Mit dem Beginn des neuen Semesters ändere ich auch ein paar Kleinigkeiten bezüglich des Programmierstils:

- Die Datei PrimitiveTypes.h wird *nicht* mehr eingebunden. Ihr Sinn bestand lediglich darin, die Namen der CEDL-Beschreibung auf die entsprechenden C++-Basistypen abzubilden. Um bei eventueller Teamarbeit mit den ungewohnten Namen nicht in Konflikt zu kommen, benutze ich die CEDL-Typen ab sofort nicht mehr. Abschließend noch eine kurze Gegenüberstellung:

CEDL	C++
`Ordinal`	`int`
`Boolean`	`bool`
`Real`	`float` oder `double`
`String`	`char*` oder `string`

- Jede Klasse wird abgeleitet von `CBasicClass` und verfügt daher über mindestens 4 Methoden:
  (1) `ClassnameOf`:
  liefert den Namen der Klasse als Text per `char*` zurück, z.B. „CBasicClass“
  (2) `Show`:
  zeigt die Attribute in Kurzform an, gedacht zur normalen Bildschirmausgabe komplexer Datenstrukturen
  (3) `ShowDebug`:
  zeigt alle Attribute ausführlich an, soll nur während der Entwicklung benutzt werden
  (4) `OutStream`:
  eigentlicher Kern von `Show` und `ShowDebug`, abstrahiert die Ausgabe, so dass beliebige Streams möglich sind
  Wenn möglich werden noch die Operatoren <<, = und == überladen.

- Die meisten Klassen enthalten in der CEDL-Beschreibung je eine Kopier- und eine Vergleichsmethode. Um vernünftige Klassenhierarchien aufbauen zu können, sollten sie `virtual` definiert werden. Die damit unabdingbar verknüpfte Signaturgleichheit bereitet mir etwas Sorgen, da sie im Endeffekt sehr fehlerträchtige Pointer-Operationen nach sich zieht. Ich habe mich entschlossen, dieses Risiko bei `Copy` und `EqualValue` einzugehen. Jedoch bauen die überladenen Operatoren = und == auf Referenzparametern auf. In den meisten Fällen sollte man sie daher bevorzugen, nur wenn zwei Klassen unbekannten, aber gleichen Typs miteinander verknüpft werden, so ist `Copy` und `EqualValue` der Vorzug zu geben.
  Für die Klassenschnittstellen heißt das, dass `Copy` und `EqualValue` nur jeweils einzeln verfügbar sind, die Operatoren = und == aber eventuell mehrfach, d.h. für jede Klasse im Ableitungsbaum - somit ist problemlos downcasting durchführbar.
  Allgemein versuche ich, möglichst viele Parameter per Referenz statt als Zeiger zu übergeben, dabei wird ggf. `const` eingesetzt. Ebenso arbeite ich lieber mit Variablen auf dem Stack als auf dem Heap.

- In meinen Augen erscheint es sinnvoll, Methoden wenn möglich als `static` zu deklarieren. Dies trifft u.a. immer auf `ClassnameOf` zu. Eigene Destruktoren deklariere ich nur, wenn sie explizit notwendig sind, ansonsten belasse ich es beim Default-Destruktor.
  Sehr kurze Methoden sind als `inline` deklariert, um die Geschwindigkeit zu erhöhen.

- Ich verwende an sinnvollen Stellen Klassen / Templates der STL oder der MFC, wie z.B. Aufzählungsklassen oder Stringvorlagen, da sie eine hohe Typsicherheit aufweisen und sehr stabil arbeiten. Außerdem, was am wichtigsten ist, ersparen sie mir viel Arbeit und erhöhen die Lesbarkeit des Quellcodes für andere Entwickler.

- Der Quellcode ist mittels Syntax-Highlighting besser überschaubar. Da es auch immer mehr Zeilen werden, kommentiere ich die Klassen getrennt und nicht mehr als ein Block komplett vorneweg.

Einige Klassen wurden schon mit der Semesterarbeit im Sinn entwickelt und gehen daher eventuell über die Hausaufgabe hinaus. Insbesondere benutze ich die Container-Klassen der STL mit ihren interessanten Iteratoren und den darauf aufbauenden Standard-Algorithmen, wie z.B. die Suche.

