Dokumentation

Ich habe in allen Klassen konsequent `ClassInvariant` umgesetzt, als Beispiel dient hier eine der komplexeren Versionen, nämlich die in `CHouse`:

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

    // empty set
    if (m_Set.empty())
        return true;

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

    while (itCursor != m_Set.end())
    {
        // 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;
}
```

Man wird sicherlich leicht erkennen, dass bereits von der STL Gebrauch gemacht wird, ich halte diese Vorgehensweise für die einzig sinnvolle, um eine Stabilität meines Codes garantieren zu können. Außerdem konnte ich gleich das Template-Konzept aus der Vorlesung in einem echten Programm testen.

Ansonsten wurde die zugrunde liegende Klassenstruktur nicht verändert, sie ist eine Kopie aus Aufgabe O2.1 bzw. O2.2.

Quellcode

BasicClass.h

```
//////////////////////////////////////////////////////
// Softwarebauelemente II, Aufgabe O3.1
//
// author:          Stephan Brumme
// last changes:    May 15, 2001


#if !defined(AFX_BASICCLASS_H__C0EA0EA0_0BD2_11D5_9BB7_B22A84F06321__INCLUDED_)
#define AFX_BASICCLASS_H__C0EA0EA0_0BD2_11D5_9BB7_B22A84F06321__INCLUDED_

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


#include <string>
#include <sstream>
using namespace std;



class CBasicClass
{
public:
    CBasicClass() {};
    virtual ~CBasicClass() {};

    // show attributes
    virtual string Show() const = 0;
    // shows all internal attributes
    virtual string ShowDebug() const = 0;

    // return class name
    virtual string ClassnameOf() const = 0;


    // copy constructors
    // non-virtual
//  CBasicClass& operator = (const CBasicClass &myclass);
    // virtual
    virtual bool Copy (const CBasicClass *pClass) = 0;

    // compare two dates
    // non-virtual
//  bool operator ==(const CBasicClass &myclass) const;
    // virtual
    virtual bool EqualValue(const CBasicClass *pClass) const = 0;

    // compare the address of two objects
    bool Equal(const CBasicClass *pClass) const
        { return (this == pClass); };

    // determine whether object contains valid data
    virtual bool ClassInvariant() const = 0;
};

#endif // !defined(AFX_BASICCLASS_H__C0EA0EA0_0BD2_11D5_9BB7_B22A84F06321__INCLUDED_)
```

Date.h

```
//////////////////////////////////////////////////////////
// Softwarebauelemente II, Aufgabe O3.1
//
// author:          Stephan Brumme
// last changes:    May 15, 2001

#if !defined(AFX_DATE_H__A8EA7861_08E1_11D5_9BB7_A8652B9FDD20__INCLUDED_)
#define AFX_DATE_H__A8EA7861_08E1_11D5_9BB7_A8652B9FDD20__INCLUDED_

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


// for basic class behaviour
#include "BasicClass.h"

#include <string>
using namespace std;

//////////////////////////////////////////////////////////
// the CDate class is based upon the Gregorian calender
// it DOES work for year between ca. 1600 and <2^31
// no year validation is performed
//
// general order of parameters: dd/mm/yyyy hh:mm:ss
class CDate : public CBasicClass
{
public:
    // constructor, default value is current date and time
    CDate();
    // copy constructor
    CDate(const CDate &date);
    // set user defined date at construction time
    CDate(unsigned int nDay,    unsigned int nMonth,    unsigned int nYear,
          unsigned int nHour=0, unsigned int nMinute=0, unsigned int nSecond=0);

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

    // show attributes
    virtual string Show() const;
    // shows all internal attributes
    virtual string ShowDebug() const;

    // return today's date
    static void GetToday(unsigned int &nDay,  unsigned int &nMonth,  unsigned int &nYear,
                         unsigned int &nHour, unsigned int &nMinute, unsigned int &nSecond);

    // validate a date
    virtual bool ClassInvariant() const;
    // determine whether it is a leap year
    static bool IsLeapYear(unsigned int nYear);

