Dokumentation

Die Implementation auf Basis eines Iterators 1.Art widerspricht etwas meinem bisherigen Design, weil ich in Konflikt mit den STL-Iteratoren komme. Da aber bereits feststeht, dass das Praktikum über die Sommerferien mit Hilfe der MFC geschrieben wird und ich auch deren Mengenorganisationen benutzen werden, halte ich meine „Hilfskonstruktion“ für kurzfristig vertretbar.

Als erstes definiere ich einen Datentyp für einen Zeiger auf eine Funktion, die ein Element der Menge bearbeiten kann:

```
    typedef void (*TIteratorFunc) (CRoom*);
```

Jede denkbare Funktion, die diese Signatur aufweist, kann mittels `ForEachDo` über die gesamte Menge `m_Set` iterieren:

```
// emulates basic iterator independent of those from STL
void CHouse::ForEachDo(const TIteratorFunc iteratorProc)
{
    // save current cursor
    TSetCursor itCursor = m_itCursor;

    m_itCursor = m_Set.begin();
    while (m_itCursor != m_Set.end())
    {
        CRoom* room = &(*m_itCursor);
        iteratorProc(room);
        m_itCursor++;
    }

    // restore cursor
    m_itCursor = itCursor;
}
```

Man erkennt recht deutlich, dass bei mir quasi ein Iterator auf einen anderen umgesetzt wird. Recht umständlich ist der Cast für `room`, leider fiel mir keine elegantere Lösung ein.
Eine Beispielverwendung von `ForEachDo` ist die Anzeige aller Elemente der Menge. Zunächst benötigt man eine statische Funktion, die in jedem Durchlauf aufgerufen wird. Ihr Ergebnis (bei mir wird nur ein `string` erzeugt und keine direkte Bildschirmausgabe generiert) muss in einer globalen Variablen abgelegt werden, diese Aufgabe übernimmt `_strShow`.

```
// helper
static ostringstream _strShow;
static void _ShowProc(CRoom* room)
{
    _strShow << room->Show();
}
```

Der Aufruf gestaltet sich einfach (man beachte den `const_cast`-Operator):

```
// same functionality as Show, I just use ForEachDo this time
string CHouse::ShowUsingForEachDo() const
{
    CHouse* house = const_cast<CHouse*>(this);
    house->ForEachDo(_ShowProc);

    return _strShow.str();
}
```

Im Hauptprogramm ist dann kein Unterschied zur bisherigen Funktion `Show` feststellbar:

```
    cout<<myHouse.ShowUsingForEachDo();
    cout<<myHouse.Show();
```

Quellcode

Fast alle Klassen wurden unverändert aus Aufgabe O3.5 übernommen, deshalb drucke ich hier nur House.h, House.cpp und O3-5.cpp ab.

House.h

```
//////////////////////////////////////////////////////
// Softwarebauelemente II, Aufgabe O3.5
//
// author:          Stephan Brumme
// last changes:    June 27, 2001

#if !defined(AFX_HOUSE_H__A8EA7860_08E1_11D5_9BB7_A8652B9FDD20__INCLUDED_)
#define AFX_HOUSE_H__A8EA7860_08E1_11D5_9BB7_A8652B9FDD20__INCLUDED_


#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000


// STL's list container
#include <list>

// CRoom used as member variable
#include "Room.h"
// CDate used as member variable
#include "Date.h"


class CHouse : public CBasicClass
{
    // define new types for using the set
    typedef list<CRoom>                 TSetOfRooms;
    typedef TSetOfRooms::iterator       TSetCursor;
    typedef TSetOfRooms::const_iterator TSetConstCursor;

    // my own iterator
    typedef void (*TIteratorFunc) (CRoom*);

public:
    // constructs a new house using its date of foundation (default: today)
    CHouse();
    CHouse(const CHouse& house);
    CHouse(const CDate& dtDateOfFoundation);

    // return class name
    virtual string ClassnameOf() const { return "CHouse"; }

    // display the attributes
    virtual string Show() const;
    // only for internal purposes !
    virtual string ShowDebug() const;

    // validate a house
    virtual bool ClassInvariant() const;

    // compare two houses
    bool operator==(const CHouse& house) const;
    virtual bool EqualValue(const CBasicClass* pClass) const;

    // copy one house to another one
    CHouse&  operator=(const CHouse &house);
    virtual bool Copy(const CBasicClass* pClass);

    // return date of foundation
    CDate GetDateOfFoundation() const;
    // return number of stored rooms
    int Card() const;
```

```
    // insert a new room into the set, TRUE if successful
    // will fail if room already exists
    bool Insert(const CRoom& room);
    // return first stored room, TRUE if successful
    bool GetFirst(CRoom& room);
    // return next room, TRUE if successful
    bool GetNext(CRoom& room);
    // look for a room and set cursor, TRUE if successful
    bool Find(const CRoom& room);
    // return the room the cursor points to, TRUE if successful
    bool GetCurrent(CRoom& room) const;
    // erase current room, TRUE if successful
    bool Scratch();

    // iterate through the set
    void ForEachDo(const TIteratorFunc iterator);
    // show elements by calling ForEachDo
    string ShowUsingForEachDo() const;

private:
    CDate m_dtDateOfFoundation;

    TSetOfRooms m_Set;
    TSetCursor  m_itCursor;
};

#endif // !defined(AFX_HOUSE_H__A8EA7860_08E1_11D5_9BB7_A8652B9FDD20__INCLUDED_)
```

