Abend 1
Notenliste

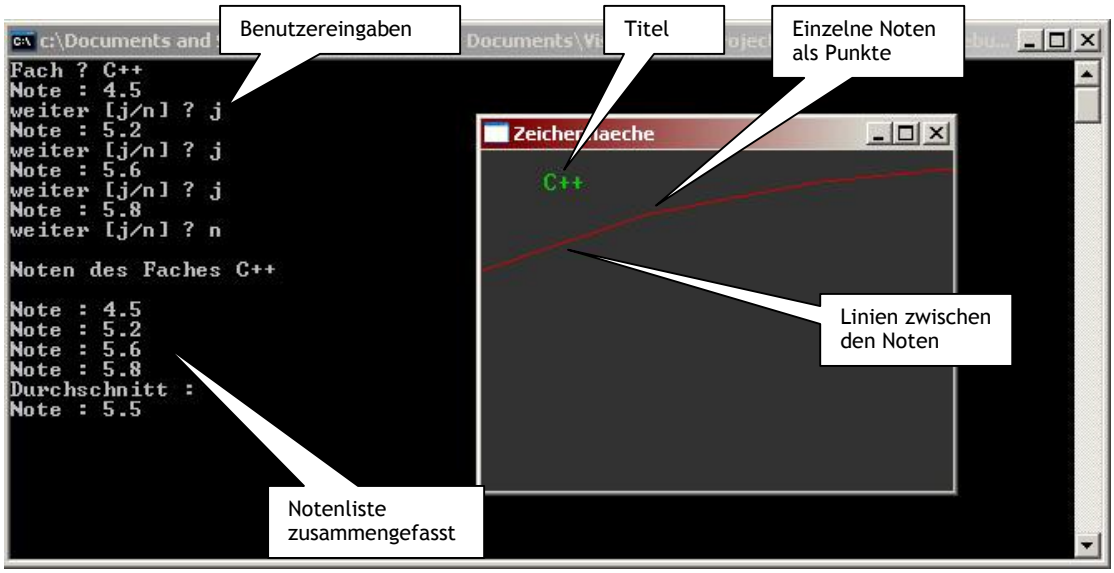
Ziel, Inhalt

- Wir werden anhand des Beispiels Notenliste eine Problemstellung in ein objektorientiertes Design umwandeln und implementieren. Wir werden auf dem Weg dahin folgende Themen aufgreifen: Konstruktoren, Referenzen und Container-Klassen der STL (Standard Template Library)

- Parallel dazu werden wir Themen wie sichere, fehlerfreie Programmierung und gut lesbare und trotzdem effiziente Programmierung aufgreifen

Abend 1 Notenliste 1
Ziel, Inhalt 1
Das Programm Notenliste 2
Zeichenfläche 2
Analyse-Modell 3
Übung 3
Statisches Klassenmodell 3
Klassenbeschreibung 3
Design und Implementation 4
Klasse Note 4
Übung Note 4
Profi Tipps: const-correctness 4
Übung Methode runden() 5
Profi-Tipps: Anwendung von assert 6
Anwendung von *assert* in der Klasse Note 7
Klasse Notenliste 8
Profi-Tipp: Zur Wahl von vector oder list 8
Noten im vector 8
Profi-Tipp: Schmale Schnittstellen 9
Notenliste mit vector 9
Übung Klasse Notenliste 9

Das Programm Notenliste



Diese Zeichnung zeigt ein Programm, das die Mindestanforderungen erfüllt. Wir haben eine Benutzerschnittstelle (Benutzereingaben), in der wir den Namen des Faches eingebenen können und danach die einzelnen Noten. Eine Notenliste wird auch graphisch dargestellt. Die Graphik enthält den Namen des Faches. Sobald mehr als zwei Noten vorhanden sind, sollen diese durch verbundene Linien dargestellt werden.
Sobald der Benutzer die Eingaben abschliesst, sollen die Noten zusammengefasst ausgegeben werden. Der Durchschnitt wird auch angegeben.
Eine weitere Anforderung ist, dass die einzelnen Noten gerundet werden können. Der Benutzer soll die Noten so eingeben, wie er sie erhält, der Durchschnitt soll aber auf Ganze, Halbe, Viertel oder Zehntelnoten genau gerundet ausgegeben werden können.