    // copy constructors
    // non virtual
    CDate& operator = (const CDate &date);
    // virtual
    virtual bool Copy (const CBasicClass *pClass);

    // compare two dates
    // non virtual
    bool operator == (const CDate &date) const;
    // virtual
    virtual bool EqualValue(const CBasicClass *pClass) const;


    // set attributes, returns validility of date
    void SetDay   (unsigned int nDay);
    void SetMonth (unsigned int nMonth);
    void SetYear  (unsigned int nYear);
    void SetHour  (unsigned int nHour);
    void SetMinute(unsigned int nMinute);
```

```
    void SetSecond(unsigned int nSecond);

    // return attributes
    unsigned int GetDay   () const;
    unsigned int GetMonth () const;
    unsigned int GetYear  () const;
    unsigned int GetHour  () const;
    unsigned int GetMinute() const;
    unsigned int GetSecond() const;

    // constants for month's names
    enum { JANUARY   = 1, FEBRUARY = 2, MARCH    = 3, APRIL    = 4,
           MAY       = 5, JUNE     = 6, JULY     = 7, AUGUST   = 8,
           SEPTEMBER = 9, OCTOBER  =10, NOVEMBER =11, DECEMBER =12 };

private:
    // attributes
    unsigned int m_nDay;
    unsigned int m_nMonth;
    unsigned int m_nYear;

    unsigned int m_nHour;
    unsigned int m_nMinute;
    unsigned int m_nSecond;
};


#endif // !defined(AFX_DATE_H__A8EA7861_08E1_11D5_9BB7_A8652B9FDD20__INCLUDED_)
```

Date.cpp

```
//////////////////////////////////////////////////////////
// Softwarebauelemente II, Aufgabe O3.1
//
// author:          Stephan Brumme
// last changes:    May 15, 2001


#include "Date.h"


// we are using OS-specific date/time operations
#include <time.h>
#include <iomanip>
#include <iostream>
using namespace std;



// constructor, default value is current date and time
CDate::CDate()
{
    GetToday(m_nDay, m_nMonth, m_nYear, m_nHour, m_nMinute, m_nSecond);
}


// copy constructor
CDate::CDate(const CDate &date)
{
    operator=(date);
}


// set user defined date at construction time
CDate::CDate(unsigned int nDay, unsigned int nMonth, unsigned int nYear,
             unsigned int nHour, unsigned int nMinute, unsigned int nSecond)
{
    // store date
    m_nDay    = nDay;
    m_nMonth  = nMonth;
    m_nYear   = nYear;
```

```
    // store time
    m_nHour   = nHour;
    m_nMinute = nMinute;
    m_nSecond = nSecond;
}


// show attributes
string CDate::Show() const
{
    ostringstream strOutput;

    strOutput << setw(2) << setfill('0') << m_nDay << "."
              << setw(2) << setfill('0') << m_nMonth << "."
              << m_nYear << " - "
              << setw(2) << setfill('0') << m_nHour<< ":"
              << setw(2) << setfill('0') << m_nMinute << ":"
              << setw(2) << setfill('0') << m_nSecond;

    return strOutput.str();
}


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

    strOutput << "DEBUG info for 'CDate'" << endl
              << "      m_nDay    = " << m_nDay    << endl
              << "      m_nMonth  = " << m_nMonth  << endl
              << "      m_nYear   = " << m_nYear   << endl
              << "      m_nHour   = " << m_nHour   << endl
              << "      m_nMinute = " << m_nMinute << endl
              << "      m_nSecond = " << m_nSecond << endl;

    return strOutput.str();
}


// return current date
void CDate::GetToday(unsigned int &nDay,  unsigned int &nMonth,  unsigned int &nYear,
                     unsigned int &nHour, unsigned int &nMinute, unsigned int &nSecond)
{

    // the following code is basically taken from MSDN

    // use system functions to get the current date as UTC
    time_t secondsSince1970;
    time(&secondsSince1970);

    // convert UTC to local time zone
    struct tm *localTime;
    localTime = localtime(&secondsSince1970);

