

## 1 Allgemein

Folgend ist die allgemeine Dokumentation angeführt, die für beide Aufgabenstellungen anzuwenden ist.

Es soll die MiniLib-Library verwendet werden und es sollen alle Klassen von der zur Verfügung gestellten Klasse Object abgeleitet werden um deren Methoden AsString() usw. wiederverwenden bzw. überschreiben zu können. Des Weiteren soll während der Entwicklung stark mit WriteMetaInfo() der MiniLib-Library gearbeitet werden, da diese es erlaubt zu prüfen ob auch alle Instanzen vollständig gelöscht wurden. Jedoch soll bei der finalen Version darauf verzichtet werden um die Ausgabe der Tests gering zu halten. Bei Bedarf kann die Ausgabe der Metainformationen hinzugefügt werden.

Bezüglich der Tests soll diesmal versucht werden diese mit einem Unit Test Framework (CUTE, BOOST) zu implementieren, damit diese auch mit vordefinierten Assert Funktionen versehen werden können sofern diese anzuwenden sind (Bsp.: print usw. erfordern keine Assert Funktion).

Dieses Test Framework soll mit in das Projekt aufgenommen werden, damit diese auch portiert werden können ohne das man dieses Testframework selbstständig einbinden muss und ohne dass man das Projekt selbst konfigurieren muss. Dies sollte kein Problem darstellen, da hierbei lediglich der Source beim Kompilieren eingebunden werden soll.

*Während der Entwicklung wurde WriteMetaInfo von ML::Object verwendet und es zeigte sich das etwas mit Register(CLAZZ, BASE\_CLAZZ) nicht funktionierte. Das Erstellen der Instanz wurde nicht korrekt erkannt das Löschen dieser jedoch schon. Daher gab es Error in der Ausgabe da zwar 0 Instanzen erstellt aber n Instanzen gelöscht wurden, was sich natürlicherweise in -n Instanzen in der Ausgabe niederschlug. Ich kann leider nicht sagen was das Problem ist. Vielleicht liegt es an der Testsuite über CUTE was sich hier negativ auswirkt. Es scheint so als würde beim Erstellen der Instanz nicht erkannt werden ob diese dynamisch ist, aber hierzu reichen meine Kenntnisse und auch die zur Verfügung stehende Zeit nicht aus. Des Weiteren ist die Abhängigkeit des Testframeworks nicht mehr zu entfernen. Ich nehme aber an das dies für einen geübten C++ Entwickler kein Aufwand wäre diesen Bug zu lösen, leider gehöre ich hier nicht dazu.*

## 1.1 Copy Constructor

Standardmäßig Wird vom Compiler ein Copy Constructor zur Verfügung gestellt, welcher lediglich ein shadow copy durchführt, also z.B.: Pointer-Referenzen kopiert (Adresse) aber kein deep Copy durchführt. Bei einem deep copy würden die Objekte kopiert und nicht nur die Pointer-Referenzen kopiert werden. Diese wird aber nicht standardmäßig durchgeführt und muss selbstständig implementiert werden. Dies ist aber lediglich erforderlich bei Klassen die Pointer Objekte verwenden. Der Kopierkonstruktor erstellt eine Kopie der zu kopierenden Instanz von Scratch also es sind keine Membervariablen initialisiert oder haben irgendwelche Werte gesetzt, da es sich um eine frische Instanz handelt.

Der Kopierkonstruktor wird aufgerufen wenn (Bsp.: MyClass otherInstance()):

1. MyClass c = otherInstance;
2. MyClass c(otherInstance);
3. Function(otherinstance) // Da call by value

Ein Kopierkonstruktor hat die Syntax:

```
<CLASS> :: <CLASS>( <CLASS> & )
```

## 1.2 Assign Operator

Der Assign Operator wird ebenfalls vom Compiler erstellt und hat dieselbe Problematik mit Pointer wie der Kopierkonstruktor. Jedoch wird hierbei einer bereits existierenden Instanz der State der zu kopierenden Instanz übertragen, was dazu führt das etwaige bereits existierende Membervariablen vorherig ordentlich gelöscht werden müssen (sofern erforderlich, Pointer), ansonsten verursacht man einen Memoryleak, da Referenzen verloren gehen würden.

Ein Assign Operator hat die Syntax:

```
<CLASS> & <CLASS> :: operator= ( <CLASS> )
<CLASS> & <CLASS> :: operator= ( <CLASS> & )
```

## 1.3 Implementierung

Folgend sind die allgemeinen Implementierung angeführt.

### 1.3.1 TreeTestsMain.cpp

Folgend ist die Implementierung des Test Handlers angeführt, welcher die Test Suite erstellt und die Tests durchführt.

```
/*
 * TreeTestsMain.cpp
 * This is the implementation which creates the test suite and runs all of the implemented
 tests.
 *
 * Created on: Dec 23, 2014
 * Author: Thomas Herzog
 */

#include "cute.h"
#include "ide_listener.h"
#include "xml_listener.h"
#include "cute_runner.h"
#include "TreeTests.h"
#include "FileSystemTest.h"

using namespace cute;

static void runSuite(int argc, char const *argv[]) {
    suite s;
    xml_file_opener xmlfile(argc, argv);
    xml_listener<ide_listener<> > lis(xmlfile.out);

    make_suite_TreeTests(&s);
    make_suite_FileSystemTests(&s);
    makeRunner(lis, argc, argv)(s, "AllTests");
}

int main(int argc, char const *argv[]) {
    runSuite(argc, argv);
}
```

## 2 Allgemeine Bäume

Folgend ist die Dokumentation für die Aufgabe Allgemeine Bäume angeführt.

### 2.1 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 Implementierungen.

Die Tree Implementierung soll um Methoden erweitert werden, die es erleichtern in dieser Baumstruktur zu navigieren. Auf jeden Fall soll darauf geachtet werden, dass diese nicht nach außen sichtbar sind sondern höchstens geschützt innerhalb der Vererbungshierarchie, wenn man davon ausgehen kann dass diese Methoden in einer abgeleiteten Klasse benötigt werden können und nicht die Implementierung negativ beeinflussen. Die Member Variablen sollen auf jeden Fall nur über Getter Methoden nach außen zugreifbar sein oder höchstens geschützt innerhalb der Vererbungshierarchie sichtbar sein.

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 ein Baum keine Zyklen enthalten darf und daher auch keine HeapOverflows zu erwarten sind. Daher soll hierbei der Validierung des Baumes besondere Aufmerksamkeit geschenkt werden, da man sich bei diesen Ansatz darauf verlässt, dass es sich immer um einen gültigen Baum handelt.