Im folgenden sind Aufgabe O2.1 und O2.2 bearbeitet worden, der Quellcode umfasst die Lösungen für beide.

Aufgabe O2.1

Direkte Lösung zur Aufgabenstellung

Da `Show` extrem kurz ausfällt und lediglich die Basisimplementation in `CBasicClass` ist, führe ich noch die eigentliche ausführende Methode `OutStream` auf:

```
// show attributes
void CBasicClass::Show() const
{
    // print any information about this class
    OutStream(cout, false);
}


// used to perform proper display in derived classes
ostream& CHouse::OutStream(ostream& mystream, bool bDebug) const
{
    if (!bDebug)
    {
        // some general information
        mystream << "The house was founded on " << m_dtDateOfFoundation
                 << " and consists of "
                 << m_Set.size() << " rooms"  << endl;
    }
    else
    {
        // print any information about this class
        mystream << "DEBUG info for `" << ClassnameOf() << "'" << endl
                 << "        m_dtDateOfFoundation=" << m_dtDateOfFoundation
                 << "  rooms="   << m_Set.size() << endl;
    }

    // stream all rooms
    if (m_Set.size() > 0)
    {
        // this iterator needs to be const ...
        TSetConstCursor itCursor;
        for (itCursor = m_Set.begin(); itCursor != m_Set.end(); itCursor++)
            mystream << "        " << *itCursor << endl;
    }

    return mystream;
}
```

Eine Alternative zu `Show` ist der pipe-Operator `<<`:

```
ostream& operator<<(ostream& mystream, const CHouse& house)
{
    return house.OutStream(mystream);
}
```

`mystream` in `OutStream` kann beliebig umgelenkt werden, hier ist z.B. eine Bildschirmausgabe im Normalbetrieb und eine Dateiausgabe bei Fehlerprotokolle denkbar. Da `<<` für alle Klassen implementiert wurde, kann ebenso das Datum sehr einfach gestreamt werden:

```
        cout << "m_dtDateOfFoundation=" << m_dtDateOfFoundation
```

Die abschließende Ausgabe aller gespeicherte Räume bedient sich des gleichen Operators (diesmal natürlich für `CRoom`) und iteriert über den gesamten Container, wobei meine Wahl auf `list` fiel, da hier effizient Einfüge- und Löschoperationen möglich sind.

Aufgabe O2.2

Direkte Lösung zur Aufgabenstellung

Der Klassenname wird bereits in der jeweiligen Deklarationsdatei festgelegt, die entsprechenden Zeilen lauten:

```
        static char* ClassnameOf() { return "CDate"; }
        static char* ClassnameOf() { return "CRoom"; }
        static char* ClassnameOf() { return "CHouse"; }
```

Ich habe bewusst darauf verzichtet, einen Instanzzähler zu implementieren.

Als Kopierkonstruktor ziehe ich stellvertretend nur `CHouse` heran.

```
// 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 CHouse *house)
{
    // invalid class
    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;
}
```

Wie ich bereits erwähnte, sind jeweils `operator=` und `Copy` implementiert, wobei letzteres nur ein paar Zeiger- und Typüberprüfungen vornimmt und dann ersteres für den eigentlich Kopiervorgang aufruft.

CBasicClass

Die Klasse bildet die Grundlage für alle Klassen, da sie wesentlich Methoden für korrektes Streaming und Debugging bereitstellt.
So stehen automatisch für jede Klasse `Show` und `ShowDebug` bereit. Sie bedienen sich `OutStream`, was hier als rein virtuelle Methode definiert wurde, d.h. sie *muss* von einer abgeleiteten Klasse überschrieben werden.
Ebenso kann über `ClassnameOf` der Klassenname erfragt werden, es ist unbedingt empfehlenswert, diese Methode jedes Mal zu überschreiben.
Wie man in der Deklaration der Klasse sieht, ist relativ viel auskommentiert. Hier fiel mir leider keine bessere Lösung ein, da ich kein rein virtuellen Funktionsrümpfe vorgeben kann, weil die Parametertypen im voraus unbekannt sind. Jedoch sollte jede Klasse auch diese Methoden implementieren.
Ich öffne den sehr oft verwendeten Namensraum `std` teilweise, indem ich `cout`, `endl` und `ostream` global zur Verfügung stelle.

