dynamic_cast

Ich bleibe meiner Strategie treu, jeweils zwei Funktionen für das Kopieren bzw. Vergleichen bereitzustellen: eine auf Basis von Zeigern, der auch Polymorphie beherrscht und eine, die intuitiver die Operatoren überlädt, was per Referenzparameter eine höhere Typsicherheit aufweist. Letzteres verzichtet bewusst auf polymorphe Fähigkeiten und benötigt darum auch kein `dynamic_cast`.

Es wurde durchgängig `Copy` bzw. `EqualValue` in allen Klassen umgestellt, als Beispiel dient `CHouse`:

```
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;
}
```

Da es sich um ein `dynamic_cast` auf einen Zeiger handelt, wird ein ungültiger Typ durch die Rückgabe von `NULL` markiert. Diesen Fall fange ich problemlos ab, etwas anders würde es bei einer Referenz aussehen. Leider tritt dieser Fall in meinem System nirgends auf, sodass ich ein etwas theoretisches Beispiel konstruiere:

```
void CClass::Dummy(const CBasicClass& Class)
{
    // cast to CHouse
    try
    {
        const CBetterClass betterclass = dynamic_cast<CBetterClass>(Class);
        betterclass.FeelGood();
    }
    // invalid class
    catch (bad_cast)
    {
        cout << "sorry, wrong class type !";
    }
}
```

CException

Es war unabdingbar, dass ich eine Klasse dem System hinzufüge, die zur Verwaltung von Ausnahmen zuständig ist. Als wesentliche Eigenschaften kann sie eine erläuternde Zeichenkette mit sich führen und besitzt ebenso einen Zeiger auf die Instanz, die für die Ausnahme zuständig war, was das Debugging erheblich vereinfacht.

Zur Wahrung der Konsistenz stammt auch diese Klasse von `CBasicClass` ab. Gerade hier bestätigt sich mein seit etwa einem halben Semester verfolgtes Konzept einer generellen Basisklasse, da jede Klasse des gesamten Systems im Ausnahmezustand mittels `ShowDebug` die Fehlerquelle gut lokalisieren kann.

Wenn man sich den Code von `CHouse` genauer ansieht, stellt man fest, dass dann doch eher sparsam mit Exceptions umgegangen wird. Ich halte diese Methodik daher für angemessen, weil nicht jeder Zustand, der als Ausnahme denkbar wäre, auch einen schwerwiegenden Fehler darstellt, der tiefe Eingriffe in den Programmlauf erfordert. Im allgemeinen reicht die Signalisierung über den Rückgabewert aus – wie in dem oben erwähnten `Copy`-Beispiel. Über die scharfe Abgrenzung „lokaler Fehler / Ausnahme“ lässt sich lange diskutieren, hier sind Fingerspitzengefühl und Erfahrung gefragt, die das zugrundeliegende Design bestimmen. Für mich steht der stabile Programmlauf an oberster Stelle. Wenn die Rückgabewerte ordentlich überprüft werden, kann eine

angemessene lokale Reaktion für eine problemlose Ausführung des Programms sorgen, was Ausnahmen unnötig macht. Letztere halte ich nur für angebracht, wenn es nicht unbedingt möglich ist, im aufrufenden Code auf den Fehler zu reagieren. Wie eng diese beiden Fälle beieinander liegen, zeigt `Insert` von `CHouse`:

```
// 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;
}
```

So kann das Einfügen eines ungültigen Raumes als Ausnahme gewertet werden, ich denke aber, dass dies nicht notwendig ist. Die nachfolgende Überprüfung, ob der Hauptspeicher ausreicht, ist von derart wesentlicher Bedeutung, dass eventuell eine Programmterminierung erforderlich ist, was sicherlich eine Ausnahme erfordert.

Quellcode

Da sich sehr wenig in der Source gegenüber Aufgabe O3.1 geändert hat, folgen hier nur `CException` und `CHouse`.