Es soll eine clone() Methode implementiert werden die eine vollständige Kopie der aktuellen Instanz erstellt. Diese clone Methode könnte auch außerhalb der Implementierung verwendet werden.

Da ein Tree auch ein gültiger Tree ist, der auch keine Knoten enthalten kann, soll hier die Möglichkeit offen gelassen werden eine Tree Instanz zu erstellen, die keinen Knoten besitzt, also auch keinen Root besitzt. 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 kein neuer Knoten angelegt werden.

Hierbei ist der Aufrufer für die Verwaltung der nicht mehr referenzierten Knoten verantwortlich.

**DeleteElements:**

Hier sollen alle Knoten gelöscht 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 man für würdig erachtet in einer abgeleiteten Klasse überschrieben zu werden sollen mit dem Schlüsselwort virtual versehen werden.

Des Weiteren sollen alle Methoden ausführlich dokumentiert werden.

## 2.2 Implementierung

Folgend sind die Sources der ersten Aufgabenstellung angeführt.

### 2.2.1 Tree.h

Folgend ist die Spezifikation von Tree angeführt.

```

/*
 * Tree.h
 * The specification of the Tree implementation.
 *
 * Created on: Dec 23, 2014
 * Author: Thomas Herzog
 */

#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 = nullptr;

    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 start the search from.
     * @param
     *     searchNode: the node to be searched
     * @return
     *     true if the node is managed, false otherwise
     */
    bool isManagedNode(const Node* node, const Node* searchNode) 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.
 */
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.
 */
virtual Node* getRoot() const;

/**
 * Setter for 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: the child node to be appended to the given parent node.
    */
    virtual void insertChild(Node* parent, Node* child);

    /**
    * Deletes the subtree held by the 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 ostream instance to 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_ */

```



## 2.2.2 Tree.cpp

Folgend ist die Implementierung der Spezifikation Tree.h angeführt.

```

/*
 * Tree.cpp
 * The implementation of the Tree specification.
 * Created on: Dec 23, 2014
 * Author: Thomas Herzog
 */

#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 Node* searchNode) const {
    bool isManaged = false;
    /* recursion anchor */
    if (node != nullptr) {
        isManaged = (node == searchNode);
        if (!isManaged) {
            Node* tmp = node->getFirstChild();
            while ((tmp != nullptr) && (!isManaged)) {
                isManaged = (tmp == searchNode);
                if (tmp->getFirstChild() != nullptr) {
                    isManaged = isManagedNode(tmp, searchNode);
                } /* if */
                tmp = tmp->getNextSibling();
            } /* while */
        } /* if */
    } /* if */
    return isManaged;
} /* 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 */
    } /* 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();
    } /* if */
    Register(TREE_CLASS, OBJECT_CLASS);
} /* Tree::Tree */

Tree::~Tree() {
    if (root != nullptr) {

```

```

        delete root;
    } /* if */
} /* 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;
    }
    /* if current root node is not null delete it */
    else if (root != nullptr) {
        deleteElements();
        /* clear tree */
        clear();
    } /* if */

} /* 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 */
    else if ((parent == nullptr) || (child == nullptr)) {
        cout << "Parent and child are not allowed to be null" << endl << flush;
    }
    /* Check for cyclic references */
    else 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;
    }
    /* Check if parent node is managed by this tree */
    else if (!isManagedNode(root, parent)) {
        cout << "Parent is not managed by this tree" << endl << flush;
    }
    /* Check if child already managed by this tree */
    else if (isManagedNode(root, child)) {
        cout << "Child already managed by this tree" << endl << flush;
    }
    /* set as first child if no first child present */
    else if (parent->getFirstChild() == nullptr) {
        parent->setFirstChild(child);
        size++;
    }
    /* 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);
        size++;
    } /* if */
} /* Tree::insertChild */

```

```

void Tree::deleteSubTree(Node* node) {
    bool isRoot = false;
    Node* tmp = nullptr;

    /* Error if instance is not managed */
    if (!isManagedNode(root, 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 = nullptr;
    size = 0;
} /* 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 << "    ";
    node = (*parentIterator)->getFirstChild();
    while (node != nullptr) {
        os << *node;
        if (node->getNextSibling() != nullptr) {
            os << " - ";
        } /* if */
        node = node->getNextSibling();
    } /* while */
    parentIterator++;
    cout << endl;
} /* 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;
        } /* if */
        root = other.getRoot()->clone();
        size = other.getSize();
    } /* if */
    return *this;
} /* Tree::operator= */

```

### 2.2.3 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
     *     ostream: the ostream 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_ */

```

## 2.2.4 Node.cpp

Folgend 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 */

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);
} /* Node::Node */

////////////////////////////////////
// Getter and Setter //
////////////////////////////////////
Node* Node::getFirstChild() const {
    return firstChild;
} /* Node::getFirstChild */

Node* Node::getNextSibling() const {
    return nextSibling;
} /* Node::getNextSibling */

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 */
} /* Node::setFirstChild */

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 */
} /* Node::setNextSibling */

////////////////////////////////////
// Utils //
////////////////////////////////////
void Node::print(ostream & os) const {
    os << AsString();
}

```



```

} /* Node::print */

Node* Node::clone() const {
    return new Node(*this);
} /* Node::clone */

////////////////////////////////////
// operator                               //
////////////////////////////////////
ostream& operator<<(ostream & os, const Node & node) {
    node.print(os);
    return os;
} /* operator<< */

```

## 2.2.5 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_ */

```

## 2.2.6 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 */

```

## 2.2.7 TreeTests.h

Folgend ist die Spezifikation für die Tree Tests angeführt.

```

/*
 * TreeTests.h
 * This is the specification for the tree tests.
 *
 * Created on: Dec 23, 2014
 * Author: Thomas Herzog
 */
#ifndef TREETESTS_H
#define TREETESTS_H

#include "cute.h"
#include <ostream>
#include <string>
#include "../Tree.h"
#include "../Node.h"
#include "../IntNode.h"

/**
 * This function puts the test function of this specification into
 * the given test suite which gets invoked by the test handler
 *
 * @param
 *      s: the pointer to the suite where to add the test function references (is a vector)
 */
void make_suite_TreeTests(cute::suite* s);

////////////////////////////////////
// insertChild
////////////////////////////////////
void test_insertChild_null_parent();

void test_insertChild_null_child();

void test_insertChild_unmanaged_parent();

void test_insertChild_unmanaged_child();

////////////////////////////////////
// clear
////////////////////////////////////
void test_clear();

////////////////////////////////////
// deleteElements
////////////////////////////////////
void test_deleteElements();

////////////////////////////////////
// deleteSubtree
////////////////////////////////////
void test_deleteSubtree_root();

void test_deleteSubtree_child();

void test_deleteSubtree_sibling();

////////////////////////////////////
// Examples and operators
////////////////////////////////////
void test_tree_example();

////////////////////////////////////
// print tree
////////////////////////////////////
void test_print_example();

////////////////////////////////////
// constructor and assign operator
////////////////////////////////////
void test_copy_constructor();

void test_assign_operator();

