/

**}** /\* if \*/

**}** /\* Tree::deleteSubTree \*/

void Tree**::**deleteElements**()** **{**

deleteSubTree**(**root**);**

**}** /\* Tree::deleteElements \*/

void Tree**::**clear**()** **{**

root **=** **new** Node**();**

size **=** 1**;**

**}** /\* Tree::clear \*/

////////////////////////////////////////////////////////////

// Utils //

////////////////////////////////////////////////////////////

void Tree**::**print**(**ostream **&** os**)** const **{**

Node**\*** node **=** root**;**

vector**<**Node**\*>** firstLevel**;**

map**<**int**,** vector**<**Node**\*>>** leveledMap**;**

/\* header including root node \*/

os **<<** "##################################################" **<<** endl**;**

os **<<** "node count: " **<<** size **<<** endl**;**

os **<<** "##################################################" **<<** endl**;**

os **<<** "Level: 0" **<<** endl**;**

os **<<** "Parent: (root has not parent)" **<<** endl**;**

os **<<** "Nodes: " **<<** **\***root **<<** endl **<<** endl**;**

/\* build root map for each level under root \*/

node **=** root**;**

firstLevel**.**push\_back**(**root**);**

leveledMap**.**insert**(**pair**<**int**,** vector**<**Node**\*>>(**1**,** firstLevel**));**

buildLeveledMap**(**node**,** 2**,** leveledMap**);**

/\* iterate over each level \*/

map**<**int**,** vector**<**Node**\*>>::**iterator it **=** leveledMap**.**begin**();**

**while** **(**it **!=** leveledMap**.**end**())** **{**

os **<<** "-----------------------------------------------------" **<<** endl**;**

os **<<** "Level: " **<<** it**->**first **<<** endl**;**

os **<<** "-----------------------------------------------------" **<<** endl**;**

/\* print all nodes and their parents of the current level \*/

vector**<**Node**\*>::**iterator parentIterator **=** it**->**second**.**begin**();**

**while** **(**parentIterator **!=** it**->**second**.**end**())** **{**

os **<<** "Parent: " **<<** **\*\***parentIterator **<<** endl**;**

os **<<** "Nodes: "**;**

node **=** **(\***parentIterator**)->**getFirstChild**();**

**while** **(**node **!=** nullptr**)** **{**

os **<<** **\***node**;**

**if** **(**node**->**getNextSibling**()** **!=** nullptr**)** **{**

os **<<** " - "**;**

**}** /\* if \*/

node **=** node**->**getNextSibling**();**

**}** /\* while \*/

parentIterator**++;**

**}** /\* while \*/

os **<<** endl **<<** flush**;**

it**++;**

**}** /\* while \*/

**}** /\* Tree::print \*/

////////////////////////////////////////////////////////////

// Operators //

////////////////////////////////////////////////////////////

ostream**&** **operator<<(**ostream **&** os**,** const Tree **&** tree**)** **{**

tree**.**print**(**os**);**

**return** os**;**

**}** /\* operator<< \*/

Tree **&** Tree**::operator=(**const Tree **&** other**)** **{**

**if** **(this** **!=** **&**other**)** **{**

/\* delete current root \*/

**if** **(**root **!=** nullptr**)** **{**

**delete** root**;**

**}**

root **=** other**.**getRoot**()->**clone**();**

size **=** other**.**getSize**();**

**}**

**return** **\*this;**

**}** /\* Tree::operator= \*/

### Node.h

Folgend ist die Spezifikation von Node nageführt.

/\*

\* Node.h

\* This is the specification of the IntNode

\*

\* Created on: Dec 23, 2014

\* Author: Thomas Herzog

\*/

#ifndef NODE\_H\_

#define NODE\_H\_

#include "MLObject.h"

#include "MetaInfo.h"

#define NODE\_CLASS "Node"

#define OBJECT\_CLASS "Object"

/\*\*

\* This class is the base node which can be used to be handled in a tree.

\* This class provides all necessary methods which a tree node needs.

\*/

class Node**:** public ML**::**Object **{**

protected**:**

Node **\***firstChild**;**

Node **\***nextSibling**;**

public**:**

////////////////////////////////////////////////////////////

// Constructor and Destructor //

////////////////////////////////////////////////////////////

/\*\*

\* Default constructor which sets the first child and next sibling to null.

\*

\* **@param**

\* firstChild: nullptr for the first child reference

\* **@param**

\* nextSibling: nullptr for the first nextSibling reference

\*/

explicit inline Node**(**Node **\***firstChild **=** nullptr**,** Node **\***nextSibling **=**

nullptr**)** **:**

firstChild**(**nullptr**),** nextSibling**(**nullptr**)** **{**

Register**(**NODE\_CLASS**,** OBJECT\_CLASS**);**

**}**

/\*\*

\* Destructor which deletes the held references if they are not null.

\* Their references nodes will be deleted same as here when their destructor gets called.

\*/

virtual **~**Node**();**

/\*\*

\* This constructor copies the held nodes which causes that the whole subtree

\* held by this node is copied.

\* Therefore that this is a tree node this is no problem,

\* because recursions are not allowed here.

\*

\* **@param**

\* node: the node to be copied

\*/

Node**(**const Node**&** other**);**

////////////////////////////////////////////////////////////

// Getter and Setter //

////////////////////////////////////////////////////////////

/\*\*

\* Getter method for the first child references.

\*

\* **@return**

\* the first child reference

\*/

virtual Node**\*** getFirstChild**()** const**;**

/\*\*

\* Getter method for the first next neighbor references.

\*

\* **@return**

\* the next sibling reference

\*/

virtual Node**\*** getNextSibling**()** const**;**

/\*\*

\* Sets the first child references.

\* The caller is responsible for handling the instance lifecycle of the former referenced node.

\*

\* **@param**

\* firstChild: the new first child node reference

\*/

virtual void setFirstChild**(**Node**\*** firstChild**);**

/\*\*

\* Sets the next neighbor references.

\* The caller is responsible for handling the instance lifecycle of the former referenced node.

\*

\* **@param**

\* nextSibling: the new nextSibling reference

\*/

virtual void setNextSibling**(**Node**\*** nextSibling**);**

////////////////////////////////////////////////////////////

// Utils //

////////////////////////////////////////////////////////////

/\*\*

\* Prints the node and all of its referenced nodes.

\* Means the subtree held by this node will be printed.

\*

\* **@param**

\* ostram: the ostram to put printed text on

\*/

virtual void print**(**std**::**ostream **&** os**)** const**;**

/\*\*

\* This method clones this node by copying the held references of this node.

\*

\* **@return**

\* A deep copy of this node

\*/

virtual Node**\*** clone**()** const**;**

////////////////////////////////////////////////////////////

// operator //

////////////////////////////////////////////////////////////

/\*\*

\* The friend method for the << operator.