Zeichenfläche

Für die graphische Ausgabe steht ein sogenannter COM-Server zur Verfügung, der in der Lage ist eine Zeichenfläche als Fenster darzustellen. Auf der Zeichenfläche gibt es die Möglichkeit Texte, Kreise und Linien darzustellen. Um den Server zu verwenden ist es nötig gewisse Dateien vom Internet zu holen und zu installieren. Die gezippten Dateien findest Du unter:
www.devmentor.ch/teaching/additional/TsuZeichnen/TsuZeichnen.zip
Eine Anleitung, wie diese Dateien zu verwenden sind, findest du in folgender pdf-Datei:
www.devmentor.ch/teaching/additional/01I/TsuZeichnen/UebungZeichnen.pdf. Beachte dort nur die Beschreibung, wie die Klassen angewendet werden, die Übung kannst Du natürlich freiwillig auch lösen.

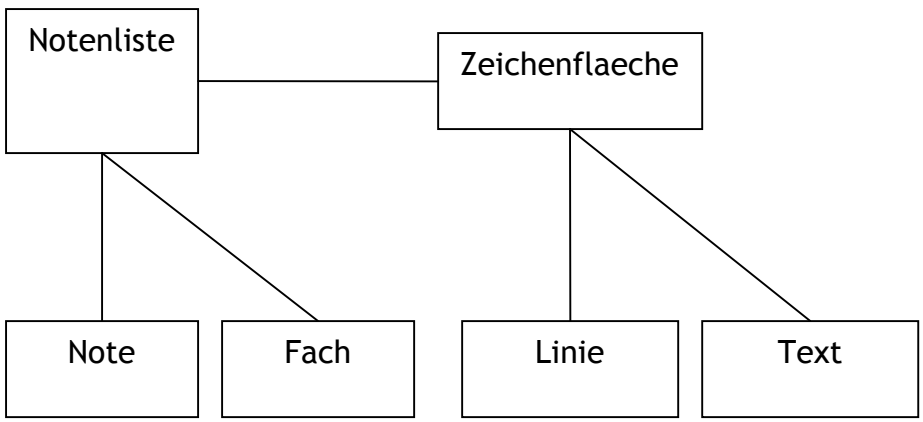
Analyse-Modell

Während der Analyse ist es von Vorteil nicht zu weit voraus zu denken, sondern einfach eine Zeichnung erstellen und darauf mögliche Klassennamen in Rechtecken aufzuzeichnen. Dabei muss jeglicher Gedanke an die Implementation unterbunden werden, es dürfen dabei keine Gedanken an Datentypen oder dynamische Abläufe verschwendet werden. Das Ziel ist es ein statisches Klassenmodell zu erhalten, welches als Grundlage für die weiteren Schritte herhalten kann.
Versuche jetzt also Klassen zu finden. Schau Dir das Bild und die Beschreibung der Problemstellung an. Eine gute Taktik ist es die Nomen anzusehen und sich schnell zu überlegen, ob diese als Klassen Sinn machen.

Übung

Zeichne ein Klassenmodell, indem du du Klassennamen in Rechtecke schreibst. Falls verschiedene Klassen ein Beziehung zueinander haben, kannst du das spezifizieren, indem du Linien zwischen den Klassen einzeichnest.

Statisches Klassenmodell



Klassenbeschreibung

- Note, enthält die Note und ist in der Lage diese Note gerundet auszugeben, oder zurückzugeben

- Notenliste, enthält mehrere Noten und das Fach. Sie ist in der Lage neue Noten zu erzeugen. Die Liste kann auf der Konsole ausgegeben werden. Zu jeder Liste gehört eine Zeichenfläche mit Linien.

