1 Organisatorisches

1.1 Team

- Reinhard Penn, s1110306019
- Bernhard Selymes, s1110306024

1.2 Aufteilung

- Reinhard Penn
  - Planung
  - Klassendiagramm
  - Implementierung und Testen der Klassen Side, Wall, Door
- Bernhard Selymes
  - Planung
  - Klassendiagramm
  - Implementierung und Testen der Klassen Object, Roomlayout, Room
  - Dokumentation

1.3 Zeitaufwand

- geschätzte Mh: 7h
- tatsächlich: Reinhard (?h), Bernhard (6h)

2 Systemspezifikation

Eine Software für einen Raumplan soll entwickelt werden. Ein Raumplan enthält mehrere Räume, jeder Raum hat 4 Seiten, die wiederum eine Wand oder ein Durchgang sein können. Die Räume sind alle gleich groß und liegen alle untereinander und nicht nebeneinander. Eine Wand hat eine Farbe, ein Durchgang ist offen oder geschlossen und verbindet einen oder zwei Räume - es gibt auch Durchgänge die nach draußen führen. Wenn zwei Durchgänge aufeinander treffen wird nur einer ausgegeben. Bei zwei Wänden werden beide ausgegeben. In der gesamten Datenstruktur können nur Objekte hinzugefügt, aber nicht gelöscht werden. Der gesamte Raumplan kann ausgedruckt werden.

3 Systementwurf

3.1 Klassendiagramm

3.2 Komponentenübersicht

- Klasse ”Object”:
  Basis aller Basisklassen.
- Klasse ”RoomLayout”:
  Beinhaltet die Räume und kann diese mithilfe einer Printfunktion ausgeben.
- Klasse ”Room”:
  Beinhaltet vier Seiten und kann mithilfe einer Printfunktion ausgegeben werden.
- Klasse ”Side”:
  Abstrakte Klasse, von der Wall und Door abgeleitet werden. Speichert die Himmelsrichtung der Seite.
- Klasse ”Wall”:
  Hat einen Member der die Farbe der Wand speichert.
- Klasse ”Door”:
  Speichert ob Door offen oder geschlossen ist und welche Räume von Door verbunden werden.
- Enumeration ”Direction”:
  Hat vier Zustände: North, West, East, South

4 Komponentenentwurf

4.1 Klasse ”Object”

Abstrakte Basisklasse aller Klassen. Von ihr werden alle anderen Klassen abgeleitet. Beinhaltet einen virtuelle Dekonstruktor.

4.2 Klasse ”RoomLayout”

Besitzt eine Liste die Zeiger auf die Klasse Room beinhaltet, einen Zeiger, der auf den vorherigen Raum zeigt und einen Wahrheitswert, der angibt ob im letzten Raum im Süden ein Durchgang war.

Methode ”Print”: Gibt die Räume die in der Liste gespeichert sind der Reihe nach aus und merkt sich immer ob im vorherigen Raum eine Durchgang im Süden war. Wenn ein Durchgang war wird der Durchgang nur einmal ausgegeben.

Methode ”AddRoom”: Fügt Räume zum Raumplan hinzu. Ein Raum wird nur dann hinzugefügt wenn die südliche Seite der vorherigen Raumes mit der nördlichen Seite des Raumes, der eingefügt werden soll, übereinstimmt oder der gesamte Raumplan noch leer ist. Wenn ein Durchgang war: Im vorherigen Raum wird bei Durchgang der aktuelle Raum hinzugefügt und beim Durchgang vom aktuellen Raum wird der vorherigen Raum gespeichert. Weiters wird gespeichert ob der aktuelle Raum einen Durchgang im Süden hat und der Zeiger, der auf den vorherigen Raum gezeigt hat, zeigt jetzt auf den aktuellen Raum.

4.3 Klasse ”Room”

Beinhaltet einen Vektor der Zeiger auf die Klasse Side hat.

Methode ”Print”: Gibt den Raum aus. Durchgang wird im Norden nicht ausgegeben, wenn im letzten Raum im Süden ein Durchgang war. Die restlichen Seiten werden ganz normal ausgegeben, eine Wand im Norden oder Süden wird durch 9 Sterne gekennzeichnet, im Westen oder Osten durch 5. Ein Durchgang wird durch ein ”D” gekennzeichnet.