BasicClass.h:

```
////////////////////////////////////////////////////////
// Softwarebauelemente II, Aufgabe O2.1
//
// author:          Stephan Brumme
// last changes:    February 26, 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 <iostream>
using std::ostream;
using std::cout;
using std::endl;

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

    // show attributes
    virtual void Show() const;
    // shows all internal attributes
    virtual void ShowDebug() const;
    // display the attributes
//  friend ostream& operator<<(ostream& mystream, const CBasicClass& myclass);

    // return class name
    static char* ClassnameOf() { return "ERROR: CBasicClass::ClassnameOf not overwritten !!!";
}

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

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

protected:
    // used to perform proper display in derived classes
    virtual ostream& OutStream(ostream& mystream, bool bDebug=false) const = 0;
};

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

BasicClass.cpp:

```
//////////////////////////////////////////////////////////
// Softwarebauelemente II, Aufgabe O2.1
//
// author:          Stephan Brumme
// last changes:    February 26, 2001

#include "BasicClass.h"


CBasicClass::CBasicClass()
{
}

CBasicClass::~CBasicClass()
{
}


// show attributes
void CBasicClass::Show() const
{
    // print any information about this class
    OutStream(cout, false);
}


// shows all internal attributes
void CBasicClass::ShowDebug() const
{
    // first parameter is display stream
    OutStream(cout, true);
}
```

CDate

Alle Variablen sind nur gekapselt implementiert, d.h. für Änderungen der Datums sind `SetDay`/`GetDay`, `SetMonth`/`GetMonth` und `SetYear`/`GetYear` zuständig. Jedes Datum kann auf seine Gültigkeit überprüft werden (`IsValid`), wobei auch Schaltjahre korrekt berücksichtigt werden (`IsLeapYear`). Die Monatszahlen sind als Konstanten verfügbar.

Date.h:

```
////////////////////////////////////////////////////////
// Softwarebauelemente II, Aufgabe O2.1
//
// author:          Stephan Brumme
// last changes:    February 26, 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"


////////////////////////////////////////////////////////
// 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

class CDate : public CBasicClass
{
public:
    // constructor, default value is 0/0/00 (invalid date !)
    CDate(int nDay=0, int nMonth=0, int nYear=0);

    // show attributes
    friend ostream& operator<<(ostream &mystream, const CDate& date);
    // return class name
    static char* ClassnameOf() { return "CDate"; }

    // return current date
    static CDate GetToday();

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

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

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


    // set attributes, returns validility of date
    bool SetDay  (int nDay);
    bool SetMonth(int nMonth);
    bool SetYear (int nYear);
```

```
    // return attributes
    int  GetDay  () const;
    int  GetMonth() const;
    int  GetYear () const;

    // constants for month 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 };

protected:
    // used to perform proper display in derived classes
    virtual ostream& OutStream(ostream& mystream, bool bDebug=false) const;