\*

\* **@param**

\* os: the ostream instance to print result on

\*

\* **@param**

\* node: the node part of the operation

\*/

friend std**::**ostream**&** **operator<<(**std**::**ostream **&** os**,** const Node **&** node**);**

**};**

#endif /\* NODE\_H\_ \*/

### Node.cpp

Folgen ist die Implementierung der Spezifikation Node.h angeführt.

/\*

\* Node.cpp

\* This is the implementation of the IntNode specification.

\*

\* Created on: Jan 11, 2015

\* Author: Thomas Herzog

\*/

#include "MLObject.h"

#include "MetaInfo.h"

#include "Node.h"

**using** **namespace** std**;**

**using** **namespace** ML**;**

////////////////////////////////////////////////////////////

// Constructor and Destructor //

////////////////////////////////////////////////////////////

Node**::~**Node**()** **{**

**if** **(**firstChild **!=** nullptr**)** **{**

**delete** firstChild**;**

**}** /\* if \*/

**if** **(**nextSibling **!=** nullptr**)** **{**

**delete** nextSibling**;**

**}** /\* if \*/

**}**

Node**::**Node**(**const Node**&** other**)** **:**

firstChild**(**nullptr**),** nextSibling**(**nullptr**)** **{**

**if** **(**other**.**getFirstChild**()** **!=** nullptr**)** **{**

setFirstChild**(**other**.**getFirstChild**()->**clone**());**

**}**

**if** **(**other**.**getNextSibling**()** **!=** nullptr**)** **{**

setNextSibling**(**other**.**getNextSibling**()->**clone**());**

**}**

Register**(**NODE\_CLASS**,** OBJECT\_CLASS**);**

**}**

////////////////////////////////////////////////////////////

// Getter and Setter //

////////////////////////////////////////////////////////////

Node**\*** Node**::**getFirstChild**()** const **{**

**return** firstChild**;**

**}**

Node**\*** Node**::**getNextSibling**()** const **{**

**return** nextSibling**;**

**}**

void Node**::**setFirstChild**(**Node**\*** firstChild**)** **{**

**if** **(this** **==** firstChild**)** **{**

cout **<<** "Node is not allowed to reference itself on firstChild"

**<<** flush **<<** endl**;**

**}** **else** **{**

**this->**firstChild **=** firstChild**;**

**}** /\* if \*/

**}**

void Node**::**setNextSibling**(**Node**\*** nextSibling**)** **{**

**if** **(this** **==** nextSibling**)** **{**

cout **<<** "Node is not allowed to reference itself on nextSibling"

**<<** flush **<<** endl**;**

**}** **else** **{**

**this->**nextSibling **=** nextSibling**;**

**}** /\* if \*/

**}**

////////////////////////////////////////////////////////////

// Utils //

////////////////////////////////////////////////////////////

void Node**::**print**(**ostream **&** os**)** const **{**

os **<<** AsString**();**

**}**

Node**\*** Node**::**clone**()** const **{**

**return** **new** Node**(\*this);**

**}**

////////////////////////////////////////////////////////////

// operator //

////////////////////////////////////////////////////////////

ostream**&** **operator<<(**ostream **&** os**,** const Node **&** node**)** **{**

node**.**print**(**os**);**

**return** os**;**

**}**

### IntNode.h

Folgend ist die Spezifikation von IntNode angeführt.

/\*

\* IntNode.h

\* The specification for the IntNode.

\*

\* Created on: Dec 23, 2014

\* Author: Thomas Herzog

\*/

#ifndef INTNODE\_H\_

#define INTNODE\_H\_

#include "Node.h"

#define INT\_NODE\_CLASS "IntNode"

/\*\*

\* This class inherit Node class and defines a int value which can be accessed via its defined getter method.

\*

\*/

class IntNode**:** public Node **{**

private**:**

////////////////////////////////////////////////////////////

// Private Members //

////////////////////////////////////////////////////////////

int value**;**

public**:**

////////////////////////////////////////////////////////////

// Constructor and Destructor //

////////////////////////////////////////////////////////////

/\*\*

\* Constructor which takes value as the argument

\*

\* **@param**

\* value: the value to be hold by this IntNode

\*/

IntNode**(**int value**);**

/\*\*

\* Copy constructor which clones the IntNode deeply.

\* All references are cloned to, so the whole subtree will be copied.

\*

\* **@param**

\* other: the node to be copied deeply

\*/

IntNode**(**const IntNode **&** other**);**

/\*\*

\* The destructor of this class which does nothing because deletion is performed via Node

\* implemented destructor.

\*/

virtual **~**IntNode**();**

////////////////////////////////////////////////////////////

// Getter and Setter //

////////////////////////////////////////////////////////////

/\*\*

\* Gets the hold value,

\*

\* **@return**

\* the hold value

\*/

virtual int getValue**()** const**;**

/\*\*

\* Sets a new value.

\*

\* **@param**

\* value: the value to bet set

\*/

virtual void setValue**(**int value**);**

/\*\*

\* Returns a string representation of this class.

\*

\* **@return**

\* the string representation of this class

\*/

virtual std**::**string AsString**()** const**;**

////////////////////////////////////////////////////////////

// Utils //

////////////////////////////////////////////////////////////

/\*\*

\* This method clones this node by copying the held references of this node.

\*

\* **@return**

\* A deep copy of this node

\*/

virtual IntNode**\*** clone**()** const**;**

**};**

#endif /\* INTNODE\_H\_ \*/

### IntNode.cpp

Folgend ist die Spezifikation von IntNode.h angeführt.

/\*

\* IntNode.cpp

\* This is the implementation of the IntNode specification.

\*

\* Created on: Jan 11, 2015

\* Author: Thomas Herzog

\*/

#include <ostream>

#include <string>

#include <sstream>

#include "IntNode.h"

#include "MetaInfo.h"

**using** **namespace** std**;**

**using** **namespace** ML**;**

////////////////////////////////////////////////////////////

// Constructor and Destructor //

////////////////////////////////////////////////////////////

IntNode**::**IntNode**(**int value**)** **:**

Node**(),** value**(**value**)** **{**

Register**(**INT\_NODE\_CLASS**,** NODE\_CLASS**);**

**}** /\* IntNode::IntNode \*/

IntNode**::**IntNode**(**const IntNode **&** other**)** **:**

Node**(**other**),** value**(**other**.**getValue**())** **{**

Register**(**INT\_NODE\_CLASS**,** NODE\_CLASS**);**

**}** /\* IntNode::IntNode \*/

IntNode**::~**IntNode**()** **{**

**}** /\* IntNode::~IntNode \*/

////////////////////////////////////////////////////////////

// Getter and Setter //

////////////////////////////////////////////////////////////

int IntNode**::**getValue**()** const **{**

**return** value**;**

**}** /\* IntNode::getValue \*/

void IntNode**::**setValue**(**int value**)** **{**

**this->**value **=** value**;**

**}** /\* IntNode::setValue \*/

string IntNode**::**AsString**()** const **{**

stringstream ss**;**

ss **<<** "IntNode(" **<<** value **<<** ")" **<<** flush**;**

**return** ss**.**str**();**

**}** /\* IntNode::AsString \*/

////////////////////////////////////////////////////////////

// Utils //

////////////////////////////////////////////////////////////

IntNode**\*** IntNode**::**clone**()** const **{**

**return** **new** IntNode**(\*this);**

**}** /\* IntNode::clone \*/

### Shell Script – Tests

## Tests

# Hierarchisches Dateisystem

Folgend ist die Dokumentation der Aufgabe Hierarchisches Dateisystem angeführt.