```

```
#endif /* TREETESTS_H_ */
```

## 2.2.8 TreeTest.cpp

Folgend ist die Implementierung der Spezifikation TreeTests.h angeführt.

```
/*
 * TreeTest.h
 * This is the implementation of the tree tests.
 *
 * Created on: Dec 23, 2014
 * Author: Thomas Herzog
 */
#include "TreeTests.h"
#include "MetaInfo.h"

using namespace std;
using namespace ML;

void make_suite_TreeTests(cute::suite* s) {
    (*s).push_back(CUTE(test_insertChild_null_parent));
    (*s).push_back(CUTE(test_insertChild_null_child));
    (*s).push_back(CUTE(test_insertChild_unmanaged_parent));
    (*s).push_back(CUTE(test_insertChild_unmanaged_child));
    (*s).push_back(CUTE(test_clear));
    (*s).push_back(CUTE(test_deleteElements));
    (*s).push_back(CUTE(test_deleteElements));
    (*s).push_back(CUTE(test_clear));
    (*s).push_back(CUTE(test_deleteSubtree_child));
    (*s).push_back(CUTE(test_deleteSubtree_root));
    (*s).push_back(CUTE(test_deleteSubtree_sibling));
    (*s).push_back(CUTE(test_tree_example));
    (*s).push_back(CUTE(test_print_example));
    (*s).push_back(CUTE(test_copy_constructor));
    (*s).push_back(CUTE(test_assign_operator));
}

void test_insertChild_null_parent() {
    Tree* tree = new Tree((*new Node()));
    Node* child = new Node();
    tree->insertChild(nullptr, child);

    ASSERT_EQUAL(1, tree->getSize());

    delete child;
    delete tree;
}

void test_insertChild_null_child() {
    Tree* tree = new Tree((*new Node()));
    Node* parent = new Node();
    tree->insertChild(parent, nullptr);

    ASSERT_EQUAL(1, tree->getSize());

    delete parent;
    delete tree;
}

void test_insertChild_unmanaged_parent() {
    Node* root = new Node();
    Tree* tree = new Tree(*root);
    Node* unmanaged = new Node();
    Node* newChild = new Node();

    tree->insertChild(tree->getRoot(), new Node());
    Node* n3 = new Node();
    tree->insertChild(tree->getRoot(), n3);
    Node* n2 = new Node();
    tree->insertChild(tree->getRoot(), n2);

    tree->insertChild(n2, new Node());
    tree->insertChild(n2, new Node());

    Node* n4 = new Node();
```

```

    tree->insertChild(n3, new Node());
    tree->insertChild(n3, new Node());
    tree->insertChild(n3, n4);
    tree->insertChild(unmanaged, newChild);

    delete unmanaged;
    delete newChild;
    delete tree;
}

void test_insertChild_unmanaged_child() {
    Node* root = new Node();
    Tree* tree = new Tree(*root);
    Node* newChild = new Node();

    tree->insertChild(tree->getRoot(), new Node());
    Node* n3 = new Node();
    tree->insertChild(tree->getRoot(), n3);
    Node* n2 = new Node();
    tree->insertChild(tree->getRoot(), n2);

    tree->insertChild(n2, new Node());
    tree->insertChild(n2, new Node());

    Node* n4 = new Node();
    tree->insertChild(n3, new Node());
    tree->insertChild(n3, new Node());
    tree->insertChild(n3, n4);
    tree->insertChild(n3, n4);

    delete newChild;
    delete tree;
}

void test_clear() {
    Tree* tree = new Tree(*new Node());
    tree->insertChild(tree->getRoot(), new Node());
    tree->insertChild(tree->getRoot(), new Node());
    tree->insertChild(tree->getRoot(), new Node());

    Node* root = tree->getRoot();
    tree->clear();

    ASSERT_EQUAL(0, tree->getSize());

    delete root;
    delete tree;
}

void test_deleteElements() {
    Tree* tree = new Tree(*new Node());
    tree->insertChild(tree->getRoot(), new Node());
    tree->insertChild(tree->getRoot(), new Node());
    tree->insertChild(tree->getRoot(), new Node());
    tree->insertChild(tree->getRoot(), new Node());
    tree->insertChild(tree->getRoot(), new Node());
    tree->insertChild(tree->getRoot(), new Node());

    tree->deleteElements();

    ASSERT_EQUAL(0, tree->getSize());

    delete tree;
}

void test_deleteSubtree_root() {
    Tree* tree = new Tree(*new Node());
    Node* subParent = new Node();
    tree->insertChild(tree->getRoot(), new Node());
    tree->insertChild(tree->getRoot(), subParent);
    tree->insertChild(tree->getRoot(), new Node());
    tree->insertChild(subParent, new Node());
    tree->insertChild(subParent, new Node());
    tree->insertChild(subParent, new Node());

    tree->deleteSubTree(tree->getRoot());
}