    // store retrieved date
    nDay    = localTime->tm_mday;
    nMonth  = localTime->tm_mon + 1;
    nYear   = localTime->tm_year + 1900;
    // store time
    nHour   = localTime->tm_hour;
    nMinute = localTime->tm_min;
    nSecond = localTime->tm_sec;
}


// validate a date
bool CDate::ClassInvariant() const
{
    // validate month
    if (m_nMonth < JANUARY || m_nMonth > DECEMBER)
        return false;

    // days per month, february may vary, note that array starts with 0, not JANUARY
```

```
    unsigned int nDaysPerMonth[] = {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

    // adjust days of february
    if (IsLeapYear(m_nYear))
        nDaysPerMonth[FEBRUARY]++;

    // validate day
    if (m_nDay < 1 || m_nDay > nDaysPerMonth[m_nMonth])
        return false;
    // day and month are valid

    // now check the time
    if (m_nHour < 0 || m_nHour > 23)
        return false;
    if (m_nMinute < 0 || m_nMinute > 59)
        return false;
    if (m_nSecond < 0 || m_nSecond > 59)
        return false;

    // instance must be valid now
    return true;
}


// determine whether it is a leap year
bool CDate::IsLeapYear(unsigned int nYear)
{
    // used to speed up code, may be deleted
    if (nYear % 4   != 0)
        return false;

    // algorithm taken from MSDN, just converted from VBA to C++
    if (nYear % 400 == 0)
        return true;
    if (nYear % 100 == 0)
        return false;
    if (nYear % 4   == 0)
        return true;

    // this line won't be executed because of optimization (see above)
    return false;
}


// copy constructor
CDate& CDate::operator =(const CDate &date)
{
    // date
    m_nDay    = date.m_nDay;
    m_nMonth  = date.m_nMonth;
    m_nYear   = date.m_nYear;
    // time
    m_nHour   = date.m_nHour;
    m_nMinute = date.m_nMinute;
    m_nSecond = date.m_nSecond;

    return *this;
}


// virtual, see operator=
bool CDate::Copy(const CBasicClass *pClass)
{
    // cast to CDate
    CBasicClass *basic;
    CDate *date;
    basic = const_cast<CBasicClass*>(pClass);
    date = dynamic_cast<CDate*>(basic);

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

    // use non virtual reference based copy
    // return value isn't needed
```

```
    operator=(*date);

    // we're done
    return true;
}


// compare two dates
bool CDate::operator ==(const CDate &date) const
{
    // compare date and time attributes
    return (m_nDay    == date.m_nDay    &&
            m_nMonth  == date.m_nMonth  &&
            m_nYear   == date.m_nYear   &&
            m_nHour   == date.m_nHour   &&
            m_nMinute == date.m_nMinute &&
            m_nSecond == date.m_nSecond);
}


// virtual, see operator==
bool CDate::EqualValue(const CBasicClass *pClass) const
{
    // cast to CDate
    CBasicClass *basic;
    CDate *date;
    basic = const_cast<CBasicClass*>(pClass);
    date = dynamic_cast<CDate*>(basic);

    // invalid class (is NULL when pClass is not a CDate)
    if (date == NULL)
        return false;

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


// set attributes, returns validility of date
void CDate::SetDay(unsigned int nDay)
{
    m_nDay = nDay;
}
void CDate::SetMonth(unsigned int nMonth)
{
    m_nMonth = nMonth;
}
void CDate::SetYear(unsigned int nYear)
{
    m_nYear = nYear;
}
void CDate::SetHour(unsigned int nHour)
{
    m_nHour = nHour;
}
void CDate::SetMinute(unsigned int nMinute)
{
    m_nMinute = nMinute;
}
void CDate::SetSecond(unsigned int nSecond)
{
    m_nSecond = nSecond;
}

// return attributes
unsigned int CDate::GetDay() const
{
    return m_nDay;
}
unsigned int CDate::GetMonth() const
{
    return m_nMonth;
}
unsigned int CDate::GetYear() const
{
```