## Lösungsidee

Bei dieser Implementierung soll von der Klasse Tree abgeleitet werden.

Da diese Klasse aber Methoden als public definiert, die nicht bei einer FileSystem Implementierung sichtbar sein sollen (insertChild(…), deleteSubTree(…),…), sollen diese Methoden von dem public Bereich in den private Bereich verschoben werden. Dies ist möglich da die Klasse FileSystem eigene Methoden zur Manipulation des Baumes zur Verfügung stellt, die wiederum diese in Tree implementierten Methoden nutzt.

Da hier auch gegen eigene Datentypen implementiert wird (Directory, File, FsNode) sollen die Datentypen der Methoden so abgeändert werden, dass keine Typprobleme auftreten können.

Als Beispiel sei hier die Methode insertChild(Node\* parent, Node\* child angeführt).

In einem FileSystem kann der Parent immer nur ein Verzeichnis sein, das Kind aber ein File oder ein Verzeichnis. Daher soll der Datentyp des Parent auf Directory\* und der Datentyp des Kindes auf FSnode\* abgeändert werden. Somit können hier keine Typprobleme auftreten. Des Weiteren könnten, wenn die Datentypen auf Node\* verbleiben würden, jeder abgeleitete Typ verwendet werden. Also auch IntNode\*, was aber zu verhindern ist, da dieser Typ in einem FileSystem nichts zu suchen hat.

Es sollen also Datentypen der Formalparameter so abgeändert das keine Typprobleme auftreten können. Da diese Datentypen in der Vererbungshierarchie von Node\* liegen gibt es keine Probleme bei der Delegation von Methoden an die Basisklasse Tree.

Bei den casts soll dynamic\_cast verwendet werden um ungültige Pointertypen zu vermeiden. Mit nullptr ist leichter umzugehen als mit ungültigen Pointertypen.

Ansonsten sind die verlangten Klassen gemäß der Spezifikation zu implementieren.

Zusätzlich soll ein ErrorHandler implementiert werden, der das Handling der Error Messages übernimmt. Um den Speicherbedarf gering zu halten soll dieser als Singleton implementiert werden, da hierbei immer nur eine Instanz verwendet werden kann. So erreicht man das die Messages nicht unnötig kopiert und mit mehreren Instanzen hantiert werden muss. Die Singleton Instanz wird zwar bei Programmende zerstört, dennoch soll eine destroy Methode implementiert werden, die die Singleton Instanz sauber löscht.

## Implementierung

Folgend sind die Implementierungen der zweiten Aufgabenstellung angeführt.

### FsNode.h

Folgend ist die Spezifikation von FsNode angeführt.

/\*

\* FSNode.h

\* The FsNode specification which is the root class for the file system nodes.

\*

\* Created on: Dec 27, 2014

\* Author: Thomas Herzog

\*/

#ifndef FSNODE\_H\_

#define FSNODE\_H\_

#include "Node.h"

#define FS\_NODE\_CLASS "FsNode"

/\*\*

\* This class is the root class for the file system nodes.

\*/

class FsNode**:** public Node **{**

protected**:**

const std**::**string name**;**

public**:**

////////////////////////////////////////////////////////////

// Constructor and Destructor //

////////////////////////////////////////////////////////////

/\*\*

\* This class is not allowed to be instantiated via an empty constructor

\*/

FsNode**()** **=** **delete;**

/\*\*

\* Constructs an instance with a name

\*/

FsNode**(**const std**::**string name**);**

/\*\*

\* Copy constructor which performs a deep copy

\*/

FsNode**(**const FsNode **&** other**);**

/\*\*

\* Destructor which does nothing.

\* Deletion is performed via Node constructor

\*/

virtual **~**FsNode**();**

////////////////////////////////////////////////////////////

// Getter and Setter //

////////////////////////////////////////////////////////////

/\*\*

\* Gets teh first child reference

\*

\* **@return**

\* the hold first child reference

\*/

virtual FsNode**\*** getFirstChild**()** const**;**

/\*\*

\* Gets the next sibling reference

\*

\* **@return**

\* the held next sibling reference

\*/

virtual FsNode**\*** getNextSibling**()** const**;**

////////////////////////////////////////////////////////////

// Utils //

////////////////////////////////////////////////////////////

/\*\*

\* Returns the string representation of this class

\*

\* **@return**

\* the string representation of this class

\*/

virtual std**::**string AsString**();**

/\*\*

\* Performs a deep copy of this class

\*

\* **@return**

\* the copied instance

\*/

virtual FsNode**\*** clone**()** const**;**

virtual std**::**string getName**()** const**;**

**};**

#endif /\* FSNODE\_H\_ \*/

### FsNode.cpp

Folgend ist die Implementierung der Spezifikation FsNode.h angeführt.

/\*

\* FsNode.cpp

\* This is the implementaion of the Fsnode specification.