```

```

    ASSERT_EQUAL(0, tree->getSize());
    delete tree;
}

void test_deleteSubtree_child() {
    Tree* tree = new Tree((*new Node()));
    Node* l1 = new Node();
    Node* l2 = new Node();
    Node* l3 = new Node();

    tree->insertChild(tree->getRoot(), l1);
    tree->insertChild(l1, l2);
    tree->insertChild(l2, l3);
    tree->insertChild(l2, new Node());
    tree->insertChild(l2, new Node());
    tree->insertChild(l2, new Node());

    tree->deleteSubTree(l2);
    ASSERT_EQUAL(2, tree->getSize());
    delete tree;
}

void test_deleteSubtree_sibling() {
    Tree* tree = new Tree((*new Node()));
    Node* subParent = new Node();
    tree->insertChild(tree->getRoot(), new Node());
    tree->insertChild(tree->getRoot(), subParent);
    tree->insertChild(tree->getRoot(), new Node());
    tree->insertChild(subParent, new Node());
    tree->insertChild(subParent, new Node());
    tree->insertChild(subParent, new Node());

    tree->deleteSubTree(subParent);
    ASSERT_EQUAL(3, tree->getSize());
    delete tree;
}

void test_tree_example() {
    Node* root = new IntNode(1);
    Tree* tree = new Tree(*root);

    tree->insertChild(tree->getRoot(), new IntNode(4));
    Node* n3 = new IntNode(3);
    tree->insertChild(tree->getRoot(), n3);
    Node* n2 = new IntNode(2);
    tree->insertChild(tree->getRoot(), n2);

    tree->insertChild(n2, new IntNode(5));
    tree->insertChild(n2, new IntNode(6));

    tree->insertChild(n3, new IntNode(7));
    tree->insertChild(n3, new IntNode(8));
    tree->insertChild(n3, new IntNode(9));
    tree->insertChild(n3, new IntNode(10));

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

    delete tree;
}

void test_print_example() {
    Node* root = new IntNode(1);
    Tree* tree = new Tree(*root);

    Node* n2 = new IntNode(2);
    tree->insertChild(tree->getRoot(), n2);
    Node* n3 = new IntNode(3);
    tree->insertChild(tree->getRoot(), n3);
    tree->insertChild(tree->getRoot(), new IntNode(4));

    tree->insertChild(n2, new IntNode(5));
    tree->insertChild(n2, new IntNode(6));

    tree->insertChild(n3, new IntNode(7));

```

```

    tree->insertChild(n3, new IntNode(8));
    tree->insertChild(n3, new IntNode(9));
    tree->insertChild(n3, new IntNode(10));

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

    delete tree;
}

void test_copy_constructor() {
    Tree tree(new IntNode(1));

    tree.insertChild(tree.getRoot(), new IntNode(4));
    Node* n3 = new IntNode(3);
    tree.insertChild(tree.getRoot(), n3);
    Node* n2 = new IntNode(2);
    tree.insertChild(tree.getRoot(), n2);

    tree.insertChild(n2, new IntNode(5));
    tree.insertChild(n2, new IntNode(6));

    tree.insertChild(n3, new IntNode(7));
    tree.insertChild(n3, new IntNode(8));
    tree.insertChild(n3, new IntNode(9));
    tree.insertChild(n3, new IntNode(10));

    cout << "Original tree: " << endl;
    cout << tree << endl;

    cout << "Copying tree via copy constructor" << endl;
    Tree copied = tree;

    cout
        << "Copied tree: (Deep copy no references are shared between instances)"
        << endl;
    cout << copied;

    tree.deleteElements();
    copied.deleteElements();
}

void test_assign_operator() {
    Tree tree(new IntNode(1));

    tree.insertChild(tree.getRoot(), new IntNode(4));
    Node* n3 = new IntNode(3);
    tree.insertChild(tree.getRoot(), n3);
    Node* n2 = new IntNode(2);
    tree.insertChild(tree.getRoot(), n2);

    tree.insertChild(n2, new IntNode(5));
    tree.insertChild(n2, new IntNode(6));

    tree.insertChild(n3, new IntNode(7));
    tree.insertChild(n3, new IntNode(8));
    tree.insertChild(n3, new IntNode(9));
    tree.insertChild(n3, new IntNode(10));

    cout << "Copying the original tree by performing a copy via operator="
        << endl;
    /* need to first declare otherwise copy constructor would be invoked */
    Tree referenced;
    /* Copy via assign operator */
    referenced = tree;

    cout << "Original tree:" << endl << tree << endl << flush;
    cout
        << "Copied tree: (Deep copy no references are shared between instances)"
        << endl << tree << endl << flush;

    tree.deleteElements();
    referenced.deleteElements();
}

```




## 2.3 Tests

Folgend sind die Tests für die Tree Implementierung und dessen Node Spezifikationen Node und IntNode angeführt.

Diese Tests wurden mit einer Testsuite erstellt und die Testfunktionen sollten selbstsprechend sein und wiedergeben.

Für weitere Informationen sei auf den vorherig angeführten Source in verwiesen.



```

<terminated> Trees [C/C++ Application] /mnt/hgfs/cchet/Docum

#beginning AllTests 35

#starting test_insertChild null_parent
Parent and child are not allowed to be null

#success test_insertChild_null_parent OK

#starting test_insertChild null_child
Parent and child are not allowed to be null

#success test_insertChild_null_child OK

#starting test_insertChild_unmanaged_parent
Parent is not managed by this tree

#success test_insertChild_unmanaged_parent OK

#starting test_insertChild unmanaged_child
Child already managed by this tree

#success test_insertChild_unmanaged_child OK

#starting test_clear

#success test_clear OK

#starting test_deleteElements

#success test_deleteElements OK

#starting test_deleteElements

#success test_deleteElements OK

#starting test_clear

#success test_clear OK

#starting test_deleteSubtree_child

#success test_deleteSubtree_child OK

#starting test_deleteSubtree_root

#success test_deleteSubtree_root OK

#starting test_deleteSubtree_sibling

#success test_deleteSubtree_sibling OK
    
```

```

Problems Tasks Console Properties 1010 0101 Call Graph Search
<terminated> Tree [C/C++ Application] /mnt/hgfs/cchet/Documents/Git Reposito

#starting test_tree_example
#####
node count: 10
#####
Level: 0
Parent: (root has not parent)
Nodes: IntNode(1)

-----
Level: 1
-----
Parent: IntNode(1)
       IntNode(2) - IntNode(3) - IntNode(4)

-----
Level: 2
-----
Parent: IntNode(2)
       IntNode(6) - IntNode(5)
Parent: IntNode(3)
       IntNode(10) - IntNode(9) - IntNode(8) - IntNode(7)

#success test_tree_example OK

#starting test_print_example
#####
node count: 10
#####
Level: 0
Parent: (root has not parent)
Nodes: IntNode(1)

-----
Level: 1
-----
Parent: IntNode(1)
       IntNode(4) - IntNode(3) - IntNode(2)

-----
Level: 2
-----
Parent: IntNode(3)
       IntNode(10) - IntNode(9) - IntNode(8) - IntNode(7)
Parent: IntNode(2)
       IntNode(6) - IntNode(5)

#success test_print_example OK