Methode ”AddSide”: Fügt Seiten zu einem Raum hinzu. Eine Seite wird nur dann hinzugefügt, wenn noch keine Seite für diese Himmelsrichtung existiert. Der Vektor mit den Seiten wird nach dem Einfügen nach der Definiton der Enumeration sortiert: North, West, East, South. Wenn eine Seite ein Durchgang ist, wird im Durchgang der aktuelle Raum hinzugfügt.

4.4 Klasse ”Side”

Abstrakte Klasse, die eine Seite eines Raumes repräsentiert. Enthält einen Member der die ”Direction” der Seite speichert und einen der angibt ob es ein Durchgang oder eine Wand ist.

4.5 Klasse ”Wall”

Repräsentiert eine Wand. Hat einen Konstruktor dem die Farbe und die Richtung der Wand übergeben werden.

4.6 Klasse ”Door”

Repräsentiert einen Durchgang. Hat einen Member der speichert ob die Tür offen oder geschlossen ist und einen Vektor der die Räume speichert, die der Durchgang verbindet.

Methode ”AddRoom”: Fügt zum Durchgang einen Raum hinzu, wenn noch nicht die maximale Anzahl der Räume die ein Durchgang verbinden kann erreicht ist.

4.7 Enumeration ”Direction”

- North
- West
- East
- South

5 Testprotokollierung

6 Source Code

```
1  #ifndef OBJECT_H
2  #define OBJECT_H
3
4  class Object
5  {
6  public:
7     virtual ~Object();
8  protected:
9     Object();
10 };
11
12 #endif
```

```
1 #include "Object.h"
2
3 Object::Object()
4 {}
5
6 Object::~Object()
7 {}
```

```
1  #ifndef ROOMLAYOUT_H
2  #define ROOMLAYOUT_H
3
4  #include <list>
5  #include "Object.h"
6  #include "Room.h"
7
8  typedef std::list<Room*> Rooms;
9  typedef Rooms::const_iterator RoomsItor;
10
11 class RoomLayout :
12    public Object
13 {
14 public:
15    RoomLayout();
16    virtual ~RoomLayout();
17    void Print();
18    bool AddRoom(Room* room);
19
20 private:
21    Rooms mRooms;
22    Room* prevRoom;
23    bool mWasDoor;
24
25    void PrintRoom(Room* room, bool WasDoor);
26 };
27
28 #endif
```

```cpp
#include <iostream>
#include <algorithm>
#include <iterator>
#include "RoomLayout.h"


RoomLayout::RoomLayout()
   : mWasDoor(false)
{
}

RoomLayout::~RoomLayout()
{
   RoomsItor itor = mRooms.begin();

   while (itor != mRooms.end())
   {
      delete (*itor);
      ++itor;
   }
}

void RoomLayout::PrintRoom(Room* room, bool WasDoor)
{
   room->Print(WasDoor);
}

void RoomLayout::Print()
{
   bool WasDoor = false;

   for (std::list<Room*>::const_iterator first = mRooms.begin() ; first!=
      mRooms.end(); ++first )
   {
      PrintRoom(*first, WasDoor);
      WasDoor = (*first)->IsSouthDoor();
   }
}

bool RoomLayout::AddRoom(Room* room)
{
   if (mWasDoor == room->IsNorthDoor() || mRooms.empty())
   {
      mRooms.push_back(room);

      if (mWasDoor)
      {
         Side* CurrentDoor = room->GetNorthSide();
         Side* PrevDoor = prevRoom->GetSouthSide();

         prevRoom->AddRoomToDoor(CurrentDoor);
         room->AddRoomToDoor(PrevDoor);
      }

      mWasDoor = room->IsSouthDoor();
      prevRoom = room;
      return true;
   }
   return false;
}
```

```
1  #ifndef ROOM_H
2  #define ROOM_H
3
4  #include <vector>
5  #include <string>
6  #include "Object.h"
7  #include "Side.h"
8
9
10 typedef std::vector<Side*> TVec;
11 typedef TVec::const_iterator TVecItor;
12
13 std::string const WallString_N = "*********";
14 std::string const DoorString_N = "****D****";
15 std::string const SideSpaces = "        ";
16 char const WallSign = '*';
17 char const DoorSign = 'D';
18
