Dokumentation - Projekt Matrix - Informatik IIa

Diese Dokumentation ist nur für den Gebrauch in der Gruppe gedacht und soll als Orientierungshilfe dienen. Sie ist wie folgt gegliedert:

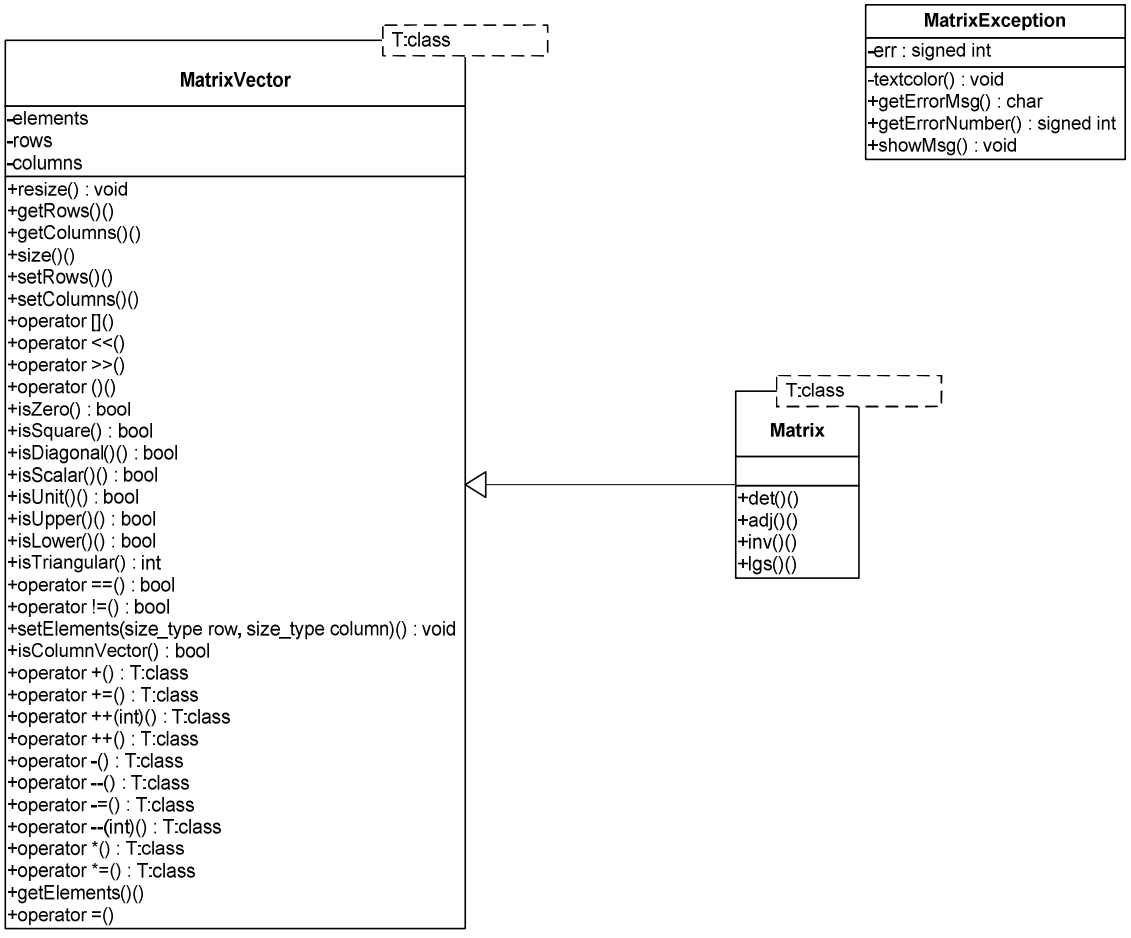
1. Klassenstruktur
2. Vererbung
3. Konstruktoren / Destruktoren
4. Attribute / Methoden
5. Fehlerbehandlung

Die genutzten Quellen werde ich an den entsprechenden Stellen in der Dokumentation nennen.

1. Klassenstruktur

Hier ein Klassendiagramm zum derzeitigen Stand. Da ich noch nicht herausgefunden habe in welcher Beziehung die Fehlerklasse „MatrixException“ steht, habe ich auch noch keine Verbindung gezogen.

Das Diagramm soll auch nur zur Übersicht sein. Erklärungen zu den Einzelheiten folgen in den entsprechenden Kapiteln.