```

```

Problems Tasks Console Properties Call Graph Search
<terminated> Tree [C/C++ Application] /mnt/hgfs/cchet/Documents/Git Repositor

#starting test_copy_constructor
Original tree:
#####
node count: 10
#####
Level: 0
Parent: (root has not parent)
Nodes: IntNode(1)

-----
Level: 1
-----
Parent: IntNode(1)
      IntNode(2) - IntNode(3) - IntNode(4)

-----
Level: 2
-----
Parent: IntNode(2)
      IntNode(6) - IntNode(5)
Parent: IntNode(3)
      IntNode(10) - IntNode(9) - IntNode(8) - IntNode(7)

Copying tree via copy constructor
Copied tree: (Deep copy no references are shared between instances)
#####
node count: 10
#####
Level: 0
Parent: (root has not parent)
Nodes: IntNode(1)

-----
Level: 1
-----
Parent: IntNode(1)
      IntNode(2) - IntNode(3) - IntNode(4)

-----
Level: 2
-----
Parent: IntNode(2)
      IntNode(6) - IntNode(5)
Parent: IntNode(3)
      IntNode(10) - IntNode(9) - IntNode(8) - IntNode(7)

#success test_copy_constructor OK

```

```

Problems Tasks Console Properties Call Graph Search
<terminated> Tree [C/C++ Application] /mnt/hgfs/cchet/Documents/Git Repositories:
#starting test_assign_operator
Copying the original tree by performing a copy via operator=
Original tree:
#####
node count: 10
#####
Level: 0
Parent: (root has not parent)
Nodes: IntNode(1)

-----
Level: 1
-----
Parent: IntNode(1)
       IntNode(2) - IntNode(3) - IntNode(4)

-----
Level: 2
-----
Parent: IntNode(2)
       IntNode(6) - IntNode(5)
Parent: IntNode(3)
       IntNode(10) - IntNode(9) - IntNode(8) - IntNode(7)

Copied tree: (Deep copy no references are shared between instances)
#####
node count: 10
#####
Level: 0
Parent: (root has not parent)
Nodes: IntNode(1)

-----
Level: 1
-----
Parent: IntNode(1)
       IntNode(2) - IntNode(3) - IntNode(4)

-----
Level: 2
-----
Parent: IntNode(2)
       IntNode(6) - IntNode(5)
Parent: IntNode(3)
       IntNode(10) - IntNode(9) - IntNode(8) - IntNode(7)

#success test_assign_operator OK

```

### 3 Hierarchisches Dateisystem

Folgend ist die Dokumentation der Aufgabenstellung Hierarchisches Dateisystem angeführt.

#### 3.1 Lösungsidee

Bei dieser Implementierung soll von der Klasse Tree abgeleitet werden.

Da diese Klasse aber Methoden als public definiert hat, 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 Dateisystem 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 mit anderen Variationen der Klasse Node 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 Dateisystem 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 dieser 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 Fehlermeldungen übernimmt. Um den Speicherbedarf gering zu halten soll dieser als Singleton implementiert werden, da hierbei immer nur eine Instanz verwendet werden soll. So erreicht man das die Meldungen (Strings) 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.

## 3.2 Implementierung

Folgend sind die Implementierungen der zweiten Aufgabenstellung angeführt.

### 3.2.1 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_ */
```

### 3.2.2 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 */

```

### 3.2.3 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 constructor
     */
    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
 *      ostream: the ostream to put printed text on
 */
virtual void print(std::ostream & os) const;
};

#endif /* DIRECTORY_H_ */

```

### 3.2.4 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 */

```

### 3.2.5 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
     *     ostream: the ostream 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_ */

```

### 3.2.6 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 */

```

### 3.2.7 FileSystem.h

Folgend 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
     * will be the node visited before the error occurred, otherwise the last visited node
     * 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 exists.
 *
 * @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_ */

```

### 3.2.8 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;

    /* child name is not supposed to be empty */
    if (childName.empty()) {
        result.error = FsErrorType::INVALID_CHILD;
        return result;
    }

    /* 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 */

```

### 3.2.9 FsErrorHandler.h

Folgend ist die Spezifikation und Implementierung von FsErrorHandler 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,
    /* Child name is invalid */
    INVALID_CHILD = 7
};

/**
 * 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"));
    errorMessages.insert(
        std::pair<FsErrorType, std::string>(FsErrorType::INVALID_CHILD,
            "Child name is invalid"));
}

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 */

```

### 3.2.10 FileSystemTest.h

Folgend ist die Spezifikation der FileSystem Tests angeführt.

```

/*
 * FileSystemTest.h
 * This is the specification file system tests.
 *
 * Created on: Dec 27, 2014
 * Author: Thomas Herzog
 */

#ifndef FILESYSTEMTEST_H_
#define FILESYSTEMTEST_H

#include "cute.h"
#include <ostream>
#include <string>
#include "../FileSystem.h"
#include "../File.h"
#include "../FSNode.h"

/**
 * This function puts the test function of this specification into
 * the given test suite which gets invoked by the test handler
 *
 * @param
 *      s: the pointer to the suite where to add the test function references (is a vector)
 */
void make_suite_FileSystemTests(cute::suite* s);

/* Cleans the singletons instance nicely */
void cleanup_tests();

////////////////////////////////////
// mkdir tests
////////////////////////////////////
void test_mkdir_invalid_path();

void test_mkdir_path_contain_files();

void test_mkdir_valid();

////////////////////////////////////
// touch tests
////////////////////////////////////
void test_touch_invalid_path();

void test_touch_path_contain_files();

void test_touch_valid_new();

void test_touch_valid_existing();

////////////////////////////////////
// rm tests
////////////////////////////////////
void test_rm_invalid_path();

```

```

void test_rm_path_contain_files();

void test_rm_not_found();

void test_rm_directory();

void test_rm_valid();

////////////////////////////////////
// rmdir tests
////////////////////////////////////
void test_rmdir_invalid_path();

void test_rmdir_path_contain_files();

void test_rmdir_not_found();

void test_rmdir_file();

void test_rmdir_valid();

////////////////////////////////////
// ls tests
////////////////////////////////////
void test_ls();

////////////////////////////////////
// complex example tests
////////////////////////////////////
void test_complex_example();

#endif /* FILESYSTEMTEST_H_ */

```

### 3.2.11 FileSystemTest.cpp

Folgend ist die Implementierung der Spezifikation FileSystemtests.h angeführt.