```
    return m_nYear;
}
unsigned int CDate::GetHour() const
{
    return m_nHour;
}
unsigned int CDate::GetMinute() const
{
    return m_nMinute;
}
unsigned int CDate::GetSecond() const
{
    return m_nSecond;
}
```

Room.h

```
//////////////////////////////////////////////////////
// Softwarebauelemente II, Aufgabe O3.1
//
// author:          Stephan Brumme
// last changes:    May 15, 2001

#if !defined(AFX_ROOM_H__34138CE0_E97C_11D4_9BB7_8BA1BD2C3421__INCLUDED_)
#define AFX_ROOM_H__34138CE0_E97C_11D4_9BB7_8BA1BD2C3421__INCLUDED_

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


#include "BasicClass.h"


// declare class CRoom
class CRoom : public CBasicClass
{
public:
    // constructors
    CRoom();
    CRoom(const CRoom& room);
    CRoom(int nNumberOfRoom, int nArea);

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

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

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

    // copy constructors
    // non virtual
    CRoom&  operator = (const CRoom &room);
    // virtual
    virtual bool Copy(const CBasicClass* pClass);

    // compare two dates
    // non virtual
    bool operator == (const CRoom &room) const;
    // virtual
    virtual bool EqualValue(const CBasicClass* pClass) const;

    // access m_nNumberOfRoom
    int GetNumberOfRoom() const;
    void SetNumberOfRoom(const int nNumberOfRoom);

    // retrieve covered area
    int GetArea() const;

    enum { MAXNUMBER = 100, MAXAREA = 100 };

private:
    // hide the member variables
    int m_nNumberOfRoom;
    int m_nArea;
};

#endif // !defined(AFX_ROOM_H__34138CE0_E97C_11D4_9BB7_8BA1BD2C3421__INCLUDED_)
```

Room.cpp

```
//////////////////////////////////////////////////////
```

```
// Softwarebauelemente II, Aufgabe O3.1
//
// author:          Stephan Brumme
// last changes:    May 15, 2001

#include "Room.h"
#include <iostream>


// default constructor
CRoom::CRoom()
{
    m_nArea = 0;
    m_nNumberOfRoom = 0;
}

// copy constructor
CRoom::CRoom(const CRoom &room)
{
    operator=(room);
}

CRoom::CRoom(int nNumberOfRoom, int nArea)
{
    m_nArea = nArea;
    m_nNumberOfRoom = nNumberOfRoom;
}


// show attributes
string CRoom::Show() const
{
    ostringstream strOutput;

    strOutput << "Room no. " << m_nNumberOfRoom
              << " covers an area of " << m_nArea << " square feet." << endl;

    return strOutput.str();
}


// display the attributes
// only for internal purposes !
string CRoom::ShowDebug() const
{
    ostringstream strOutput;

    strOutput << "DEBUG info for `CRoom`" << endl
              << "      m_nArea         = " << m_nArea
              << "      m_nNumberOfRoom = " << m_nNumberOfRoom << endl;

    return strOutput.str();
}


// verify invariance
bool CRoom::ClassInvariant() const
{
    // only positive numbers allowed
    return (m_nArea         > 0 && m_nArea         <= MAXAREA &&
            m_nNumberOfRoom > 0 && m_nNumberOfRoom <= MAXNUMBER);
}


// copy one room to another one
// prevents user from copying an object to itself
CRoom& CRoom::operator=(const CRoom &room)
{
    // copy all variables
    m_nArea         = room.m_nArea;
    m_nNumberOfRoom = room.m_nNumberOfRoom;

    return *this;
}
```

```
// virtual, see operator=
bool CRoom::Copy(const CBasicClass* pClass)
{
    // cast to CRoom
    CBasicClass *basic;
    CRoom *room;
    basic = const_cast<CBasicClass*>(pClass);
    room = dynamic_cast<CRoom*>(basic);

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

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

    // we're done
    return true;
}


// compare the room with another one
bool CRoom::operator==(const CRoom &room) const
{
    return ((room.m_nArea         == m_nArea) &&
            (room.m_nNumberOfRoom == m_nNumberOfRoom));
}


