Dokumentation

Die Klassen `CRoom` und `CCashOffice` müssen um den <-Operator erweitert werden. Ich entschloss mich, nicht alle Attribute in diesen Vergleich einzubeziehen, sondern lediglich die Raumnummern zu benutzen. Der daraus entstehende Code ist dementsprechend kurz:

```
bool CRoom::operator<(const CRoom &room) const
{
    return m_nNumberOfRoom < room.m_nNumberOfRoom;
}
```

und

```
bool CCashOffice::operator<(const CCashOffice &cashOffice) const
{
    return m_nNumberOfCounter < cashOffice.m_nNumberOfCounter;
}
```

Um daraus einen Container zu konstruieren, reicht vollkommen aus:

```
    set<CRoom> roomSet;
    set<CCashOffice> cofSet;
```

Konsequenterweise setzte ich auch `CHouse` als `set` um, dabei kam mir enorm zugute, dass ich – im Vorgriff – schon bisher die STL einsetzte, allerdings handelte es sich um `list`. Die notwendigen Veränderungen im Code sind derart gering, dass ich nur 3 Zeilen umzuschreiben brauchte:

1. Der Container `set` muss bekannt sein:
   ```
   #include <set>
   ```

2. Die Typdefinition der Menge muss nun lauten:
   ```
   typedef set<CRoom> TSetOfRooms;
   ```

3. Ein Element kann nicht mehr mit `push_back` hinzugefügt werden, sondern mit:
   ```
   m_Set.insert(room);
   ```

Ich empfand dies als ein wirklich eindrucksvolles Beispiel, wie sauber die STL konstruiert ist, da man sehr schnell das Programm den Umgebungsanforderungen anpassen kann, um so z.B. Optimierungen in Bezug auf Einfüge- oder Zugriffsoperationen durchzuführen.

Wenn man `set` mit einer eigenen Vergleichsoperation ausstatten will, dann ist es am besten, man führt ein eigenes Template dafür ein:

```
template <typename T>
class MyLess : public std::binary_function<T, T, bool>
{
public:
    bool operator() (const T &t1, const T &t2) const
    {
        return t1 < t2;
    }
};
```

Der eigentliche Vergleich wird doch wieder auf den <-Operator zurückgeführt, das begründet sich aber nur darin, dass ich ihn schon eh ausprogrammiert hatte. Sollte man jedoch Klassen benutzen müssen, für die man nicht (nachträglich) diesen Operator vorsieht, so kann man im Template auch direkt die entsprechenden Attribute, wie z.B. die Raumnummer, vergleichen.

Eine Menge kann für das Haus nun auch erzeugt werden, indem die Typdefinition abgeändert wird auf:

```
    typedef set<CRoom, MyLess<CRoom> >  TSetOfRooms;
```

Quellcode

Ich führe im folgenden lediglich die geänderten Dateien auf, d.h. die Interfaces bzw. Implementierungen von `CRoom`, `CCashOffice`, `CHouse` und natürlich das neue Vergleichstemplate `MyLess` sowie die Hauptdatei C1_1.cpp. Sehr viel Funktionalität ist 1:1 übernommen worden, die wesentlichen Neuerungen habe ich bereits auf der vorherigen Seite besprochen.

MyLess.tli

```
//////////////////////////////////////////////////////
// Softwarebauelemente II, Aufgabe C1.1
//
// author:          Stephan Brumme
// last changes:    July 3, 2001


#include <functional>

// Template that compares two objects

template <typename T>
class MyLess : public std::binary_function<T, T, bool>
{
public:
    bool operator() (const T &t1, const T &t2) const
    {
        return t1 < t2;
    }
};
```

Room.h

```
//////////////////////////////////////////////////////
// Softwarebauelemente II, Aufgabe C1.1
//
// author:          Stephan Brumme
// last changes:    July 3, 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;
    // needed to use in a STL set
    bool operator < (const CRoom &room) 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 C1.1
//
// author:          Stephan Brumme
// last changes:    July 3, 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
{
    // check invariance
    if (!ClassInvariant())
        return "";

    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
    const CRoom *room = dynamic_cast<const CRoom*>(pClass);

    // 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
    const CRoom *room = dynamic_cast<const CRoom*>(pClass);

    // 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;
}


// needed to use a STL set
bool CRoom::operator<(const CRoom &room) const
{
    return m_nNumberOfRoom < room.m_nNumberOfRoom;
```

```
}
```

CashOffice.h

```
//////////////////////////////////////////////////////
// Softwarebauelemente II, Aufgabe C1.1
//
// author:          Stephan Brumme
// last changes:    July 3, 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 cashoffices
    virtual bool operator==(const CCashOffice& cashOffice) const;
    virtual bool EqualValue(const CBasicClass* pClass) const;
    // needed to use in a STL set
    bool operator < (const CCashOffice &cashOffice) 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 C1.1
//
// author:          Stephan Brumme
// last changes:    July 3, 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
{
    // check invariance
    if (!ClassInvariant())
        return "";

    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
    const CCashOffice *cashOffice = dynamic_cast<const CCashOffice*>(pClass);

    // 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
    const CCashOffice *cashOffice = dynamic_cast<const CCashOffice*>(pClass);

    // 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();
}


// needed to use a STL set
bool CCashOffice::operator<(const CCashOffice &cashOffice) const
{
    return m_nNumberOfCounter < cashOffice.m_nNumberOfCounter;
}
```

House.h

```
//////////////////////////////////////////////////////
// Softwarebauelemente II, Aufgabe C1.1
//
// 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;

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

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

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

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

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

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

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

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

    // insert a new room into the set, TRUE if successful
    // will fail if room already exists
```

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

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

private:
    CDate m_dtDateOfFoundation;

    TSetOfRooms m_Set;
    TSetCursor  m_itCursor;
};

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

House.cpp

```
//////////////////////////////////////////////////////
// Softwarebauelemente II, Aufgabe C1.1
//
// 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();
}


// 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.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;
}


// 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();
}
```

C1_1.cpp

```
//////////////////////////////////////////////////////
// Softwarebauelemente II, Aufgabe C1.1
//
// 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();
}
```