```

/*
 * FileSystemTest.cpp
 * This is the implementation of the file system tests specification
 *
 * Created on: Dec 27, 2014
 * Author: Thomas Herzog
 */
#include "FileSystemTest.h"
#include "MetaInfo.h"

using namespace std;
using namespace ML;

void make_suite_FileSystemTests(cute::suite* s) {
    (*s).push_back(CUTE(test_mkdir_invalid_path));
    (*s).push_back(CUTE(test_mkdir_path_contain_files));
    (*s).push_back(CUTE(test_mkdir_valid));

    (*s).push_back(CUTE(test_touch_invalid_path));
    (*s).push_back(CUTE(test_touch_path_contain_files));
    (*s).push_back(CUTE(test_touch_valid_new));
    (*s).push_back(CUTE(test_touch_valid_existing));

    (*s).push_back(CUTE(test_rm_invalid_path));
    (*s).push_back(CUTE(test_rm_path_contain_files));
    (*s).push_back(CUTE(test_rm_not_found));
    (*s).push_back(CUTE(test_rm_directory));
    (*s).push_back(CUTE(test_rm_valid));

    (*s).push_back(CUTE(test_rmdir_invalid_path));
    (*s).push_back(CUTE(test_rmdir_path_contain_files));
    (*s).push_back(CUTE(test_rmdir_not_found));
    (*s).push_back(CUTE(test_rmdir_file));
    (*s).push_back(CUTE(test_rmdir_valid));

    (*s).push_back(CUTE(test_ls));
}

```

```

    (*s).push_back(CUTE(test_complex_example));

    (*s).push_back(CUTE(cleanup_tests));
}

void cleanup_tests() {
    FsErrorHandler::destroyInstance();
}

////////////////////////////////////
// mkdir tests
////////////////////////////////////
void test_mkdir_invalid_path() {
    FileSystem* fs = new FileSystem();
    fs->mkdir("NOT_EXISTING/Downloads", "NEW_FOLDER");

    cout << *fs << endl;

    delete fs;
}

void test_mkdir_path_contain_files() {
    FileSystem* fs = new FileSystem();
    fs->mkdir("", "home");
    fs->mkdir("home", "cchet");
    fs->mkdir("home/cchet", "Documents");
    fs->touch("home/cchet", "f1");

    fs->mkdir("home/cchet/f1/Documents", "docs");

    delete fs;
}

void test_mkdir_valid() {
    FileSystem* fs = new FileSystem();
    fs->mkdir("", "root");
    fs->mkdir("", "home");
    fs->mkdir("home", "cchet");
    fs->mkdir("home/cchet", "Documents");
    fs->mkdir("home/cchet", "Downloads");
    fs->mkdir("home/cchet/Downloads", "eclipse-stuff");

    cout << *fs << endl;

    delete fs;
}

////////////////////////////////////
// touch tests
////////////////////////////////////
void test_touch_invalid_path() {
    FileSystem* fs = new FileSystem();
    fs->touch("home", "testfile");

    cout << *fs << endl;

    delete fs;
}

void test_touch_path_contain_files() {
    FileSystem* fs = new FileSystem();
    fs->mkdir("", "root");
    fs->mkdir("", "home");
    fs->mkdir("home", "cchet");
    fs->touch("home", "PART_OF_PATH");
    fs->mkdir("home/PART_OF_PATH/cchet", "Documents");

    cout << *fs << endl;

    delete fs;
}

void test_touch_valid_new() {
    FileSystem* fs = new FileSystem();
    fs->touch("", "f1");

```

```

    fs->touch("", "f2");
    fs->touch("", "f3");

    delete fs;
}

void test_touch_valid_existing() {
    FileSystem* fs = new FileSystem();
    fs->touch("", "f1");
    fs->touch("", "f1");

    delete fs;
}

////////////////////////////////////
// rm tests                                     //
////////////////////////////////////
void test_rm_invalid_path() {
    FileSystem* fs = new FileSystem();
    fs->touch("", "f1");

    fs->rm("home", "f1");

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

    delete fs;
}

void test_rm_path_contain_files() {
    FileSystem* fs = new FileSystem();
    fs->mkdir("", "home");
    fs->mkdir("home", "cchet");
    fs->mkdir("home/cchet", "Documents");
    fs->touch("home/cchet", "f1");
    fs->touch("home/cchet/Documents", "f2");

    fs->rm("home/cchet/f1/Documents", "f2");

    delete fs;
}

void test_rm_not_found() {
    FileSystem* fs = new FileSystem();
    fs->mkdir("", "home");
    fs->mkdir("home", "cchet");
    fs->touch("home/cchet", "f1");

    fs->rm("home/cchet", "f2");

    delete fs;
}

void test_rm_directory() {
    FileSystem* fs = new FileSystem();
    fs->mkdir("", "home");
    fs->mkdir("home", "cchet");
    fs->touch("home/cchet", "f1");

    fs->rm("home", "cchet");

    delete fs;
}

void test_rm_valid() {
    FileSystem* fs = new FileSystem();
    fs->touch("", "f1");
    fs->touch("", "f2");

    fs->rm("", "f1");

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

    delete fs;
}

////////////////////////////////////
// rmdir tests                                     //

```



```

////////////////////////////////////
void test_rmdir_invalid_path() {
    FileSystem* fs = new FileSystem();
    fs->mkdir("", "home");
    fs->mkdir("home", "cchet");
    fs->mkdir("home/cchet", "Documents");
    fs->mkdir("home/cchet/Documents", "Receipts");

    fs->rmdir("home/other/Documents", "Receipts");

    delete fs;
}

void test_rmdir_path_contain_files() {
    FileSystem* fs = new FileSystem();
    fs->mkdir("", "home");
    fs->mkdir("home", "cchet");
    fs->mkdir("home/cchet", "Documents");
    fs->touch("home/cchet", "f1");
    fs->mkdir("home/cchet/Documents", "Receipts");

    fs->rmdir("home/cchet/f1/Documents", "Receipts");

    delete fs;
}

void test_rmdir_not_found() {
    FileSystem* fs = new FileSystem();
    fs->mkdir("", "home");
    fs->mkdir("home", "cchet");
    fs->mkdir("home/cchet", "Documents");
    fs->mkdir("home/cchet/Documents", "Receipts");

    fs->rmdir("home/cchet/Documents", "blabla");

    delete fs;
}

void test_rmdir_file() {
    FileSystem* fs = new FileSystem();
    fs->mkdir("", "home");
    fs->mkdir("home", "cchet");
    fs->mkdir("home/cchet", "Documents");
    fs->touch("home/cchet", "f1");
    fs->mkdir("home/cchet/Documents", "Receipts");

    fs->rmdir("home/cchet/", "f1");

    delete fs;
}

void test_rmdir_valid() {
    FileSystem* fs = new FileSystem();
    fs->mkdir("", "home");
    fs->mkdir("home", "cchet");
    fs->mkdir("home/cchet", "Documents");
    fs->touch("home/cchet", "f1");
    fs->touch("home/cchet", "f2");
    fs->touch("home/cchet", "f3");
    fs->touch("home/cchet", "f4");

    fs->rmdir("home", "cchet");