House.cpp

source file "House.cpp"

```
////////////////////////////////////////////////////////
// Softwarebauelemente II, Aufgabe O3.5
//
// author:          Stephan Brumme
// last changes:    June 27, 2001


#include "House.h"
#include "Exception.h"

// needed for std::find
#include <algorithm>



CHouse::CHouse()
{
    m_Set.empty();
    m_itCursor = m_Set.begin();
}

CHouse::CHouse(const CHouse& house)
{
    operator=(house);
}

// constructs a new house using its date of foundation (default: today)
CHouse::CHouse(const CDate& dtDateOfFoundation)
{
    m_dtDateOfFoundation = dtDateOfFoundation;
    m_Set.empty();
    m_itCursor = m_Set.begin();
}


// show attributes
string CHouse::Show() const
{
    // check invariance
```

```
    if (!ClassInvariant())
        return "";

    ostringstream strOutput;

    // some general information
    strOutput << "The house was founded on " << m_dtDateOfFoundation.Show()
              << " and consists of "
              << m_Set.size() << " rooms: " << endl;

    // stream all rooms
    TSetConstCursor itCursor;

    for (itCursor = m_Set.begin(); itCursor != m_Set.end(); itCursor++)
        strOutput << "   " << itCursor->Show() << endl;

    return strOutput.str();
}


// shows all internal attributes
string CHouse::ShowDebug() const
{
    ostringstream strOutput;

    // print any information about this class
    strOutput << "DEBUG info for 'CHouse'" << endl
         <<  "  m_dtDateOfFoundation=" << m_dtDateOfFoundation.Show() << endl
         << "   m_nCount="   << m_Set.size() << endl;

    // stream all rooms
    // this iterator needs to be const ...
    TSetConstCursor itCursor;

    strOutput <<  "  m_vecSet="   << endl;
    for (itCursor = m_Set.begin(); itCursor != m_Set.end(); itCursor++)
        strOutput << "        " << itCursor->Show() << endl;

    return strOutput.str();
}


// validate a house
bool CHouse::ClassInvariant() const
{
    // check date of foundation
    if (!m_dtDateOfFoundation.ClassInvariant())
        return false;

    // no room should be held twice in the array
    TSetConstCursor itCursor;
    itCursor = m_Set.begin();

    while (itCursor != m_Set.end())
    {
        // verify invariance of each stored room
        if (!itCursor->ClassInvariant())
            return false;

        // next room
        itCursor++;

        // look for exact copy
        TSetConstCursor itFind;
        itFind = std::find(itCursor, m_Set.end(), *itCursor);

        // copy found ?
//        if (itFind != m_Set.end())
//            return false;
    }

    return true;
}
```

```
// copy constructor
CHouse& CHouse::operator =(const CHouse &house)
{
    m_dtDateOfFoundation = house.m_dtDateOfFoundation;
    m_itCursor = house.m_itCursor;
    // container performs the whole copy internally
    m_Set      = house.m_Set;

    return *this;
}


// virtual, see operator=
bool CHouse::Copy(const CBasicClass* pClass)
{
    // cast to CHouse
    const CHouse *house = dynamic_cast<const CHouse*>(pClass);

    // invalid class (is NULL when pClass is not a CHouse)
    if (house == NULL || house == this)
        return false;

    // use non virtual reference based copy
    // return value isn't needed
    operator=(*house);

    // we're done
    return true;
}


// compare two houses
bool CHouse::operator ==(const CHouse &house) const
{
    // compare date of foundation
    if (!(m_dtDateOfFoundation == house.m_dtDateOfFoundation))
        return false;

    // find each room of the given house in our house
    TSetConstCursor itCursor;
    for (itCursor = house.m_Set.begin(); itCursor != house.m_Set.end(); itCursor++)
        // I re-implement the find method because of keeping this method const
        if (std::find(m_Set.begin(), m_Set.end(), *itCursor) == m_Set.end())
            return false;

    // all rooms were found
    return true;
}


// virtual, see operator==
bool CHouse::EqualValue(const CBasicClass* pClass) const
{
    // cast to CHouse
    const CHouse *house = dynamic_cast<const CHouse*>(pClass);

    // invalid class (is NULL when pClass is not a CHouse)
    if (house == NULL || house == this)
        return false;