// virtual, see operator==
bool CRoom::EqualValue(const CBasicClass* pClass) const
{
    // cast to CRoom
    CBasicClass *basic;
    CRoom *room;
    basic = const_cast<CBasicClass*>(pClass);
    room = dynamic_cast<CRoom*>(basic);

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

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


// retrieve the private value of m_nNumberOfRoom
int CRoom::GetNumberOfRoom() const
{
    return m_nNumberOfRoom;
}


// change private m_nNumberOfRoom
void CRoom::SetNumberOfRoom(const int nNumberOfRoom)
{
    m_nNumberOfRoom = nNumberOfRoom;
}


// retrieve the private value of m_nArea
int CRoom::GetArea() const
{
    return m_nArea;
}
```

CashOffice.h

```
//////////////////////////////////////////////////////////
// Softwarebauelemente II, Aufgabe O3.1
//
// author:          Stephan Brumme
// last changes:    May 15, 2001

#if !defined(AFX_CASHOFFICE_H__26E07000_EA3B_11D4_9BB7_AFEE07846A21__INCLUDED_)
#define AFX_CASHOFFICE_H__26E07000_EA3B_11D4_9BB7_AFEE07846A21__INCLUDED_

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


// we derive from CRoom
#include "Room.h"


// derive publicly
class CCashOffice : public CRoom
{
public:
    // constructors
    CCashOffice();
    CCashOffice(const CCashOffice& cashOffice);
    CCashOffice(int nNumberOfRoom, int nArea, int nNumberOfCounter);

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

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

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

    // copy one cashoffice to another one
    virtual CCashOffice& operator=(const CCashOffice& cashOffice);
    virtual bool Copy(const CBasicClass* pClass);

    // compare two rooms
    virtual bool operator==(const CCashOffice& cashOffice) const;
    virtual bool EqualValue(const CBasicClass* pClass) const;

    // access m_nNumberOfCounter
    int  GetNumberOfCounter() const;
    void SetNumberOfCounter(const int nNumberOfCounter);

    // access m_nNumberOfRoom of CRoom
    int  GetNumberOfCashOffice() const;
    void SetNumberOfCashOffice(const int nNumberOfCashOffice);

    // deliver m_nArea of CRoom
    int GetAreaOfCashOffice() const;

    enum { MAXCOUNTER = 100 };
private:
    int m_nNumberOfCounter;
};

#endif // !defined(AFX_CASHOFFICE_H__26E07000_EA3B_11D4_9BB7_AFEE07846A21__INCLUDED_)
```

CashOffice.cpp

```
//////////////////////////////////////////////////////////
// Softwarebauelemente II, Aufgabe O3.1
//
```

```
// author:          Stephan Brumme
// last changes:    May 15, 2001

#include "CashOffice.h"


// default constructor
CCashOffice::CCashOffice() : CRoom()
{
    m_nNumberOfCounter = 0;
}

// copy constructor
CCashOffice::CCashOffice(const CCashOffice& cashOffice)
{
    operator=(cashOffice);
}

CCashOffice::CCashOffice(int nNumberOfRoom, int nArea, int nNumberOfCounter)
    : CRoom(nNumberOfRoom, nArea)
{
    m_nNumberOfCounter = nNumberOfCounter;
}


// display the attributes
string CCashOffice::Show() const
{
    ostringstream strOutput;

    strOutput << CRoom::Show() << " Counter no. "
              << m_nNumberOfCounter << "." << endl;

    return strOutput.str();
}


// display the attributes
// only for internal purposes !
string CCashOffice::ShowDebug() const
{
    ostringstream strOutput;

    strOutput << CRoom::ShowDebug()
              << "DEBUG info for `CCashOffice'" << endl
              << "      m_nNumberOfCounter = " << m_nNumberOfCounter << endl;

    return strOutput.str();
}


// verify invariance
bool CCashOffice::ClassInvariant() const
{
    // only positive numbers allowed
    return (CRoom::ClassInvariant() &&
            m_nNumberOfCounter > 0  && m_nNumberOfCounter <= MAXCOUNTER);
}