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

    delete fs;
}

////////////////////////////////////
// ls tests
////////////////////////////////////
void test_ls() {
    FileSystem* fs = new FileSystem();
    fs->mkdir("", "home");
    fs->mkdir("home", "cchet");
    fs->mkdir("home/cchet", "Documents");

```

```
fs->touch("home/cchet", "f1");
fs->touch("home/cchet", "f2");
fs->touch("home/cchet", "f3");
fs->touch("home/cchet", "f4");

fs->ls();

delete fs;
}

////////////////////////////////////
// complex example tests
////////////////////////////////////
void test_complex_example() {
    FileSystem* fs = new FileSystem();
    fs->mkdir("", "root");
    fs->touch("root", "ReadMe.txt");
    fs->mkdir("root", "bin");
    fs->touch("root/bin", "a.exe");
    fs->touch("root/bin", "b.exe");
    //fs->ls();
    fs->rm("root/bin", "a.exe");
    fs->rmdir("", "root");
    //fs->ls();
    fs->rm("root/bin", "b.exe");
    fs->rmdir("", "root/bin");
    fs->rm("root", "ReadMe.txt");
    fs->rmdir("", "root");
    fs->ls();

    delete fs;
}
```

### 3.3 Tests

Folgend sind die Tests für die FileSystem Implementierung angeführt.

Auch diese Tests wurden mit der Test Suite erstellt und. Auch diese Testfunktionen haben selbstsprechende Name die widerspiegeln was getestet wird.

Für weitere Informationen sei auf den vorherig angeführten Source verwiesen.

*Achtung: Sollte man bei diesen Tests WriteMetaInfo() hinzufügen wird man ab den ersten Mal wo die FsErrorHandler Singleton Instanz erstellt wurde (Erster Test der eine Fehlermeldung liefert) immer die Meldung bekommen, dass noch ein Objekt existiert und nicht gelöscht wurde. Dies liegt daran, dass die Singleton Instanz erst gelöscht wird wenn alle Tests abgeschlossen sind (test\_cleanup), also kurz vor Programmende.*



```

Problems Tasks Console Properties Call Graph Search
<terminated> Tree [C/C++ Application] /mnt/hgfs/cchet/Documents/Git Repo
#beginning AllTests 20

#starting test_mkdir_invalid_path
singleton instance created
Given path is invalid !!! (last visited node: not available)
#####
node count: 1
#####
Level: 0
Parent: (root has not parent)
Nodes: Directory('')

-----
Level: 1
-----
Parent: Directory('')

#success test_mkdir_invalid_path OK

#starting test_mkdir_path_contain_files
directory created 'home'
directory created 'cchet'
directory created 'Documents'
touched newly created file 'f1'
returning existing singleton instance
Cannot walk path which contains files (last visited node: f1)

#success test_mkdir_path_contain_files OK

```

```

Problems Tasks Console Properties Call Graph Search
<terminated> Tree [C/C++ Application] /mnt/hgfs/cchet/Documents/Git Repos
#starting test_mkdir_path_contain_files
directory created 'home'
directory created 'cchet'
directory created 'Documents'
touched newly created file 'f1'
returning existing singleton instance
Cannot walk path which contains files (last visited node: f1)

#success test_mkdir_path_contain_files OK

#starting test_mkdir_valid
directory created 'root'
directory created 'home'
directory created 'cchet'
directory created 'Documents'
directory created 'Downloads'
directory created 'eclipse-stuff'
#####
node count: 7
#####
Level: 0
Parent: (root has not parent)
Nodes: Directory('')

-----
Level: 1
-----
Parent: Directory('')
       Directory('home') - Directory('root')

-----
Level: 2
-----
Parent: Directory('home')
       Directory('cchet')

-----
Level: 3
-----
Parent: Directory('cchet')
       Directory('Downloads') - Directory('Documents')

-----
Level: 4
-----
Parent: Directory('Downloads')
       Directory('eclipse-stuff')

#success test_mkdir_valid OK

```

```

Problems Tasks Console Properties Call Graph Search
<terminated> Tree [C/C++ Application] /mnt/hgfs/cchet/Documents/Git Repositories/Gi
#starting test_touch_invalid_path
returning existing singleton instance
Given path is invalid !!! (last visited node: not available)
#####
node count: 1
#####
Level: 0
Parent: (root has not parent)
Nodes: Directory('')

-----
Level: 1
-----
Parent: Directory('')

#success test_touch_invalid_path OK

#starting test_touch_path_contain_files
directory created 'root'
directory created 'home'
directory created 'cchet'
touched newly created file 'PART_OF_PATH'
returning existing singleton instance
Cannot walk path which contains files (last visited node: PART_OF_PATH)
#####
node count: 5
#####
Level: 0
Parent: (root has not parent)
Nodes: Directory('')

-----
Level: 1
-----
Parent: Directory('')
       Directory('home') - Directory('root')

-----
Level: 2
-----
Parent: Directory('home')
       File('PART_OF_PATH') - Directory('cchet')

#success test_touch_path_contain_files OK

```

```

Problems Tasks Console Properties 1010 0101 Call Graph
<terminated> Tree [C/C++ Application] /mnt/hgfs/cchet/Document

#starting test_touch_valid_new
touched newly created file 'f1'
touched newly created file 'f2'
touched newly created file 'f3'

#success test_touch_valid_new OK

#starting test_touch_valid_existing
touched newly created file 'f1'
Touch existing file 'f1'

#success test_touch_valid_existing OK

#starting test_rm_invalid_path
touched newly created file 'f1'
returning existing singleton instance
Given path is invalid !!!
#####
node count: 2
#####
Level: 0
Parent: (root has not parent)
Nodes: Directory('')