- Fach, enthält die Bezeichnung des Faches

- Zeichenflaeche, Linie und Text, diese Klassen sind gegeben und bereits dokumentiert

Design und Implementation

In einem grossen Projekt werden diese beiden Punkte einzeln behandelt und dokumentiert. Für dieses kleine Projekt werden wir aber, nachdem wir uns einige Gedanken zum Design gemacht haben, versuchen das direkt zu implementieren. Wir beginnen mit der Klasse Note.

Klasse Note

Übung Note

Diese Klasse kapselt also eine Note. Versuche als Übung eine Klasse Note zu definieren. Erzeuge zuerst ein neues Projekt mit dem Visual-Studio (Konsolen-Anwendung) und definiere die Klasse Note in einer eigenen Header-Datei. Was für Datenelemente braucht diese Klasse ? Welche Methoden sind sinnvoll? Was für Konstruktoren sind sinnvoll? Schau Dir zur Repetition den Stoff an (aktuelle Kursunterlagen findest Du auch hier : www.devmentor.ch/teaching/additional/01I/Abend12/Abend12.html).
Zur Klasse Note gehört auch die Möglichkeit einen gerundeten Wert zu berechnen. Wir schränken die Rundung vorerst auf ein paar wenige Rundungswerte ein und definieren hierfür eine Aufzählung (enum) Rundung. Definieren eine solche Methode (z.B. holeGerundeteNote(Rundung x)). Anstelle von Methoden um den Notenwert zu setzen/lesen zu schreiben (get/set), überlege dir, was du wirklich brauchst. Möglicherweise brauchst du nur Methoden wie : einlesen(), ausgeben(), gerundetAusgeben(), etc.

Profi Tipps: const-correctness

Es ist wichtig sich an *const* zu gewöhnen! Den *const* ist dein Freund. Methoden, die ein Objekt nicht ändern, sollen *const* definiert sein. Ein Objekt bleibt unverändert, wenn sich seine Datenelemente in einem Methodenaufruf nicht ändern. Solche Methoden lassen sich am einfachsten finden, wenn das Design stimmt und die Methoden vernünftige Namen haben.

```
class Namen
{
    public:
        string getName() const // das Objekt bleibt unverändert
    private:
        string m_name;
};
string Namen::getName() const
{
    return m_name;
}
```

Eine Methode setNamen wäre hier oben nicht const, den das Datenelement *m_name* würde sich dabei ändern.
const bei Parameterübergabe hat eine andere Bedeutung:

```
class Namen
{
    public:
        void setName(string name);
    private:
        string m_name;
};

void Namen::setName(string name)
{
    m_name = name;
}
```

Die Methode setName erzeugt beim Aufruf aus folgendem main eine Kopie des strings:

```
int main()
{
    Namen einNamen;
    string test(„Markus“);
    einNamen.setName(test);
    return 0;
}
```

Da dabei viele Buchstaben kopiert werden müssen, ist das nicht so effizient. Durch Übergabe per Referenz sparen wir uns diese String-Kopie. Da wir aber sicherstellen wollen, dass der string (hier *test*) dabei nicht geändert wird übergeben wir eine const-Referenz!

```
class Namen
{
    public:
        void setName(const string& name);
};

void Namen::setName(const string& name)
{
    m_name = name; // name bleibt unverändert!
}
```

Übung Methode runden()

Die Implementation der Klasse in der .cpp Datei sollte eigentlich bis auf die Methode zum Runden keine Probleme bereiten. Der Algorithmus zum Runden ist aber ein wenig trickreich.
Um in C++ Zahlen zu runden, nutzen wir den Umstand, dass wenn man einen *double* oder *float* in einen *int* umwandelt, die Stellen nach dem Komma verloren gehen. Einfach ist es die Stellen abzuschneiden wenn man ganze Noten braucht.

```
double test = 5.1;
int tg = (int)test;
test = tg; // ist jetzt 5.0
```