Wie man sieht besteht die Klasse Matrix aus der Basisklasse (in manchen Büchern auch Superklasse genannt) MatrixVector und der davon abgeleiteten (oder auch Subklasse) Matrix.

Dargestellt habe ich hier die Generalisierung. Wobei es praktisch keinen Unterschied macht ob ich von Generalisierung oder Spezialisierung spreche. Der Unterschied liegt in der Darstellungsweise. (mehr dazu im Buch UML 2 – Das umfassende Handbuch von Christoph Kecher).

Weiterhin ist Erkennbar, dass ich nicht nur Methode zur Funktionalität der Klasse MatrixVector in diese gepackt habe, sondern auch die arithmetischen Operatoren. Hier wäre ein Ansatz gegeben, ob man zur Übersichtlichkeit nicht noch eine Klasse, nur für die arithmetischen Operationen, dazwischen schiebt.

Wie ist die Klasse MatrixVector nun aufgebaut?

Zunächst das Gerüst der Klasse:

```
template<class T>
class MatrixVector : public std::vector<std::vector<T> >
{

public:

      typedef vector<vector<T> > base_type;
      typedef MatrixVector<T> my_type;
      typedef size_t size_type;


private:
      base_type elements;
      size_type rows;
      size_type columns;


public:

.
.
.

};
```

Die erste Zeile gibt an, dass es sich um eine template – Klasse handelt. Damit kann eine typenlose Klasse geschrieben werden.

```
template<class T>
```

Der Parameter class gibt an das es sich beim Typ der übergeben wird um eine Klasse handelt und T ist der Bezeichner der innerhalb der eigenen Klasse (hier MatrixVector) verwendet wird. Dieser ist überall da zu verwenden, wo eine Typangabe zu machen ist, welche auf die Elemente der Matrix verweisen.

In der zweiten klasse wird die Klasse definiert.

```
class MatrixVector : public std::vector<std::vector<T> >
{
```

class ist ein Bezeichner wie z.B. struct. Der Compiler weiß somit, dass das nachfolgende eine Klasse ist. MatrixVector ist der Bezeichner der Klasse. Mit diesem werden in einem Programm welches die klasse verwenden soll die Objekte erzeugt.

Die nachfolgende Anweisung

```
: public std::vector<std::vector<T> >
```

Gibt im Grunde die Datenstruktur an. Wenn man es genau nimmt, erbt die Klasse MatrixVector alle Eigenschaften und Methoden der Klasse vector in der Verschachtelung vector<vector<T> >.

Wie genau dies funktioniert muss ich ehrlich gesagt noch nachlesen, da ich dies aus dem Buch „Designing Components with the C++ STL“ von Ulrich Breymann (Seite 206) entnommen habe. Da ihr ja wisst, wie es um mein Englisch steht, könnt ihr euch ja denken, dass ich hier noch nicht wirklich viel gelesen habe.

Als nächstes folgt ein public – Bereich in dem ich drei Typen definiert habe.

```
public:

	typedef vector<vector<T> > base_type;
	typedef MatrixVector<T> my_type;
	typedef size_t size_type;
```

Der base_type ist der Basistyp der Matrix. my_type verweist auf die Klasse MatrixVector<T> selber. Der andere dient als Typ für die Größenangaben der Matrix, dazu mehr im Kapitel Konstruktoren / Destruktor.

Die Typdefinition dient dazu nur an einer Stelle den Datentyp ändern zu müssen und nicht in der gesammten klassen, falls dies mal notwendig werden sollte.

Das nächste Feld ist schon interessanter, da es die Attribute der Matrix enthält.

```
private:
	base_type elements;
	size_type rows;
	size_type columns;
```

Die Variable elements beinhaltet alle Werte der Matrix. Sie ist vom Type base_type, also vom Type vector<vector<T> >. Wie man die Daten ein- bzw. ausliest behandle ich im Kapitel Attribute und Methoden.

In den Variablen rows und columns wird die Anzahl der Zeilen und Spalten gespeichert. Dabei ist rows für die Zeilen und columns für die Spalten gedacht.

Diese Attribute sind private gekennzeichnet, damit Sie nicht von außen geändert werden können, ohne das die Klasse MatrixVector etwas davon mitbekommt. Hier könnte man darüber nachdenken, ob man Sie nicht protected macht. Protected bedeutet, dass abgeleitete Klassen lesend auf die Attribute zugreifen können. Dies ist im übrigen die erste Kapselung der Klasse, damit der Begriff auch mal fällt und auf dem Bewertungsbogen nicht im Raum steht. Also alles was ich vor dem Zugriff von außerhalb der Klasse schütze ist im Grunde eine Kapselung.