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


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

Date.cpp:

```
//////////////////////////////////////////////////////
// Softwarebauelemente II, Aufgabe O2.1
//
// author:          Stephan Brumme
// last changes:    February 26, 2001


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

// we are using OS-specific date/time operations
#include <time.h>


// constructor, default value is 0/0/00 (invalid date !)
CDate::CDate(int nDay, int nMonth, int nYear)
{
    // store date
    m_nDay   = nDay;
    m_nMonth = nMonth;
    m_nYear  = nYear;
}


// attention ! this method is NOT part of CDate
ostream& operator<<(ostream &mystream, const CDate& date)
{
    return date.OutStream(mystream);
}


// used to perform proper display in derived classes
ostream& CDate::OutStream(ostream& mystream, bool bDebug) const
{
    if (!bDebug)
    {
        mystream << m_nDay << "." << m_nMonth << "." << m_nYear;
    }
    else
    {
        mystream << "DEBUG info for ‘" << ClassnameOf() << "’" << endl
            << "   m_nDay="   << m_nDay
            <<  "  m_nMonth=" << m_nMonth
            <<  "  m_nYear="  << m_nYear;
    }
    return mystream;
}
```

```
// return current date
CDate CDate::GetToday()
{
    CDate dtReturn;

    // the following code is 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
    dtReturn.m_nDay   = localTime->tm_mday;
    dtReturn.m_nMonth = localTime->tm_mon + 1;
    dtReturn.m_nYear  = localTime->tm_year + 1900;

    return dtReturn;
}


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

    // days per month, february ma(r)y vary !!!
    int nDaysPerMonth[] = {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
    return true;
}


// determine whether it is a leap year
bool CDate::IsLeapYear(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)
{
    m_nDay   = date.m_nDay;
    m_nMonth = date.m_nMonth;
    m_nYear  = date.m_nYear;
    // m_bDebug is not copied !

    return *this;
}


// virtual, see operator=
bool CDate::Copy(const CDate *date)
{
    // invalid class
    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
{
    // m_bDebug is not neccessary
    return (m_nDay   == date.m_nDay   &&
            m_nMonth == date.m_nMonth &&
            m_nYear  == date.m_nYear);
}


// virtual, see operator==
bool CDate::EqualValue(CDate *date) const
{
    // invalid class
    if (date == NULL)
        return false;

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


// set attributes, returns validility of date
inline bool CDate::SetDay(int nDay)
{
    m_nDay = nDay;
    return IsValid();
}
inline bool CDate::SetMonth(int nMonth)
{
    m_nMonth = nMonth;
    return IsValid();
}
inline bool CDate::SetYear(int nYear)
{
    m_nYear = nYear;
    return IsValid();
}

// return attributes
inline int CDate::GetDay() const
{
    return m_nDay;
}
```

```
inline int CDate::GetMonth() const
{
    return m_nMonth;
}
inline int CDate::GetYear() const
{
    return m_nYear;
}
```

CRoom

`CRoom` wurde aus der Aufgabe O1.1 übernommen und nur stilistisch leicht an `CBasicClass`/`CDate`/`CHouse` angepasst.

Room.h:

```
//////////////////////////////////////////////////////
// Softwarebauelemente II, Aufgabe O2.1
//
// author:          Stephan Brumme
// last changes:    February 26, 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


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


// declare class CRoom
class CRoom  : public CBasicClass
{
public:
    // constructor (former Init)
    CRoom(int nNumberOfRoom, int nArea);

    // return class name
    static char* ClassnameOf() { return "CRoom"; }

    // display the attributes
    friend ostream& operator<<(ostream &mystream, const CRoom& room);

    // compare two rooms
    bool operator==(const CRoom &room) const;
    virtual bool EqualValue(const CRoom* room) const;

    // copy one room to another one
    CRoom&  operator=(const CRoom &room);
    virtual bool Copy(const CRoom* room);

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

    // retrieve covered area
    int GetArea() const;

protected:
    // used to perform proper display in derived classes
    virtual ostream& OutStream(ostream& mystream, bool bDebug=false) const;

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 O2.1
//
// author:          Stephan Brumme
// last changes:    February 26, 2001

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


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


// attention ! this method is NOT part of CRoom
ostream& operator<<(ostream &mystream, const CRoom& room)
{
    return room.OutStream(mystream);
}


// used to perform proper display in derived classes
ostream& CRoom::OutStream(ostream& mystream, bool bDebug) const
{
    if (!bDebug)
    {
        mystream << "Room no. " << m_nNumberOfRoom
                 << " covers an area of " << m_nArea << " square feet.";
    }
    else
    {
        mystream << "DEBUG info for `"    << ClassnameOf()  << "'" << endl
                 << "   m_nArea="         << m_nArea
                 <<  "  m_nNumberOfRoom=" << m_nNumberOfRoom << endl;
    }

    return mystream;
}


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


// compare the room with another one
// routes call down to "==" operator
bool CRoom::EqualValue(const CRoom* room) const
{
    // disallow NULL pointer
    if (room == NULL)
        return false;

    return (operator==(*room));
}


// 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;
}
```

```
// copy one room to another one
// prevents user from copying an object to itself
// calls "=" operator internally
bool CRoom::Copy(const CRoom* room)
{
    // disallow NULL pointer
    if (room == NULL)
        return false;

    // cannot copy an object to itself
    if (this == room)
        return false;

    // use "=" operator
    operator=(*room);

    return true;
}


// 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;
}
```