\*

\* Created on: Jan 11, 2015

\* Author: Thomas Herzog

\*/

#include <sstream>

#include "FSNode.h"

**using** **namespace** std**;**

**using** **namespace** ML**;**

////////////////////////////////////////////////////////////

// Constructor and Destructor //

////////////////////////////////////////////////////////////

FsNode**::**FsNode**(**const string name**)** **:**

Node**(),** name**(**name**)** **{**

Register**(**FS\_NODE\_CLASS**,** NODE\_CLASS**);**

**}** /\* FsNode::FsNode \*/

FsNode**::**FsNode**(**const FsNode **&** other**)** **:**

Node**::**Node**(**other**),** name**(**"copy\_of\_" **+** name**)** **{**

Register**(**FS\_NODE\_CLASS**,** NODE\_CLASS**);**

**}** /\* FsNode::FsNode \*/

FsNode**::~**FsNode**()** **{**

**}** /\* FsNode::~FsNode \*/

////////////////////////////////////////////////////////////

// Getter and Setter //

////////////////////////////////////////////////////////////

string FsNode**::**getName**()** const **{**

**return** name**;**

**}** /\* FsNode::getName \*/

FsNode**\*** FsNode**::**getFirstChild**()** const **{**

**return** **(**FsNode**\*)** Node**::**getFirstChild**();**

**}** /\* FsNode::getFirstChild \*/

FsNode**\*** FsNode**::**getNextSibling**()** const **{**

**return** **(**FsNode**\*)** Node**::**getNextSibling**();**

**}** /\* FsNode::getNextSibling \*/

////////////////////////////////////////////////////////////

// Utils //

////////////////////////////////////////////////////////////

string FsNode**::**AsString**()** **{**

stringstream ss**;**

ss **<<** "FsNode('" **<<** name **<<** "')"**;**

**return** ss**.**str**();**

**}** /\* FsNode::AsString \*/

FsNode**\*** FsNode**::**clone**()** const **{**

**return** **new** FsNode**(\*this);**

**}** /\* FsNode::clone \*/

### Directory.h

Folgend ist die Spezifikation von Directory angeführt.

/\*

\* Directory.h

\* This is the specification of the Directory

\*

\* Created on: Dec 27, 2014

\* Author: Thomas Herzog

\*/

#ifndef DIRECTORY\_H\_

#define DIRECTORY\_H\_

#include "FSNode.h"

#define DIRECTORY\_CLASS "Directory"

/\*\*

\* This class represents a directory in the file system.

\*/

class Directory**:** public FsNode **{**

public**:**

////////////////////////////////////////////////////////////

// Constructor and Destructor //

////////////////////////////////////////////////////////////

/\*\*

\* This class is not allowed to be instantiated with an empty contructor

\*/

Directory**()** **=** **delete;**

/\*\*

\* Constructs a instance with the given name

\*/

Directory**(**const std**::**string **&** name**);**

/\*\*

\* Copies the given instance via an deep copy

\*/

Directory**(**const Directory **&** other**);**

/\*\*

\* This destructor does nothing because deletion is performed in Node class

\*/

virtual **~**Directory**();**

////////////////////////////////////////////////////////////

// Utils //

////////////////////////////////////////////////////////////

/\*\*

\* Performs a deep copy of this instance

\*

\* **@return**

\* the copied instance

\*/

virtual Directory**\*** clone**()** const**;**

/\*\*

\* Returns a string representation of this class

\*

\* **@return**

\* the string representation of this class

\*/

virtual std**::**string AsString**()** const**;**

/\*\*

\* Prints the directory and all of its referenced children.

\* Means the subtree held by this node will be printed.

\*

\* **@param**

\* ostram: the ostram to put printed text on

\*/

virtual void print**(**std**::**ostream **&** os**)** const**;**

**};**

#endif /\* DIRECTORY\_H\_ \*/

### Directory.cpp

Folgend ist die Implementierung der Spezifikation Directory.h angeführt.

/\*

\* Directory.cpp

\* This is the implementation of the Directory specification

\*

\* Created on: Jan 11, 2015

\* Author: Thomas Herzog

\*/

#include <sstream>

#include "Directory.h"

**using** **namespace** std**;**

**using** **namespace** ML**;**

////////////////////////////////////////////////////////////

// Constructor and Destructor //

////////////////////////////////////////////////////////////

Directory**::**Directory**(**const string **&** name**)** **:**

FsNode**(**name**)** **{**

Register**(**DIRECTORY\_CLASS**,** FS\_NODE\_CLASS**);**

**}** /\* Directory::Directory \*/

Directory**::**Directory**(**const Directory **&** other**)** **:**

FsNode**::**FsNode**(**other**)** **{**

/\* avoid multiple registration \*/

Register**(**DIRECTORY\_CLASS**,** FS\_NODE\_CLASS**);**

**}** /\* Directory::Directory \*/

Directory**::~**Directory**()** **{**

**}** /\* Directory::~Directory \*/

////////////////////////////////////////////////////////////

// Utils //

////////////////////////////////////////////////////////////

Directory**\*** Directory**::**clone**()** const **{**

**return** **new** Directory**(\*this);**

**}** /\* Directory::clone \*/

string Directory**::**AsString**()** const **{**

stringstream ss**;**

ss **<<** "Directory('" **<<** name **<<** "')"**;**

**return** ss**.**str**();**

**}** /\* Directory::AsString \*/

void Directory**::**print**(**ostream **&** os**)** const **{**

os **<<** AsString**();**

**}** /\* Directory::print \*/

### File.h

Folgend ist die Spezifikation von File angeführt.

/\*

\* File.h

\* This is the specification of the File

\*

\* Created on: Dec 27, 2014

\* Author: Thomas Herzog

\*/

#ifndef FILE\_H\_

#define FILE\_H\_

#include "FSNode.h"

#define FILE\_CLASS "File"

/\*\*

\* This class represents a file in the file system.

\*/

class File**:** public FsNode **{**

public**:**

////////////////////////////////////////////////////////////

// Constructor and Destructor //

////////////////////////////////////////////////////////////

/\*\*

\* This class is not allowed to be instantiated via an empty constructor

\*/

File**()** **=** **delete;**

/\*\*

\* Constructs an instance with a name

\*/

File**(**const std**::**string filename**);**

/\*\*

\* Copy constructor which performs a deep copy

\*/

File**(**const File **&** other**);**

/\*\*

\* Destructor which does nothing.

\* Deletion is performed via Node constructor

\*/

virtual **~**File**();**

////////////////////////////////////////////////////////////

// Getter and Setter //

////////////////////////////////////////////////////////////

/\*\*

\* Getter method for the first child references which will

\* always return null, because File cannot define a first child.

\*

\* **@return**

\* the first child reference

\*/

virtual FsNode**\*** getFirstChild**()** const**;**

/\*\*

\* Does nothin gbecuase file is not allowed to define a first child.

\*

\* **@param**

\* firstChild: the new first child node reference

\*/

virtual void setFirstChild**(**FsNode**\*** firstChild**);**

////////////////////////////////////////////////////////////

// Utils //

////////////////////////////////////////////////////////////

/\*\*

\* Performs a deep copy of this instance.

\*

\* **@return**

\* the copied instance

\*/

virtual File**\*** clone**()** const**;**

/\*\*

\* Prints the directory and all of its referenced nodes.

\* Means the subtree held by this node will be printed.

\*

\* **@param**

\* ostram: the ostram to put printed text on

\*/

virtual void print**(**std**::**ostream **&** os**)** const**;**

/\*\*

\* Returns the string representation of this class

\*

\* **@return**

\* the string representation of this class

\*/

virtual std**::**string AsString**()** const**;**

**};**

#endif /\* FILE\_H\_ \*/

### File.cpp

Folgend ist die Implementierung von File.h angeführt.