// copy one CashOffice to another one
CCashOffice& CCashOffice::operator=(const CCashOffice& cashOffice)
{
    // copy all variables
    // first copy the attributes of CRoom
    CRoom::operator=(cashOffice);
    // and now our new ones of CCashOffice
    m_nNumberOfCounter = cashOffice.m_nNumberOfCounter;

    return *this;
}


// virtual, see operator=
bool CCashOffice::Copy(const CBasicClass* pClass)
```

```
{
    // cast to CCashOffice
    CBasicClass *basic;
    CCashOffice *cashOffice;
    basic = const_cast<CBasicClass*>(pClass);
    cashOffice = dynamic_cast<CCashOffice*>(basic);

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

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

    // we're done
    return true;
}


// compare the CashOffice with another one
bool CCashOffice::operator==(const CCashOffice& cashOffice) const
{
    // use the CRoom compare function
    return (CRoom::operator==(cashOffice) &&
            m_nNumberOfCounter == cashOffice.m_nNumberOfCounter);
}


// compare the CashOffice with another one
// routes call down to "==" operator
bool CCashOffice::EqualValue(const CBasicClass* pClass) const
{
    // cast to CCashOffice
    CBasicClass *basic;
    CCashOffice *cashOffice;
    basic = const_cast<CBasicClass*>(pClass);
    cashOffice = dynamic_cast<CCashOffice*>(basic);

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

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


// retrieve the private value of m_nNumberOfCounter
int CCashOffice::GetNumberOfCounter() const
{
    return m_nNumberOfCounter;
}


// change private m_nNumberOfCounter
void CCashOffice::SetNumberOfCounter(const int nNumberOfCounter)
{
    m_nNumberOfCounter = nNumberOfCounter;
}


// retrieve the private value of CRoom::m_nNumberOfCashOffice
int CCashOffice::GetNumberOfCashOffice() const
{
    return CRoom::GetNumberOfRoom();
}


// change private CRoom::m_nNumberOfCashOffice
void CCashOffice::SetNumberOfCashOffice(const int nNumberOfCashOffice)
{
    CRoom::SetNumberOfRoom(nNumberOfCashOffice);
}
```

```
// deliver CRoom::m_nArea
int CCashOffice::GetAreaOfCashOffice() const
{
    return CRoom::GetArea();
}
```

House.h

```
//////////////////////////////////////////////////////
// Softwarebauelemente II, Aufgabe O3.1
//
// author:          Stephan Brumme
// last changes:    May 15, 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.cpp

```
//////////////////////////////////////////////////////////
// Softwarebauelemente II, Aufgabe O3.1
//
// author:          Stephan Brumme
// last changes:    May 15, 2001


#include "House.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
{
    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;

    // empty set
    if (m_Set.empty())
        return true;

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

    while (itCursor != m_Set.end())
    {
        // 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
    CBasicClass *basic;
    CHouse *house;
    basic = const_cast<CBasicClass*>(pClass);
    house = dynamic_cast<CHouse*>(basic);

    // 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
    CBasicClass *basic;
    CHouse *house;
    basic = const_cast<CBasicClass*>(pClass);
    house = dynamic_cast<CHouse*>(basic);

    // 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)
{
    // there must be some memory to grow the set
    // and the room must not be part of the list
    if (m_Set.size() < m_Set.max_size() &&
        !Find(room))
    {
        // insert at the tail of the list
        m_Set.push_back(room);
        return true;
    }
    else
        return false;
```

```
}


// return first stored room, TRUE if successful
bool CHouse::GetFirst(CRoom &room)
{
    // set must not be empty
    if (m_Set.empty())
        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() && m_itCursor != m_Set.end())
        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())
        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())
        return false;

    // 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())
        return false;

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

O3_1.cpp

```
////////////////////////////////////////////////////////
// Softwarebauelemente II, Aufgabe O3.1
//
// author:          Stephan Brumme
// last changes:    May 28, 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;
}
```