CHouse

`CHouse` bereitete mir relativ viel Arbeit, da ich unbedingt die vorgefertigten STL-Container verwenden wollte. Diese wiederum kollidierten etwas mit den Streams, was ich aber in Griff bekam und nur an meinem ungaren Design lag.
Grundlage der Datenhaltung bildet eine `list`, die als

```
typedef std::list<CRoom> TSetOfRooms;
```

definiert ist. Die dazu notwendigen Iteratoren (variabel und konstant) lauten:

```
typedef TSetOfRooms::iterator       TSetCursor;
typedef TSetOfRooms::const_iterator TSetConstCursor;
```

Aufgrund der Schnittstelle von `list` ist es auch nicht mehr notwendig für `CHouse` ein Attribut `Count` mitzuführen, wie es noch in der CEDL-Beschreibung angegeben war, da diese Information über `list.size()` abrufbar ist. Die Operationen zum Einfügen und Löschen reduzieren sich daher auf einen Aufruf der jeweiligen Methode von `list`. Lediglich eine Abfrage muss vorhanden sein, um Zugriffe ein leeres Set zu vermeiden (d.h. es sind keine Räume für das Haus gespeichert).

Sehr elegant funktioniert das Streaming, so dass erneut ein `cout << myHouse;` machbar ist. Genauso einfach kann ich mich der gleichartig aufgebauten Methoden von `CRoom` und `CDate` bedienen. Ich möchte noch darauf hinweisen, dass `operator<<` nur ein `friend` der jeweiligen Klasse ist, der als nicht-objektorientiere Funktion implementiert ist und nicht einen festen Bestandteil in Form einer Methode darstellt.

House.h:

```
//////////////////////////////////////////////////////
// Softwarebauelemente II, Aufgabe O2.1
//
// author:          Stephan Brumme
// last changes:    February 26, 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"
// for basic class behaviour
#include "BasicClass.h"



class CHouse : public CBasicClass
{
    // define new types for using the set
    typedef std::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(const CDate& dtDateOfFoundation = CDate::GetToday());

    // return class name
    static char* ClassnameOf() { return "CHouse"; }

    // display the attributes
    friend ostream& operator<<(ostream& mystream, const CHouse& house);
```

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

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

    // 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();

protected:
    // used to perform proper display in derived classes
    virtual ostream& OutStream(ostream& mystream, bool bDebug=false) 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 O2.1
//
// author:          Stephan Brumme
// last changes:    February 26, 2001


#include "House.h"

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


// 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();
}


ostream& operator<<(ostream& mystream, const CHouse& house)
{
    return house.OutStream(mystream);
}
```

```
// used to perform proper display in derived classes
ostream& CHouse::OutStream(ostream& mystream, bool bDebug) const
{
    if (!bDebug)
    {
        // some general information
        mystream << "The house was founded on " << m_dtDateOfFoundation
                 << " and consists of "
                 << m_Set.size() << " rooms"  << endl;
    }
    else
    {
        // print any information about this class
        mystream << "DEBUG info for `" << ClassnameOf() << "`" << endl
                 << "        m_dtDateOfFoundation=" << m_dtDateOfFoundation
                 << "  rooms="   << m_Set.size() << endl;
    }

    // stream all rooms
    if (m_Set.size() > 0)
    {
        // this iterator needs to be const ...
        TSetConstCursor itCursor;
        for (itCursor = m_Set.begin(); itCursor != m_Set.end(); itCursor++)
            mystream << "        " << *itCursor << endl;
    }

    return mystream;
}


// 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 CHouse *house)
{
    // invalid class
    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 CHouse *house) const
{
    // invalid class
    if (house == NULL)
        return false;

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


// virtual, just compares the pointers
bool CHouse::Equal(const CHouse *house) const
{
    return (house == this);
}


// 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;
}
```

Die `main`-Routine dient mehr für meine eigenen Debugging-Zwecke und enthält kaum verwertbaren Code. Trotzdem führe ich sie hier kurz auf:

O2_1.cpp:

```
//////////////////////////////////////////////////////////
// Softwarebauelemente II, Aufgabe O2.1
//
// author:          Stephan Brumme
// last changes:    February 26, 2001


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


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

    CRoom myRoom(10,20);
    CRoom myRoom2(11,21);

    CHouse myHouse(myDate);
    CHouse myHouse2;

    myHouse.Insert(myRoom);
    myHouse.Insert(myRoom2);

    cout<<myHouse;
    cout<<myHouse2;

    myHouse2 = myHouse;

    myHouse2.ShowDebug();

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