/\*

\* File.cpp

\* This is the implementation of the File specification

\*

\* Created on: Jan 11, 2015

\* Author: Thomas Herzog

\*/

#include <string>

#include <sstream>

#include "File.h"

**using** **namespace** std**;**

**using** **namespace** ML**;**

////////////////////////////////////////////////////////////

// Constructor and Destructor //

////////////////////////////////////////////////////////////

File**::**File**(**const string filename**)** **:**

FsNode**::**FsNode**(**filename**)** **{**

Register**(**FILE\_CLASS**,** FS\_NODE\_CLASS**);**

**}** /\* File::File \*/

File**::**File**(**const File **&** other**)** **:**

FsNode**::**FsNode**(**other**)** **{**

Register**(**FILE\_CLASS**,** FS\_NODE\_CLASS**);**

**}** /\* File::File \*/

File**::~**File**()** **{**

**}** /\* File::~File \*/

////////////////////////////////////////////////////////////

// Getter and Setter //

////////////////////////////////////////////////////////////

FsNode**\*** File**::**getFirstChild**()** const **{**

**return** nullptr**;**

**}** /\* File::getFirstChild \*/

void File**::**setFirstChild**(**FsNode**\*** node**)** **{**

// Do nothing here

cout **<<** "File does not allow setting of a first child" **<<** endl**;**

**}** /\* File::setFirstChild \*/

////////////////////////////////////////////////////////////

// Utils //

////////////////////////////////////////////////////////////

string File**::**AsString**()** const **{**

stringstream ss**;**

ss **<<** "File('" **<<** name **<<** "')"**;**

**return** ss**.**str**();**

**}** /\* File::AsString \*/

File**\*** File**::**clone**()** const **{**

**return** **new** File**(\*this);**

**}** /\* File::clone \*/

void File**::**print**(**ostream **&** os**)** const **{**

os **<<** AsString**();**

**}** /\* File::print \*/

### FileSystem.h

Folgen ist die Spezifikation von FileSystem angeführt.

/\*

\* FileSystem.h

\* This is the FileSystem specification

\*

\* Created on: Dec 27, 2014

\* Author: Thomas Herzog

\*/

#ifndef FILESYSTEM\_H\_

#define FILESYSTEM\_H\_

#include <vector>

#include <string>

#include "Tree.h"

#include "FSNode.h"

#include "File.h"

#include "Directory.h"

#include "FsErrorHandler.h"

/\* define file separator for unix and win systems \*/

#ifdef \_WIN32

#define FILE\_SEPARATOR "\\"

#else

#define FILE\_SEPARATOR "/"

#endif

#define FILESYSTEM\_CLASS "FileSystem"

/\*\*

\* This struct is used to hold information about the walked path.

\*/

**typedef** struct WalkResult **{**

/\*\*

\* The error type. default NONE.

\* Indicates an error/successful path walk

\*/

FsErrorType error **=** FsErrorType**::**NONE**;**

/\*\*

\* The last visited node during walking path.

\* Do not delete this reference is managed by the FileSystem

\*/

FsNode**\*** lastVisitedNode**;**

/\*\*

\* The actual filename.

\* Filename is allowed to contain path information and

\* therefore needs to be extracted and set on the result.

\* Needs to be deleted by the caller.

\*/

std**::**string childNodeName**;**

**}** WalkResult**;**

/\*\*

\* This class represents a Tree implementation which realizes a file system.

\* Therefore that some formerly public visible methods are not allowed to be called

\* by an outside caller, they have been moved to the private section to avoid file system manipulation

\* by the caller outside the newly introduced methods such as mkdir and so on.

\*

\* Therefore that Tree implementation only depends on Node types but this implementations differs between

\* Directory (FsNode) and File(FsNode) where FsNode inherits from Node some methods where overwritten

\* to restrict the the types which must be given by the caller.

\* For instance it is not possible to add a child to a File node, which leads to the fact

\* that only directories are valid parents for inserted Nodes.

\*/

class FileSystem**:** public Tree **{**

private**:**

////////////////////////////////////////////////////////////

// private utils //

////////////////////////////////////////////////////////////

/\*\*

\* This method validates if the given path is a valid one and only contains directories.

\* If the childName contains path information the filename will be extracted and set ont he result

\* and the path information is added to the given path.

\* If the path is invalid an error code will be set.

\*

\* **@param**

\* path: the path to be validated

\* **@param**

\* childName: the name of the child (is allowed to contain path information)

\* **@return**

\* the result struct which contains all necessary information about the path and intended child

\*/

WalkResult validatePath**(**const std**::**string **&** path**,**

const std**::**string childName**)** const**;**

/\*\*

\* This method recursively walks the given full path defined by the vector.

\* If the path is invalid a error code is set for later evaluation.

\* The last visited node is always set. In the case of an error the lastVisited node

\* will be the node visited before the error occurred, otherwise the last visited node will

\* point to the last node defined by the full path

\*

\* **@param**

\* currentNode: the node to search for the direct child with the path part set as name

\* **@param**

\* fullPath: a vector containing the full path split by the file separator

\* **@param**

\* result: the result where the error code and last node are set.

\*/

void walkPath**(**const FsNode**\*** currentNode**,**

std**::**vector**<**std**::**string**>** **&** fullPath**,** WalkResult **&** result**)** const**;**

/\*\*

\* Gets the direct child of the given parent with the given name set.

\*

\* **@param**

\* node: the parent node to search on

\* **@param**

\* name: the name of the searched child

\* **@return**

\* the found node, null otherwise

\*/

FsNode**\*** getChildByName**(**const FsNode**\*** node**,** const std**::**string **&** name**)** const**;**

/\*\*

\* Deletes the given child from the parent safely.

\* Means the child is removed from the backed tree nicely.

\*

\* **@param**

\* parent: the parent node to delete child from

\* **@param**

\* child: the child to be deleted from the parent safely

\*

\*/

void deleteFsNode**(**FsNode**\*** parent**,** FsNode**\*** child**);**

////////////////////////////////////////////////////////////

// hidden former public methods //

// for type safety (Node\* -> Directory\*) //

// not allowed to be called by outside caller //

// //

// see parent class for method doc //

////////////////////////////////////////////////////////////

void deleteSubTree**(**FsNode**\*** node**);**

void setRoot**(**Directory**\*** node**);**

void insertChild**(**Directory**\*** parent**,** FsNode**\*** child**);**

Directory**\*** getRoot**()** const**;**

public**:**

////////////////////////////////////////////////////////////

// Constructor and Destructor //

////////////////////////////////////////////////////////////

/\*\*

\* Empty constructor which inits the file system with a root directory ""

\*/

FileSystem**();**

/\*\*

\* Copy constructor which delegates to the parent copy constructor because

\* no need to perform custom copy

\*/

FileSystem**(**const FileSystem **&** other**);**

/\*\*

\* Destructor which does nothing because cleanup is performed by the base class

\*/

virtual **~**FileSystem**();**

////////////////////////////////////////////////////////////

// public visible methods //

////////////////////////////////////////////////////////////

/\*\*

\* Touches a file and creates one if the file does not exist yet.