19 int const MaxWalls = 4;
20
21 class Room :
22    public Object
23 {
24 public:
25    Room();
26    virtual ~Room();
27
28    void Print(bool WasDoor);
29
30    bool AddSide(Side* side);
31    void AddRoomToDoor(Side* door);
32
33    bool IsNorthDoor();
34    bool IsSouthDoor();
35
36    Side* GetNorthSide();
37    Side* GetSouthSide();
38
39 private:
40    std::vector<Side*> mSides;
41
42    void PrintWallOrDoorNS(Side* side);
43    void PrintWallOrDoorOW(Side* side);
44    void PrintTwoWallParts();
45    void PrintSpaces();
46 };
47
48 #endif
```

```
1  #include <algorithm>
2  #include <iterator>
3  #include <vector>
4  #include <iostream>
5  #include <string>
6  #include "Room.h"
7  #include "Door.h"
8
9
10 Room::Room()
11 {
12 }
13
14 Room::~Room()
15 {
16    TVecItor itor = mSides.begin();
17
18    while (itor != mSides.end())
19    {
20       delete (*itor);
21       ++itor;
22    }
23 }
24
25 void Room::PrintWallOrDoorNS(Side* side)
26 {
27    if (!side->IsDoor())
28    {
29       std::cout << WallString_N << std::endl;
30    }
31    else
32    {
33       std::cout << DoorString_N << std::endl;
34    }
35 }
36
37 void Room::PrintWallOrDoorOW(Side* side)
38 {
39    if (!side->IsDoor())
40    {
41       std::cout << WallSign;
42    }
43    else
44    {
45       std::cout << DoorSign;
46    }
47 }
48
49 void Room::PrintTwoWallParts()
50 {
51    std::cout << WallSign;
52    PrintSpaces();
53    std::cout << WallSign << std::endl;
54 }
55
56 void Room::PrintSpaces()
57 {
58    std::cout << SideSpaces;
59 }
60
```

```
61  void Room::Print(bool WasDoor)
62  {
63     try
64     {
65        //Checks if the room has 4 Sides
66        if (mSides.size() != MaxWalls) {
67           throw("Sides are missing");
68        }
69
70        TVecItor itor = mSides.begin();
71
72        //N
73        if (!WasDoor)
74        {
75           PrintWallOrDoorNS(*(itor));
76        }
77        ++itor;
78
79        //W+O
80        PrintTwoWallParts();
81        PrintWallOrDoorOW((*itor));
82        PrintSpaces();
83        ++itor;
84        PrintWallOrDoorOW((*itor));
85        std::cout << std::endl;
86        PrintTwoWallParts();
87        ++itor;
88
89        //S
90        PrintWallOrDoorNS(*(itor));
91     }
92     catch (std::string const& ex)
93     {
94        std::cerr << "Error occured in Room::Print: " << ex << std::endl;
95     }
96     catch (...)
97     {
98        std::cerr << "Unknown Error in Room::Print" << std::endl;
99     }
100 }
101
102 bool CheckSideOrder(Side* side1, Side* side2)
103 {
104    return (side1->getDirection()) < (side2->getDirection());
105 }
106
107 bool Room::AddSide(Side* side)
108 {
109    //checks if max amount of walls have been reached
110    if (mSides.size() < MaxWalls)
111    {
112       //only inserts walls which doesn't already exist
113       for (TVecItor itor = mSides.begin(); itor != mSides.end(); ++itor)
114       {
115          if ((*itor)->getDirection() == side->getDirection())
116          {
117             return false;
118          }
119       }
120       mSides.push_back(side);
```

```
121         //Sorts the Sides
122         std::sort(mSides.begin(),mSides.end(),CheckSideOrder);
123 
124         if (side->IsDoor())
125         {
126            AddRoomToDoor(side);
127         }
128 
129         return true;
130      }
131      return false;
132   }
133 
134   bool Room::IsNorthDoor()