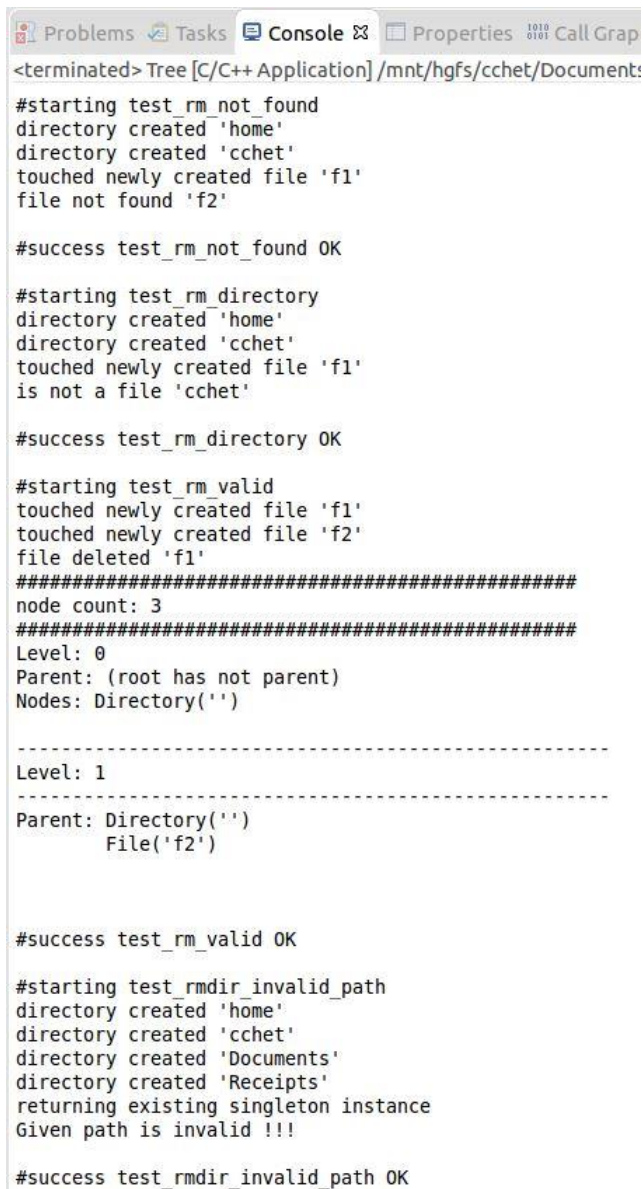
-----
Level: 1
-----
Parent: Directory('')
       File('f1')

#success test_rm_invalid_path OK

#starting test_rm_path_contain_files
directory created 'home'
directory created 'cchet'
directory created 'Documents'
touched newly created file 'f1'
touched newly created file 'f2'
returning existing singleton instance
Cannot walk path which contains files

#success test_rm_path_contain_files OK

```



The screenshot shows a console window with tabs for Problems, Tasks, Console, Properties, and Call Graph. The Console tab is active, displaying the output of a C++ application. The output consists of several test cases and their results, including directory creation, file touching, file deletion, and path validation. The tests are: test\_rm\_not\_found, test\_rm\_directory, test\_rm\_valid, and test\_rmdir\_invalid\_path. The test\_rm\_valid test includes a detailed tree structure output showing the root node and its children.

```
<terminated> Tree [C/C++ Application] /mnt/hgfs/cchet/Documents:

#starting test_rm_not_found
directory created 'home'
directory created 'cchet'
touched newly created file 'f1'
file not found 'f2'

#success test_rm_not_found OK

#starting test_rm_directory
directory created 'home'
directory created 'cchet'
touched newly created file 'f1'
is not a file 'cchet'

#success test_rm_directory OK

#starting test_rm_valid
touched newly created file 'f1'
touched newly created file 'f2'
file deleted 'f1'
#####
node count: 3
#####
Level: 0
Parent: (root has not parent)
Nodes: Directory('')

-----
Level: 1
-----
Parent: Directory('')
       File('f2')

#success test_rm_valid OK

#starting test_rmdir_invalid_path
directory created 'home'
directory created 'cchet'
directory created 'Documents'
directory created 'Receipts'
returning existing singleton instance
Given path is invalid !!!

#success test_rmdir_invalid_path OK
```



```

Problems Tasks Console Properties
<terminated> Tree [C/C++ Application] /mnt/hgfs/cchet
#starting test_rmdir_path_contain_files
directory created 'home'
directory created 'cchet'
directory created 'Documents'
touched newly created file 'f1'
directory created 'Receipts'
returning existing singleton instance
Cannot walk path which contains files

#success test_rmdir_path_contain_files OK

#starting test_rmdir_not_found
directory created 'home'
directory created 'cchet'
directory created 'Documents'
directory created 'Receipts'
directory not found 'blabla'

#success test_rmdir_not_found OK

#starting test_rmdir_file
directory created 'home'
directory created 'cchet'
directory created 'Documents'
touched newly created file 'f1'
directory created 'Receipts'
is not a directory 'f1'

#success test_rmdir_file OK

```

```

Problems Tasks Console Properties 1010 0101 Call Graph Search
<terminated> Tree [C/C++ Application] /mnt/hgfs/cchet/Documents/Git Repositories/Github/FH-Hag
#starting test_rmdir_valid
directory created 'home'
directory created 'cchet'
directory created 'Documents'
touched newly created file 'f1'
touched newly created file 'f2'
touched newly created file 'f3'
touched newly created file 'f4'
directory is not empty
#####
node count: 8
#####
Level: 0
Parent: (root has not parent)
Nodes: Directory('')

-----
Level: 1
-----
Parent: Directory('')
       Directory('home')

-----
Level: 2
-----
Parent: Directory('home')
       Directory('cchet')

-----
Level: 3
-----
Parent: Directory('cchet')
       File('f4') - File('f3') - File('f2') - File('f1') - Directory('Documents')

#success test_rmdir_valid OK

```



```

Problems Tasks Console Properties Call Graph Search
<terminated> Tree [C/C++ Application] /mnt/hgfs/cchet/Documents/Git Repositories/Github/FH-Ha

#starting test_ls
directory created 'home'
directory created 'cchet'
directory created 'Documents'
touched newly created file 'f1'
touched newly created file 'f2'
touched newly created file 'f3'
touched newly created file 'f4'
#####
node count: 8
#####
Level: 0
Parent: (root has not parent)
Nodes: Directory('')

-----
Level: 1
-----
Parent: Directory('')
       Directory('home')

-----
Level: 2
-----
Parent: Directory('home')
       Directory('cchet')

-----
Level: 3
-----
Parent: Directory('cchet')
       File('f4') - File('f3') - File('f2') - File('f1') - Directory('Documents')

#success test_ls OK

#starting test_complex_example
directory created 'root'
touched newly created file 'ReadMe.txt'
directory created 'bin'
touched newly created file 'a.exe'
touched newly created file 'b.exe'
file deleted 'a.exe'
directory is not empty
file deleted 'b.exe'
directory deleted 'bin'
file deleted 'ReadMe.txt'
directory deleted 'root'
#####
node count: 6
#####
Level: 0
Parent: (root has not parent)
Nodes: Directory('')

-----
Level: 1
-----
Parent: Directory('')

#success test_complex_example OK

#starting cleanup_tests
singleton instance destroyed

#success cleanup_tests OK

#ending AllTests

```