\*

\* **@param**

\* path: the path to the file

\* param

\* filename: the name of the file to be touched (allowed to contain path information)

\*/

virtual void touch**(**const std**::**string **&** path**,** const std**::**string **&** filename**);**

/\*\*

\* Creates a directory if it does not exist yet.

\*

\* **@param**

\* path: the path under which the directory shall be created

\* **@param**

\* dirname: the name of the directory to be created (allowed to contain path information)

\*/

virtual void mkdir**(**const std**::**string **&** path**,** const std**::**string **&** dirname**);**

/\*\*

\* Removes a file from the given path if the file exits.

\*

\* **@param**

\* path: the path where the file resides

\* **@param**

\* filename: the name of the file to be deleted

\*/

virtual void rm**(**const std**::**string **&** path**,** const std**::**string **&** filename**);**

/\*\*

\* Removes a directory from the given path if the directory exists.

\*

\* **@param**

\* path: the path where the directory resides

\* **@param**

\* filename: the name of the file to be deleted

\*/

virtual void rmdir**(**const std**::**string **&** path**,** const std**::**string **&** dirname**);**

/\*\*

\* List all of the content of the file system.

\* Uses Tree print method of Tree class.

\*/

virtual void ls**()** const**;**

**};**

#endif /\* FILESYSTEM\_H\_ \*/

### FileSystem.cpp

Folgend ist die Implementierung von FileSystem angeführt.

/\*

\* FileSystem.cpp

\* The implementation of the FileSystem specification.

\*

\* Created on: Dec 27, 2014

\* Author: Thomas Herzog

\*/

#include "FileSystem.h"

#include <cstring>

**using** **namespace** std**;**

**using** **namespace** ML**;**

/\*\*

\* No singleton present at startup.

\* Only one single instance needed for the

\*/

bool FsErrorHandler**::**createdFlag **=** **false;**

FsErrorHandler**\*** FsErrorHandler**::**instance **=** nullptr**;**

WalkResult FileSystem**::**validatePath**(**const std**::**string **&** path**,**

const std**::**string childName**)** const **{**

WalkResult result**;**

FsNode**\*** node **=** getRoot**();**

vector**<**string**>** fullPath**;**

result**.**lastVisitedNode **=** nullptr**;**

/\* split path and put into vector for later path walking \*/

char**\*** buff **=** strdup**(**path**.**c\_str**());**

char**\*** token **=** strtok**(**buff**,** FILE\_SEPARATOR**);**

**while** **(**token **!=** nullptr**)** **{**

fullPath**.**insert**(**fullPath**.**begin**(),** token**);**

token **=** strtok**(**nullptr**,** FILE\_SEPARATOR**);**

**}**

/\* add childname path information if present \*/

**delete[]** buff**;**

buff **=** nullptr**;**

buff **=** strdup**(**childName**.**c\_str**());**

token **=** strtok**(**buff**,** FILE\_SEPARATOR**);**

char**\*** prevToken **=** nullptr**;**

**while** **(**token **!=** nullptr**)** **{**

**if** **(**prevToken **!=** nullptr**)** **{**

fullPath**.**insert**(**fullPath**.**begin**(),** prevToken**);**

**}**

prevToken **=** token**;**

token **=** strtok**(**nullptr**,** FILE\_SEPARATOR**);**

**}**

/\* set actual child name which is always the previous token from the last token \*/

result**.**childNodeName **=** string**(**prevToken**);**

/\* root path always valid because points to root node \*/

**if** **((**path**.**compare**(**""**)** **==** 0**)** **&&** **(**fullPath**.**empty**()))** **{**

result**.**lastVisitedNode **=** node**;**

**return** result**;**

**}**

/\* walk the path and validate if all of the nodes of the calculated path exist \*/

walkPath**(**node**,** fullPath**,** result**);**

/\* cleanup buffer \*/

**delete[]** buff**;**

**return** result**;**

**}** /\* FileSystem::validatePath \*/

void FileSystem**::**walkPath**(**const FsNode**\*** currentNode**,** vector**<**string**>** **&** fullPath**,**

WalkResult **&** result**)** const **{**

/\* recursive anchor. only proceed if no error and path information left in vector \*/

**if** **((**result**.**error **==** FsErrorType**::**NONE**)** **&&** **(!**fullPath**.**empty**()))** **{**

string part **=** fullPath**.**back**();**

fullPath**.**pop\_back**();**

/\* try to get child from current node \*/

FsNode**\*** node **=** getChildByName**(**currentNode**,** part**);**

/\* child node found \*/

**if** **(**node **!=** nullptr**)** **{**

result**.**lastVisitedNode **=** node**;**

/\* Found but no directory \*/

**if** **(!**node**->**Class**().**compare**(**DIRECTORY\_CLASS**)** **==** 0**)** **{**

result**.**error **=** FsErrorType**::**PATH\_CONTAIN\_FILES**;**

**}**

/\* Keep walking \*/

**else** **{**

walkPath**(**node**,** fullPath**,** result**);**

**}**

**}**

/\* child node not found means path part invalid \*/

**else** **{**

result**.**error **=** FsErrorType**::**INVALID\_PATH**;**

**}**

**}**

**}** /\* FileSystem::walkPath \*/

FsNode**\*** FileSystem**::**getChildByName**(**const FsNode**\*** node**,**

const std**::**string **&** name**)** const **{**

FsNode**\*** next **=** nullptr**;**

**if** **(**node **!=** nullptr**)** **{**

next **=** node**->**getFirstChild**();**

**while** **((**next **!=** nullptr**)** **&&** **(!**next**->**getName**().**compare**(**name**)** **==** 0**))** **{**

next **=** next**->**getNextSibling**();**

**}**

next **=** **((**next **!=** nullptr**)** **&&** **(**next**->**getName**().**compare**(**name**)** **==** 0**))** **?**

next **:** nullptr**;**

**}**

**return** next**;**

**}** /\* FileSystem::getChildByName \*/

void FileSystem**::**deleteFsNode**(**FsNode**\*** parent**,** FsNode**\*** child**)** **{**