135   {
136      return mSides[0]->IsDoor();
137   }
138 
139   bool Room::IsSouthDoor()
140   {
141      return mSides[MaxWalls-1]->IsDoor();
142   }
143 
144   void Room::AddRoomToDoor(Side* side)
145   {
146      Door* door = dynamic_cast<Door*>(side);
147      door->AddRoom(this);
148   }
149 
150   Side* Room::GetNorthSide()
151   {
152      return mSides[0];
153   }
154 
155   Side* Room::GetSouthSide()
156   {
157      return mSides[MaxWalls-1];
158   }
```

```
#ifndef SIDE_H
#define SIDE_H

#include "Direction.h"
#include "Object.h"


class Side :
   public Object
{
public:
   Side();
   virtual ~Side();

   bool IsDoor();
   Direction getDirection();

protected:
   bool mIsDoor;
   Direction mDirection;
};

#endif
```

```
1  #include "Side.h"
2
3
4  Side::Side()
5  {
6  }
7
8  Side::~Side()
9  {
10 }
11
12 Direction Side::getDirection()
13 {
14    return mDirection;
15 }
16
17 bool Side::IsDoor()
18 {
19    return mIsDoor;
20 }
```

```
#ifndef WALL_H
#define WALL_H

#include <string>
#include "Side.h"


class Wall :
   public Side
{
public:
   Wall();
   Wall(std::string color, Direction direction);
   virtual ~Wall();

private:
   std::string mColor;
};

#endif
```

```cpp
#include <iostream>
#include "Wall.h"


Wall::Wall(std::string color, Direction direction)
{
   mColor = color;
   mDirection = direction;
   mIsDoor = false;
}

Wall::~Wall()
{
}
```

```
1  #ifndef DOOR_H
2  #define DOOR_H
3
4  #include <vector>
5  #include "Side.h"
6  #include "Room.h"
7
8
9  int const MaxRooms = 2;
10
11 class Door :
12    public Side
13 {
14 public:
15    Door(bool isOpen, Direction direction);
16    virtual ~Door();
17
18    void AddRoom(Room* room);
19
20 private:
21    bool mIsOpen;
22    std::vector<Room*> mRooms;
23 };
24
25 #endif
```

```
1  #include "Door.h"
2
3
4  Door::Door(bool isOpen, Direction direction)
5  {
6     mIsDoor = true;
7     mIsOpen = isOpen;
8     mDirection = direction;
9  }
10
11 Door::~Door()
12 {
13 }
14
15 void Door::AddRoom(Room* room)
16 {
17    if (mRooms.size() < MaxRooms)
18    {
19       mRooms.push_back(room);
20    }
21 }
```

```
#ifndef DIRECTION_H
#define DIRECTION_H

enum Direction
{
   North,
   West,
   East,
   South
};

#endif
```

```
1  #include <vld.h>
2  #include "Room.h"
3  #include "Wall.h"
4  #include "Door.h"
5  #include "RoomLayout.h"
6
7  int main()
8  {
9     Side* s1 = new Wall("green",North);
10    Side* s2 = new Wall("blue", West);
11    Side* s3 = new Door(false, East);
12    Side* s4 = new Door(true, South);
13
14    Room* r1 = new Room;
15    r1->AddSide(s3);
16    r1->AddSide(s1);
17    r1->AddSide(s2);
18    r1->AddSide(s4);
19
20
21    Side* s5 = new Door(true, North);
22    Side* s6 = new Door(false, West);
23    Side* s7 = new Door(false, East);
24    Side* s8 = new Wall("green", South);
25
26    Room* r2 = new Room;
27    r2->AddSide(s5);
28    r2->AddSide(s6);
29    r2->AddSide(s7);
30    r2->AddSide(s8);
31
32
33    Side* s9 = new Wall("yellow", North);
34    Side* s10 = new Door(false, West);
35    Side* s11 = new Door(false, East);
36    Side* s12 = new Door(false, South);
37
38    Room* r3 = new Room;
39    r3->AddSide(s9);
40    r3->AddSide(s10);
41    r3->AddSide(s11);
42    r3->AddSide(s12);
43
44    RoomLayout rl1;
45    rl1.AddRoom(r1);
46    rl1.AddRoom(r2);
47    rl1.AddRoom(r3);
48
49    rl1.Print();
50
51    return 0;
52 }
```

7 Testausgaben