Die Variable tg wird den Wert 5 haben. Das ist zwar noch nicht korrekt gerundet, aber immerhin sind wir so die Nachkommastellen losgeworden. Um bei einer mehrstelligen Zahl nur die erste Stelle nach dem Komma zu behalten tun wir folgendes:

```
double test = 5.234;
test = test * 10;          // test ist jetzt 52.34
int tg = (int)test         // tg ist 52
test = (double)tg / 10.0; // test ist jetzt 5.2
```

Das gleiche funktioniert für Viertel, ersetze die 10 einfach durch eine 4! Damit das Runden auch korrekt funktioniert, müssen wir dafür sorgen, dass eine Note wie 5.25 bei der Rundung auf halbe Noten eine 5.5 ergibt:

```
double test = 5.25234;
double korrektur = 0.25;
test = test + korrektur;   // test ist gleich 5.50234
test = test * 2.0;         // test ist gleich 11.00468
int tg = (int)test;        // tg ist gleich 11
test = (double)tg / 2.0    // test ist gleich 5.5
```

Versuche die gewonnenen Erkenntnisse in eine Funktion zu packen, die mit allen gewünschten Rundungen zurechtkommt!

Profi-Tipps: Anwendung von assert

Hier noch ein Hinweis zur Vermeidung von Programmierfehlern mit *assert*! Mit *assert* ist es möglich ein Programm abstürzen zu lassen. Aber halt! Wir wollen doch nicht, dass unser Programm abstürzt!
Während der Entwicklung einer Software benutzen wir ja den Debugger (!). Es gibt übrigens eine Regel, die besagt dass neuer Code immer mit Debugger im Einzelschritt-Modus abgearbeitet werden muss. So lassen sich viele Fehler sofort finden.
Im Debugging-Modus sind wir in der Lage die falsche Verwendung unserer Methoden mit der *assert* Funktion festzustellen. Assert bedeutet soviel wie „stelle sicher dass :“.

Anwendung von *assert* in der Klasse Note

Eine Note soll auf Werte zwischen 1.0 und 6.0 beschränkt werden. In der Methode kann das die Methode einlesen() selber sicherstellen, indem der Benutzer wiederholt aufgefordert wird richtige Noten einzugeben. In einer Methode setzeNote(...) verwenden wir aber *assert*. In der Header Datei sieht die Methode setzeNote etwa so aus:

```
class Note
{
    public:
        ...
        // setzeNote akzeptiert nur Werte zwischen
        // 1.0 und 6.0, andernfalls -> assert!
        void setzeNote(double notenWert);
```

Wichtig ist dabei der Kommentar, denn der Benutzer der Klasse muss diesen Lesen, er steht ja in der öffentlichen Schnittstelle der Klasse! Die Methode in der .cpp Datei sieht dann so aus:

```
#include <cassert>
using namespace std;

void Note::setzNote(double notenWert)
{
    // Bitte gültigen Bereich beachten!
    assert(1.0 <= notenWert && 6.0 >= notenWert);

    m_notenWert = notenWert;
}
```

Wichtig ist auch hier der Kommentar und die Zeile mit dem assert, die soviel bedeutet wie:
„Stelle sicher dass gilt 1.0 <= notenWert <= 6.0“
Falls das nicht erfüllt ist erzeugt das Programm beim debuggen (in der Debug Version) einen Debug-Assertion Error.
Probier das mit einem kleinen Test-main aus!
Soviel zur Klasse Note. Schreibe sie fertig und teste sie ausgiebig mit verschiedenen Eingaben.
Das Visual Studio bietet ein weiteres assert an, mit dem beim Debuggen die fehlerhafte Zeile einfacher gefunden wird:

```
#include <crtdbg.h>

void Note::setzNote(double notenWert)
{
    // Bitte gültigen Bereich beachten!
    _ASSERT(1.0 <= notenWert && 6.0 >= notenWert);

    m_notenWert = notenWert;
}
```