Soviel zum Kopf der Klasse MatrixVector. Was dann folgt sind die Methoden der Klasse. Diese werde ich aber später vorstellen.

2. Vererbung

Die Vererbung ist eigentlich mit einer Zeile abgehandelt. Ich zeige Sie mal anhand der Klasse Matrix.

```
template<class T>
class Matrix : public MatrixVector<T>
{
```

Wie man sieht, ist auch die Klasse Matrix eine template-Klasse. Würde man dies nicht machen, müsste man für die geerbte Klasse MatrixVector im Konstruktor einen Type angeben, denn mit der Angabe:

```
: public MatrixVector<T>
```

Sagen wir, dass die Klasse Matrix eine abgeleitete Klasse der MatrixVector ist. Somit erbt die Klasse Matrix alle Eigenschaften und Methoden (welche public sind) von der Klasse MatrixVector. Da MatrixVector aber einen Type angegeben haben möchte, müssen wir auch einen übergeben und daher ist die Klasse Matrix auch eine template-Klasse, denn wir (sollen) es ja demjenigen überlassen, der mit der Klasse arbeitet, von welchem Typ das entsprechende Object vom Typ Matrix sein soll.

Diese eine Zeile ist im Grunde alles, aber Achtung der Konstruktor muss auch noch angepasst werden, da schließlich Parameter übergeben werden sollen, wie groß die Matrix ist. Aber mehr davon im Kapitel Konstruktoren und Destruktor.

Der Rest der Klasse kann synonym zur Klasse MatrixVector geschrieben werden.

Zur Vererbung sei noch gesagt, dass man auch von mehreren Klassen etwas erben kann, oder auch andere Vererbungsstrukturen beiführen kann. Zu diesen Themen findet ihr im Buch C++ von A bis Z von Jürgen Wolf einiges. Siehe dazu:

Kapitel 5.2 Klassen-Templates (Seite 489 ff)
Kapitel 4.7 Vererbung (abgeleitete Klassen) (Seite 392 ff)

3. Konstruktoren / Destruktor

Zunächst einmal das Listing der Konstruktoren der Klasse MatrixVector:

```
MatrixVector(){resize(3, 3);}

MatrixVector(size_type dim) {resize(dim, dim);}

MatrixVector(size_type zeilen, size_type spalten)
{resize(zeilen, spalten);}
```

Die Konstruktoren stehen im public – Bereich und sind für die Klasse MatrixVector zwingend notwendig. Wie ihr seht habe ich drei Konstruktoren erstellt. Entsprechend der Aufgabenstellung:

„(...) Implementieren Sie das System so, dass Matrizen verschiedener Dimension, auch nichtquadratische behandelt werden können (...)
float f;
// 3x4 -, 4x2 -, 3x3 - Matrizen
Matrix <float > mf1 (3 ,4) , mf2 (4 ,2) , mf3;
(...)“

Ich habe dies also so interpretiert, dass es möglich sein soll keinen, einen bzw. zwei Parameter anzugeben, welche die Größe der Matrix bestimmen.

Aber was machen die Konstruktoren nun ?

Nehmen wir den ersten Konstruktor.

```
MatrixVector(){resize(3, 3);}
```

Dieser Konstruktor wird ausgeführt wenn der Klasse MatrixVector kein Parameter übergeben wird, wie beispielsweise aus folgender Programmzeile:

MatrixVector <double> m1;

Hier möchte der Programmierer ein Objekt m1 vom Type Matrix mit Standardgröße anlegen. Des Weiteren gibt er noch an, dass die Elemente der Matrix vom Type double sein sollen.

Was heißt nun Standardgröße ?

In unserem Fall habe ich die auf eine 3x3 Matrix festgelegt. Daher auch folgender Befehl, welcher der Konstruktor ausführt, bei der Initialisierung des Objektes m1.

```
{resize(3, 3);}
```

Dieser Befehl führt die Methode resize aus und übergibt für die Anzahl der Zeilen und Spalten den Wert 3. Die Methode resize sieht wie folgt aus:

```
void resize(size_type row, size_type column) {
      if ((row<1) || (column<1)) throw MatrixException(100);
      elements.resize(row, vector<T>(column));
      for(size_type r=0; r<row; ++r)
            elements[r].resize(column);
      columns=column;
      rows=row;
}
```

Sie ist im public – Bereich zu finden, damit der Programmier selber auch die Möglichkeit hat eine Matrix auch noch nach der Initialisierung in ihrer Größe zu verändern. Vorsicht ist dabei aber geboten, denn es gehen alle Element-Werte verloren.