Exception.h

```
////////////////////////////////////////////////////////
// Softwarebauelemente II, Aufgabe O3.2
//
// author:          Stephan Brumme
// last changes:    May 30, 2001


#if !defined(AFX_EXCEPTION_H__183BDBA1_545F_11D5_9BB8_A7F13DB29851__INCLUDED_)
#define AFX_EXCEPTION_H__183BDBA1_545F_11D5_9BB8_A7F13DB29851__INCLUDED_

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

#include "BasicClass.h"

class CException : public CBasicClass
{
public:
    // construct new exception
    CException();
    CException(const string strError, const CBasicClass* pSource);
    CException(const CException& exception);
    virtual ~CException() {};

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

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

    // validate an exception
    virtual bool ClassInvariant() const;

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

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

private:
    string m_strError;
    const CBasicClass* m_pSource;
};

#endif // !defined(AFX_EXCEPTION_H__183BDBA1_545F_11D5_9BB8_A7F13DB29851__INCLUDED_)
```

Exception.cpp

```
////////////////////////////////////////////////////////
// Softwarebauelemente II, Aufgabe O3.2
//
// author:          Stephan Brumme
// last changes:    May 30, 2001

#include "Exception.h"
```

```
CException::CException()
{
    m_strError.empty();
    m_pSource = NULL;
}

CException::CException(const CException& exception)
{
    operator=(exception);
}

// constructs a new exception
CException::CException(const string strError, const CBasicClass* pSource)
{
    m_strError = strError;
    m_pSource  = pSource;
}


// show attributes
string CException::Show() const
{
    // check invariance
    if (!ClassInvariant())
        return "";

    ostringstream strOutput;

    strOutput << "Exception " << m_strError << " thrown." << endl
              << m_pSource->Show();

    return strOutput.str();
}


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

    // print any information about this class
    strOutput << "DEBUG info for `CException'"  << endl
              << "   m_strError=" << m_strError << endl
              << "   m_pSource=" << m_pSource   << endl;

    return strOutput.str();
}


// validate an exception
bool CException::ClassInvariant() const
{
    return (m_pSource != NULL);
}


// copy constructor
CException& CException::operator =(const CException &exception)
{
    m_strError = exception.m_strError;
    m_pSource  = exception.m_pSource;
    return *this;
}


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

    // invalid class (is NULL when pClass is not a CException)
    if (exception == NULL || exception == this)
        return false;
```

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

    // we're done
    return true;
}


// compare two exceptions
bool CException::operator ==(const CException &exception) const
{
    // all rooms were found
    return (m_strError == exception.m_strError &&
            m_pSource  == exception.m_pSource);
}


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

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

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

House.h

```
////////////////////////////////////////////////////////
// Softwarebauelemente II, Aufgabe O3.2
//
// author:          Stephan Brumme
// last changes:    May 29, 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;

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.h

```
////////////////////////////////////////////////////////
// Softwarebauelemente II, Aufgabe O3.2
//
// author:          Stephan Brumme
// last changes:    May 29, 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;
}
```

O3_2.cpp

```
////////////////////////////////////////////////////////
// Softwarebauelemente II, Aufgabe O3.2
//
// author:          Stephan Brumme
// last changes:    May 29, 2001


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

#include <iostream>
```

```
using namespace std;


void main()
{
    CDate myDate(27,12,1978);
    CDate myDate2(28,12,1978);

    CCashOffice myCashOffice(10,20,1);
    CCashOffice myCashOffice2(11,21,2);

    CHouse myHouse(myDate);
    CHouse myHouse2;

    myHouse.Insert(myCashOffice);
    myHouse.Insert(myCashOffice2);

    cout<<myHouse.Show();
    cout<<myHouse2.Show();

    myHouse2 = myHouse;

    cout<<myHouse2.Show();;
    cout<<myHouse2.ShowDebug();

    cout<<(myHouse==myHouse2)<<endl;
}
```