    // use non virtual reference based copy
    return operator==(*house);
}


// return date of foundation
CDate CHouse::GetDateOfFoundation() const
{
    return m_dtDateOfFoundation;
}


// return number of stored rooms
int CHouse::Card() const
{
```

```
    return m_Set.size();
}


// insert a new room into the set, TRUE if successful
// will fail if room already exists
bool CHouse::Insert(const CRoom &room)
{
    // room should be valid
    if (!room.ClassInvariant())
//        throw CException("CHouse::Insert: invalid room", this);
        return false;

    // there must be some memory to grow the set
    if (m_Set.size() > m_Set.max_size())
        throw CException("CHouse::Insert: out of memory", this);

    // and the room must not be part of the list
    if (Find(room))
        return false;

    // insert at the tail of the list
    m_Set.push_back(room);

    return true;
}


// return first stored room, TRUE if successful
bool CHouse::GetFirst(CRoom &room)
{
    // set must not be empty
    if (m_Set.empty())
//        throw CException("CHouse::GetFirst: empty set", this);
        return false;

    // set cursor to first room
    m_itCursor = m_Set.begin();
    // get this room
    room = *m_itCursor;

    return true;
}


// return next room, TRUE if successful
bool CHouse::GetNext(CRoom &room)
{
    // set must not be empty, end must not be reached
    if (m_Set.empty())
//        throw CException("CHouse::GetNext: empty set", this);
        return false;

    if (m_itCursor != m_Set.end())
//        throw CException("CHouse::GetNext: reached end", this);
        return false;

    // iterate to next object
    m_itCursor++;
    // get this object
    room = *m_itCursor;

    return true;
}


// look for a room and set cursor, TRUE if successful
bool CHouse::Find(const CRoom &room)
{
    // set must not be empty
    if (m_Set.empty())
//        throw CException("CHouse::Find: empty set", this);
        return false;

    // create new iterator
```

```
    TSetCursor itCursor;

    // use STL's find
    itCursor = std::find(m_Set.begin(), m_Set.end(), room);

    // change m_itCursor if room was found
    if (itCursor != m_Set.end())
    {
        m_itCursor = itCursor;
        return true;
    }
    else
        return false;
}


// return the room the cursor points to, TRUE if successful
bool CHouse::GetCurrent(CRoom &room) const
{
    // set must not be empty
    if (m_Set.empty())
        throw CException("CHouse::GetCurrent: empty set", this);

    // return current room
    room = *m_itCursor;
    return true;
}


// erase current room, TRUE if successful
bool CHouse::Scratch()
{
    // set must not be empty
    if (m_Set.empty())
        throw CException("CHouse::Scratch: empty set", this);

    // erase room, set cursor to next room
    m_itCursor = m_Set.erase(m_itCursor);
    return true;
}


// emulates basic iterator independent of those from STL
void CHouse::ForEachDo(const TIteratorFunc iteratorProc)
{
    // save current cursor
    TSetCursor itCursor = m_itCursor;

    m_itCursor = m_Set.begin();
    while (m_itCursor != m_Set.end())
    {
        CRoom* room = &(*m_itCursor);
        iteratorProc(room);
        m_itCursor++;
    }

    // restore cursor
    m_itCursor = itCursor;
}

// helper
static ostringstream _strShow;
static void _ShowProc(CRoom* room)
{
    _strShow << room->Show();
}

// same functionality as Show, I just use ForEachDo this time
string CHouse::ShowUsingForEachDo() const
{
    CHouse* house = const_cast<CHouse*>(this);
    house->ForEachDo(_ShowProc);

    return _strShow.str();
}
```

O3-5.cpp

```
////////////////////////////////////////////////////////
// Softwarebauelemente II, Aufgabe O3.5
//
// author:          Stephan Brumme
// last changes:    June 27, 2001


#include "Date.h"
#include "Room.h"
#include "CashOffice.h"
#include "House.h"

#include <iostream>
using namespace std;


void main()
{
    CHouse myHouse(CDate(27,12,1978));

    myHouse.Insert(CCashOffice(10,20,1));
    myHouse.Insert(CCashOffice(11,21,2));

    cout<<"ForEachDo: "<<endl
        <<myHouse.ShowUsingForEachDo()<<endl;
    cout<<"STL:       "<<endl
        <<myHouse.Show()<<endl;
}
```