Praktikum 4: Delegation

1. Lernziele

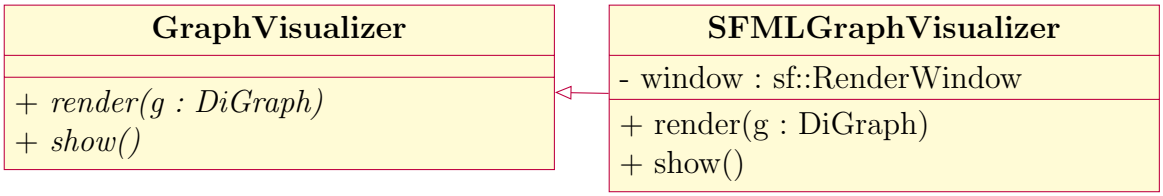
Die folgenden, in der Vorlesung behandelten Themen sollen vertieft und angewendet werden:

- Ableitung von Klassen
- Abstrakte Klassen
- Polymorphie
- Delegation

2. Aufgabe

Teil 1: Erweitern Sie den Graphen aus Praktikum 3 um eine Funktion, die für einen gegebenen Start- und Endknoten den kürzesten Pfad berechnet und zurückgibt. Der Funktionsprototyp soll wie folgt aussehen:
`Liste<Edge*> dijkstraShortestPath(std::string start, std::string end)`
Die Funktionsweise des Dijkstra-Algorithmus können Sie dem gegebenen Algorithmus auf Seite 3 oder dem Pseudocode auf Seite 4 im Anhang entnehmen. Verwenden Sie für die Datenstruktur PQ, die in Praktikum 2 entwickelte Vorrangwarteschlange mit dem Datentyp `Node*` als Template.

Teil 2: Um die Visualierung des Graphen austauschbar zu machen, erweitern Sie die Klasse `SFMLGraphVisualizer` um eine abstrakte Basisklasse `GraphVisualizer`.



Fügen Sie der Klasse `DiGraph` die folgenden Attribute und Methoden hinzu:
Attribute

- `GraphVisualizer* graphviz`
  Instanzobjekt der `GraphVisualizer`-Klasse

Methoden

- `void setVisualizer(GraphVisualizer* graphviz)`
- `GraphVisualizer* getVisualizer()`

Erweitern Sie die Basisklasse `GraphVisualizer` um die Methode `highlightPath` und implementieren Sie diese in der `SFMLGraphVisualizer` Klasse:

- `void highlightPath(Liste<Edge*> path)`

3. Test

Testen Sie Ihre Implementierung mit dem beiliegenden Hauptprogramm.

```
1  //Einbinden der Bibliothek
2  #include <digraph.hpp>
3  #include <node.hpp>
4  #include <string>
5  #include <iostream>
6  #include "graphvisualizer.hpp"
7  #include "vehicle.hpp"
8  #include "navigator.hpp"
9
10 using namespace std;
11
12 void createDummyGraph(DiGraph &g) {
13     g.addNode( new Node("Aachen", 100, 600) );
14     g.addNode( new Node("Berlin", 300, 650) );
15     g.addNode( new Node("Koeln", 300, 300) );
16     g.addNode( new Node("Essen", 900, 300) );
17     g.addNode( new Node("Bonn", 300, 150) );
18     g.addNode( new Node("Krefeld", 100, 160) );
19
20     g.createEdge("Aachen", "Berlin", 7);
21     g.createEdge("Koeln", "Aachen", 9);
22     g.createEdge("Aachen", "Krefeld", 7);
23     g.createEdge("Berlin", "Essen", 40);
24     g.createEdge("Berlin", "Koeln", 3);
25     g.createEdge("Koeln", "Essen", 31);
26     g.createEdge("Bonn", "Essen", 8);
27     g.createEdge("Krefeld", "Bonn", 1);
28 }
29
30 int main(int argc, char ** argv){
31
32     DiGraph myGraph;
33     SFMLGraphVisualizer graphviz;
34     myGraph.setVisualizer(&graphviz);
35
36     createDummyGraph(myGraph);
37     Navigator navi(&myGraph);
38     int speed = 120;
39     navi.setVehicle(new Car(speed));
40     std::cout << "Fahrzeit: "
41         << navi.planRoute("Aachen", "Essen") << std::endl;
42 }
```

4. Testat

Voraussetzung ist jeweils ein fehlerfreies, korrekt formatiertes Programm. Der korrekte Programmlauf muss nachgewiesen werden. Sie müssen in der Lage sein, Ihr Programm im Detail zu erklären und ggf. auf Anweisung hin zu modifizieren.

Das Praktikum muss spätestens zu Beginn des nächsten Praktikumtermins vollständig bearbeitet und abtestiert sein.

5. Anhang

Algorithm 1: DijkstraShortestPath finds the shortest path between two nodes in a graph with non-negative weights

Input: A graph $G = (V, E)$ as ordered pair of vertices and edges, a start node $s \in G$, an end node d $\in G$

Output: The shortest path from s to d in G

```
1  PQ ← {}
2  for each v ∈ G do
3      if v ≠ s then
4          dist[v] ← ∞
5      else
6          dist[v] ← 0
7      previous[v] ← null
8      PQ.insert(v, dist[v])
9  while PQ is not empty do
10     u ← PQ.extractMin()
11     for each neighbor v of u do
12         alt ← dist[u] + weight(u, v)
13         if alt < dist[v] then
14             dist[v] ← alt
15             previous[v] ← u
16             PQ.decreaseKey(v, alt)
17 S ← []
18 u ← d
19 while u is not null do
20     insert u at the beginning of S
21     u ← previous[u]
22 return S
```

```
ALGORITHMUS Dijkstra

BESCHREIBUNG:
Der Djikstra-Algorithmus sucht in einem Graphen den kürzesten Weg zwischen zwei Knoten

PARAMETER IN:
    Einen Graphen:  G
    Startknoten: s
    Endknoten: d
    Array mit Knoten: nodes
    Anzahl der Knoten im Graph: used

PARAMETER OUT:
    Liste mit kürzestem Weg von s nach d:  result


KERNALGORITHMUS

    Neue Liste result aus Kanten anlegen

    Neue Warteschlange (PriorityQueue) 'PQ' aus Knotenelementen anlegen

    float dist[used]

    Node* previous[used];

    FUER i = 0 BIS used SCHRITTWEITE 1

        FALLS nodes[i] == s DANN

            dist[i] = 0

        SONST
            dist[i] = unendlich

        ENDE FALLS

        previous[i] = NULL

        Füge der Warteschlange den Wert nodes[i] mit Priorität dist[i] hinzu.

    ENDE FUER

    SOLANGE ( PQ nicht leer ist)

        u = Element mit höchster Priorität aus PQ

        Lege Array outEdges an mit allen ausgehenden Kanten von u an

        size = Anazhl der Elemente in outEdges

        FÜR i = 0 BIS size SCHRITTWEITE 1

```

```
54              Node *v = Endknoten von outEdges[i]
55
56              float alt = dist[u] + Gewicht von outEdges[i]
57
58              FALLS alt < dist[v] DANN
59
60                  dist[v] = alt
61
62                  previous[v] = u
63
64                  Ändere die Priorität von v in der Warteschlange auf alt
65
66              ENDE FALLS
67
68          ENDE FÜR
69
70      ENDE SOLANGE
71
72      u = d
73
74      SOLANGE( previous[u] != NULL)
75
76          Die Kante von previous[u] nach u der Liste result hinzufügen
77
78          u = previous[u]
79
80      ENDE SOLANGE
81
82      Rückgabe von result
83
84  ENDE ALGORITHMUS
```