FsNode**\*** prev **=** nullptr**;**

FsNode**\*** node **=** parent**->**getFirstChild**();**

/\* find previous from to delete node \*/

**while** **((**node **!=** nullptr**)** **&&** **(**node **!=** child**))** **{**

prev **=** node**;**

node **=** node**->**getNextSibling**();**

**}**

/\* first child is supposed to be deleted \*/

**if** **(**prev **==** nullptr**)** **{**

parent**->**setFirstChild**(**node**->**getNextSibling**());**

**}**

/\* remove child nicely \*/

**else** **{**

prev**->**setNextSibling**(**node**->**getNextSibling**());**

**}**

/\* remove reference otherwise would be deleted via cascade \*/

node**->**setNextSibling**(**nullptr**);**

/\* delete child \*/

**delete** node**;**

**}** /\* FileSystem::deleteFsNode \*/

void FileSystem**::**deleteSubTree**(**FsNode**\*** node**)** **{**

/\* delegate to parent class \*/

Tree**::**deleteSubTree**(**node**);**

**}** /\* FileSystem::deleteSubTree \*/

void FileSystem**::**setRoot**(**Directory**\*** node**)** **{**

// FileSystem root cannot be defined from caller is always root path ""

std**::**cout **<<** "Root is not allowed to be set on a FileSystem instance !!! "

**<<** std**::**endl **<<** std**::**flush**;**

**}** /\* FileSystem::setRoot \*/

void FileSystem**::**insertChild**(**Directory**\*** parent**,** FsNode**\*** child**)** **{**

/\* delegate to parent class \*/

Tree**::**insertChild**(**parent**,** child**);**

**}** /\* FileSystem::insertChild \*/

Directory**\*** FileSystem**::**getRoot**()** const **{**

/\* it safe to cast brutally because can only be instanceof Directory \*/

**return** **(**Directory**\*)** Tree**::**getRoot**();**

**}** /\* FileSystem::getRoot \*/

////////////////////////////////////////////////////////////

// Constructor and Destructor //

////////////////////////////////////////////////////////////

FileSystem**::**FileSystem**()** **:**

Tree**::**Tree**(new** Directory**(**""**))** **{**

Register**(**FILESYSTEM\_CLASS**,** TREE\_CLASS**);**

**}** /\* FileSystem::FileSystem \*/

FileSystem**::**FileSystem**(**const FileSystem **&** other**)** **:**

Tree**::**Tree**(**other**)** **{**

Register**(**FILESYSTEM\_CLASS**,** TREE\_CLASS**);**

**}** /\* FileSystem::FileSystem \*/

FileSystem**::~**FileSystem**()** **{**

**}** /\* FileSystem::~FileSystem \*/

void FileSystem**::**touch**(**const string **&** path**,** const string **&** filename**)** **{**

WalkResult result **=** validatePath**(**path**,** filename**);**

/\* Error occurred \*/

**if** **(**result**.**error **!=** FsErrorType**::**NONE**)** **{**

string lastNodeName **=**

**(**result**.**lastVisitedNode **!=** nullptr**)** **?**

result**.**lastVisitedNode**->**getName**()** **:** "not available"**;**

cout **<<** FsErrorHandler**::**getInstance**()->**getMessage**(**result**.**error**)**

**<<** " (last visited node: " **<<** lastNodeName **<<** ")" **<<** endl

**<<** flush**;**

**}**

/\* file exists \*/

**else** **if** **(**getChildByName**(**result**.**lastVisitedNode**,** result**.**childNodeName**)**

**!=** nullptr**)** **{**

cout **<<** "Touch existing file '" **<<** result**.**childNodeName **<<** "'" **<<** endl

**<<** flush**;**

**}**

/\* create new file \*/

**else** **{**

insertChild**(dynamic\_cast<**Directory**\*>(**result**.**lastVisitedNode**),**

**new** File**(**result**.**childNodeName**));**

cout **<<** "touched newly created file '" **<<** result**.**childNodeName **<<** "'"

**<<** endl **<<** flush**;**

**}**

**}** /\* FileSystem::touch \*/

void FileSystem**::**mkdir**(**const string **&** path**,** const string **&** dirname**)** **{**

WalkResult result **=** validatePath**(**path**,** dirname**);**

/\* Error occurred \*/

**if** **(**result**.**error **!=** FsErrorType**::**NONE**)** **{**

string lastNodeName **=**

**(**result**.**lastVisitedNode **!=** nullptr**)** **?**

result**.**lastVisitedNode**->**getName**()** **:** "not available"**;**

cout **<<** FsErrorHandler**::**getInstance**()->**getMessage**(**result**.**error**)**

**<<** " (last visited node: " **<<** lastNodeName **<<** ")" **<<** endl

**<<** flush**;**

**}**

/\* create new directory \*/

**else** **{**

insertChild**(dynamic\_cast<**Directory**\*>(**result**.**lastVisitedNode**),**

**new** Directory**(**result**.**childNodeName**));**

cout **<<** "directory created '" **<<** result**.**childNodeName **<<** "'" **<<** endl

**<<** flush**;**

**}**

**}** /\* FileSystem::mkdir \*/

void FileSystem**::**rm**(**const string **&** path**,** const string **&** filename**)** **{**

WalkResult result **=** validatePath**(**path**,** filename**);**

FsNode**\*** node**;**

FsNode**\*** parent**;**

/\* Error occurred \*/

**if** **(**result**.**error **!=** FsErrorType**::**NONE**)** **{**

cout **<<** FsErrorHandler**::**getInstance**()->**getMessage**(**result**.**error**)** **<<** endl

**<<** flush**;**

**}**

/\* try to remove child node \*/

**else** **{**

parent **=** result**.**lastVisitedNode**;** //result.walkedNodes.back();

node **=** getChildByName**(**parent**,** result**.**childNodeName**);**

/\* node not found \*/

**if** **(**node **==** nullptr**)** **{**

cout **<<** "file not found '" **<<** result**.**childNodeName **<<** "'" **<<** endl

**<<** flush**;**

**}**

/\* node not a file \*/

**else** **if** **(**node**->**Class**().**compare**(**FILE\_CLASS**)** **!=** 0**)** **{**

cout **<<** "is not a file '" **<<** result**.**childNodeName **<<** "'" **<<** endl

**<<** flush**;**

**}**

/\* delete file \*/

**else** **{**

deleteFsNode**(**parent**,** node**);**

cout **<<** "file deleted '" **<<** filename **<<** "'" **<<** endl **<<** flush**;**

**}**

**}**

**}** /\* FileSystem::rm \*/

void FileSystem**::**rmdir**(**const string **&** path**,** const string **&** dirname**)** **{**

WalkResult result **=** validatePath**(**path**,** dirname**);**

FsNode**\*** node**;**

FsNode**\*** parent**;**

/\* Error occurred \*/

**if** **(**result**.**error **!=** FsErrorType**::**NONE**)** **{**

cout **<<** FsErrorHandler**::**getInstance**()->**getMessage**(**result**.**error**)** **<<** endl