Klasse Notenliste

Diese Klasse stellt ab jetzt den Dreh- und Angelpunkt unseres Designs dar. Die meisten Methoden, die man in der Aufgabenstellung „findet“, lassen sich mit der Notenliste in Verbindung bringen: „Notenliste ausgeben“, „Durchschnitt berechnen“, „Liste graphisch darstellen“ etc.
Die Notenliste enthält viele Noten, bei denen die Reihenfolge der Eingabe erhalten werden soll. Diese Information ist wichtig um den richtigen STL-Container auszuwählen. Folgende Container könnten für einen Haufen Noten Sinn machen:

```
set
list
vector
```

Das *set* ist ein ungeordneter Container, was nicht geeignet ist, denn die Noten wären nicht mehr nach Eingabereihenfolge geordnet. Es bleiben der *vector* und die *list*.

Profi-Tipps: Zur Wahl von vector oder list

Bei der Entscheidung ob eine list oder ein vector die bessere Wahl ist, gibt es ein paar Punkte, die die Entscheidung beeinflussen.
Müssen einzelne Elemente mitten in der Sammlung eingefügt oder entfernt werden, ist die Liste besser geeignet. Braucht man hingegen häufig den direkten Zugriff über einen Index, ist der Vektor die bessere Wahl. Beim nachträglichen Sortieren sollten die beiden Container fast gleich schnell sein, das gleiche gilt für das iterieren (sequentielles lesen) über den Container und für das Suchen.

Noten im vector

Für unser Beispiel sollen die Noten in einen vector gepackt werden. Wie das geschieht sehen wir hier:

```
#include „note.h“ // unsere Notenklasse
#include <vector> // STL

std::vector<Note> NotenVektor; // Definiere einen vector mit
                               // dem Namen NotenVektor
                               // und Objekten der Klasse Note
```

Die nächste Frage, die sich stellt ist, soll unsere Notenliste einen solchen vector enthalten, oder soll sie selber so ein vector sein, das heisst von vector<Note> abgeleitet sein?

```
typedef std::vector<Note> Noten; // definiere einen neuen Namen
                                 // für den Datentypen
                                 // std::vector<Note>

// Variante 1
class Notenliste
{
    private:
        Noten m_Noten;
};

// Variante 2
class Notenliste : public Noten
{
};
```

Im zweiten Fall verhält sich ein Objekt der Klasse Notenliste immer auch wie ein vector, alle Methoden, die in der Klasse vector vorhanden sind, sind auch für die Klasse Notenliste aufrufbar. Was zuerst praktisch tönt, ist nicht wirklich immer erwünscht. Ein Benutzer der Klasse könnte also spezielle Aufrufe auf ein Notenliste-Objekt machen, die nur für vector-Klassen vorhanden sind. Wenn ich später aber dann doch lieber eine *list* hätte, muss der Benutzer meiner Klasse seinen Code neu schreiben.

Profi-Tipps: Schmale Schnittstellen

Es ist im allgemeinen besser, ein Design so einfach und konkret zu halten wir möglich. Insbesondere soll die öffentliche Schnittstelle einer Klasse nur Methoden beinhalten, die wirklich gebraucht werden, andere sind purer Luxus und werden meist auch in Zukunft nicht gebraucht, denn häufig denken sich Programmierer: „Mann das ist cool, irgendwann könnte das nützlich sein“. Meine Antwort dazu ist „Yagni“ (You ain’t gonna need it = Du wirst es nicht brauchen).

Notenliste mit vector

Ich wähle also die Variante, bei der der vector Teil wird von der Notenliste und schreibe nur Methoden in die öffentliche Schnittstelle von Notenliste, die ich brauche. Dadurch ist niemand davon abhängig ob ich einen vector oder sonst irgendwas verwende um die Note-Objekte intern zu verwalten. Die Abhängigkeiten werden auf diese Weise gering gehalten.

Übung Klasse Notenliste

Definiere also eine Klasse Notenliste (Header-Datei) und finde heraus welche Datenelemente gebraucht werden. Welche Methoden werden gebraucht um unsere Problemstellung zu erfüllen. Lasse dabei die graphische Ausgabe ausser acht.