Dokumentation

Die Aufgabenstellung ähnelt enorm derjenigen, die in O3.5 gestellt wurde. Aus diesem Grunde gestaltet sich der neu zu schreibende Code recht einfach:

```
// helper
static ostringstream _strShow;
static void _ShowProc(const CRoom& room)
{
    _strShow << room.Show();
}


// 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;

    for_each(m_Set.begin(), m_Set.end(), _ShowProc);

    strOutput << _strShow.str();
    return strOutput.str();
}
```

Zum Vergleich zitiere ich den in O3.5 benutzten Code, man sieht die Ähnlichkeit:

```
// 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();
}
```

Durch die Benutzung von `for_each` erspare ich mir also die Ausprogrammierung von `ForEachDo`, was sich sehr positiv auf die Verständlichkeit auswirkt.

Quelltext

Es wurden nur Erweiterungen in House.h, House.cpp und C1_2.cpp vorgenommen, der Rest blieb unverändert und wurde aus den vorherigen Hausaufgaben übernommen.

House.h

```
//////////////////////////////////////////////////////
// Softwarebauelemente II, Aufgabe C1.2
//
// author:          Stephan Brumme
// last changes:    July 3, 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 <set>

// CRoom used as member variable
#include "Room.h"
// CDate used as member variable
#include "Date.h"
// template used to compare two objects
#include "MyLess.tli"


// disable compiler warning (too long names)
#pragma warning (disable: 4786)


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

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();

private:
    CDate m_dtDateOfFoundation;

    TSetOfRooms m_Set;
    TSetCursor  m_itCursor;
};

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

House.cpp

```
//////////////////////////////////////////////////////
// Softwarebauelemente II, Aufgabe C1.2
//
// author:          Stephan Brumme
// last changes:    July 3, 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();
}


// helper
static ostringstream _strShow;
static void _ShowProc(const CRoom& room)
{
    _strShow << room.Show();
}


// 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;

    for_each(m_Set.begin(), m_Set.end(), _ShowProc);

    strOutput << _strShow.str();
    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.insert(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;
}
```

O1_2.cpp

```
//////////////////////////////////////////////////////
// Softwarebauelemente II, Aufgabe C1.2
//
// author:          Stephan Brumme
// last changes:    July 3, 2001


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

#include <iostream>
using namespace std;


void main()
{
    set<CRoom> roomSet;
    set<CCashOffice> cofSet;

    CRoom room(5,10);
    CRoom room2(6,7);
    roomSet.insert(room);

    CCashOffice cof(1,2,3);
    cofSet.insert(cof);

    CHouse house;
    house.Insert(room);
    house.Insert(room2);
    cout << house.Show();
}
```