**<<** flush**;**

**}**

/\* try to remove directory \*/

**else** **{**

parent **=** result**.**lastVisitedNode**;** //result.walkedNodes.back();

node **=** getChildByName**(**parent**,** result**.**childNodeName**);**

/\* node not found \*/

**if** **(**node **==** nullptr**)** **{**

cout **<<** "directory not found '" **<<** result**.**childNodeName **<<** "'"

**<<** endl **<<** flush**;**

**}**

/\* node not a directory \*/

**else** **if** **(!**node**->**Class**().**compare**(**DIRECTORY\_CLASS**)** **==** 0**)** **{**

cout **<<** "is not a directory '" **<<** result**.**childNodeName **<<** "'"

**<<** endl **<<** flush**;**

**}**

/\* directory is not empty \*/

**else** **if** **(**node**->**getFirstChild**()** **!=** nullptr**)** **{**

cout **<<** "directory is not empty" **<<** endl **<<** flush**;**

**}**

/\* delete directory \*/

**else** **{**

deleteFsNode**(**parent**,** node**);**

cout **<<** "directory deleted '" **<<** result**.**childNodeName **<<** "'" **<<** endl

**<<** flush**;**

**}**

**}**

**}** /\* FileSystem::rmdir \*/

void FileSystem**::**ls**()** const **{**

print**(**cout**);**

**}** /\* FileSystem::ls \*/

### FsErrorHandler.h

Folgend ist die Spezifikation und Implementierung von FsErroHandler angeführt, der mittels Singleton Pattern implementiert wurde.

/\*

\* FsMessageHandler.h

\* This is the specification and implementation of the FsErrorHandler.

\* This class is implemented with the Singleton Pattern

\*

\* Created on: Dec 28, 2014

\* Author: Thomas Herzog

\*/

#ifndef FSMESSAGEHANDLER\_H\_

#define FSMESSAGEHANDLER\_H\_

#include <map>

#include <string>

#include <sstream>

/\*\*

\* This enumeration class holds all possible errors which can occur on the file system

\*/

enum class FsErrorType **{**

/\* No error occurred \*/

NONE**,**

/\* Given path contains files which cannot be walked as directories \*/

PATH\_CONTAIN\_FILES **=** 1**,**

/\* Path contains directories which do not exist \*/

INVALID\_PATH **=** 2**,**

/\* File does not exist under intended location \*/

FILE\_NOT\_FOUND **=** 3**,**

/\* directory does not exist under intended location \*/

DIR\_NOT\_FOUND **=** 4**,**

/\* directory is not empty and therefore cannot be deleted \*/

DIR\_NOT\_EMPTY **=** 5**,**

/\* not found directory/file \*/

NODE\_NOT\_FOUND **=** 6

**};**

/\*\*

\* Singleton error message handler which provides the error messages for the specified errors.

\* It gets constructed during first call of getInstance and destroyed when program shuts down

\*/

class FsErrorHandler **{**

private**:**

static bool createdFlag**;**

static FsErrorHandler**\*** instance**;**

std**::**map**<**FsErrorType**,** std**::**string**>** errorMessages**;**

////////////////////////////////////////////////////////////

// Constructor and Destructor //

////////////////////////////////////////////////////////////

/\*\*

\* Private constructor because not allowed to be constructed via new operator.

\* Init the message during construction

\*/

inline FsErrorHandler**()** **{**

initMessages**();**

**}**

/\*\*

\* Prevent destruction of FsErrorHandler singleton pointer.

\*/

inline **~**FsErrorHandler**()** **{**

**}**

/\*\*

\* Initializes the message map by putting all messages into a map to their corresponding enumeration.

\*/

inline void initMessages**()** **{**

errorMessages**.**insert**(**

std**::**pair**<**FsErrorType**,** std**::**string**>(**FsErrorType**::**DIR\_NOT\_EMPTY**,**

"Cannot delete not empty directory"**));**

errorMessages**.**insert**(**

std**::**pair**<**FsErrorType**,** std**::**string**>(**FsErrorType**::**DIR\_NOT\_FOUND**,**

"Directory not found"**));**

errorMessages**.**insert**(**

std**::**pair**<**FsErrorType**,** std**::**string**>(**FsErrorType**::**FILE\_NOT\_FOUND**,**

"File not found"**));**

errorMessages**.**insert**(**

std**::**pair**<**FsErrorType**,** std**::**string**>(**FsErrorType**::**INVALID\_PATH**,**

"Given path is invalid !!!"**));**

errorMessages**.**insert**(**

std**::**pair**<**FsErrorType**,** std**::**string**>(**

FsErrorType**::**PATH\_CONTAIN\_FILES**,**

"Cannot walk path which contains files"**));**

**}**

public**:**

/\*\*

\* Returns the existing instance or creates a new instance

\* if the instance is not present yet.

\*

\* **@return**

\* the created singleton instance

\*/

static inline FsErrorHandler**\*** getInstance**()** **{**

**if** **(!**createdFlag**)** **{**

std**::**cout **<<** "singleton instance created" **<<** std**::**endl

**<<** std**::**flush**;**

instance **=** **new** FsErrorHandler**();**

createdFlag **=** **true;**

**return** instance**;**

**}** **else** **{**

std**::**cout **<<** "returning existing singleton instance" **<<** std**::**endl

**<<** std**::**flush**;**

**return** instance**;**

**}**

**}**

/\*\*

\* Allows to nicely destroy the singleton instance.

\*/

static inline void destroyInstance**()** **{**

**if** **(**createdFlag**)** **{**

**delete** instance**;**

createdFlag **=** **false;**

std**::**cout **<<** "singleton instance destroyed" **<<** std**::**endl

**<<** std**::**flush**;**

**}** **else** **{**

std**::**cout **<<** "no singleton instance present for destroy "

**<<** std**::**endl **<<** std**::**flush**;**

**}**

**}**

/\*\*

\* Returns the message mapped to the given enum.

\*

\* @param:

\* type: the error type mapped to a error message

\* **@return**

\* the mapped message or a message which tells the caller that this key is not mapped

\* to any message

\*/

inline std**::**string getMessage**(**const FsErrorType **&** type**)** **{**

**try** **{**

**return** errorMessages**.**at**(**type**);**

**}** **catch** **(**std**::**out\_of\_range **&** e**)** **{**

std**::**stringstream ss**;**

ss **<<** "Key not managed by this handler !!!" **<<** std**::**endl **<<** e**.**what**()**

**<<** std**::**endl **<<** std**::**flush**;**

**return** ss**.**str**();**

**}**

**}**

**};**

#endif /\* FSMESSAGEHANDLER\_H\_ \*/