Was macht die Methode resize nun im Einzelnen?

Der Methode sind als Parameter die Anzahl Zeilen und Spalten zu übergeben. Dabei initalisiert die Methode das Attribut element. Wir erinnern uns, dass im Attribut element die Werte der Matrix gespeichert sind.

Wie erfolgt die Initialisierung ?

Zunächst werden im Attribute element die Zeilen angelegt. Dies geschieht mithilfe der Zeile:

```
elements.resize(row, vector<T>(column));
```

Dabei greift die Methode resize widerum auf eine Methode resize zurück. Diese stammt aber von der Klasse vector. Diese Methode (vector.resize) legt row Zeilen vom Type vector<T> an, welche column Spalten enthält.

In der darauffolgenden Schleife werden die Zeilen initialisiert:.

```
for(size_type r=0; r<row; ++r)
                  elements[r].resize(column);
```

Während dieser Initialisierung werden die Werte gleich auf 0 gesetzt. Dabei ist die Null auch gleich eine vom entsprechenden Typ T.

Die letzten beiden Zeilen

```
columns=column;
rows=row;
```

übergeben den Attributen rows und columns die Anzahl der Zeilen und Spalten.

Aber was macht eigentlich folgende Zeile?

```
if ((row<1) || (column<1)) throw MatrixException(100);
```

Entschuldigung wenn ich mir hier mal einen Scherz erlaube, aber der Programmierer der die Klasse MatrixVector verwenden, will könnte ja ebenfalls durch die Prüfung in Algebra gefallen sein und eine Matrix der Größe (-3)x(-5) anlegen wollen oder aber auch 0x0.

Was sollen dies aber für Matrizen sein?

Die obige Zeile fängt solche Angaben also ab und stoppt das weitere ausführen der Methode resize und übergibt eine Exception der Klasse MatrixException mit dem Parameter 100, was in diesem Fall einfach nur eine Fehlernummer darstellt. Was da genau passiert, werde ich im Kapitel Fehlerbehandlung erläutern.

Somit ist das abarbeiten der Methode resize erledigt und der Konstruktor hat seine Schuldigkeit getan. Wir haben nun eine Objekt vom Typ MatrixVector der Größe 3x3 und die Elemente wurden entsprechend als double angelegt und auf den Wert 0.0 initialisiert.

Was machen aber nun die anderen zwei Konstruktoren ?

Schauen wir uns mal den zweiten Konstruktor an.

```
MatrixVector(size_type dim) {resize(dim, dim);}
```

Wie man sieht ist dieser um einen Parameter erweitert worden. Gibt man nun im Programm folgende Zeile ein

MatrixVector<double> m1(2);

wird eine Matrix vom Typ 2x2 angelegt, bei der die Elemente vom Typ double sind. Wird also ein Parameter angegeben so wird nicht der erste (Standardkonstruktor) ausgeführt, sondern dieser hier, der den Parameter dim (für dimension) an die Methode resize übergibt und somit eine quadratische Matrix vom Typ dim x dim angelegt wird.

Der dritte Konstruktor macht im Grunde das Gleiche, nur das hier ein weiterer Parameter dazukommt und man im Programm folgendes angeben kann:

MatrixVector<double> m1(3,2).

Was macht der Konstruktor nun?

Schauen wir uns auch hier die den Konstruktor noch einmal an.

```
MatrixVector(size_type zeilen, size_type spalten)
{resize(zeilen, spalten);}
```

Er übergibt also auch hier die Parameter der Methode resize so, dass eine Matrix vom Typ zeilen x spalten erzeugt wird, in der die Elemente vom Typ double sind.

Also alles keine wilde Sache.

Kommen wir zum Destruktor.

```
~MatrixVector() {}
```

Wie man sieht enthält dieser weder Paramter noch Code der ausgeführt werden soll, wenn der Destruktor aufgerufen wird.

Warum dies?

Ganz einfach, wir verwenden für unsere Daten den Typ vector. Dieser Datentyp verwaltet sich selbst und hat selber Konstruktoren und Destruktoren. Diese erledigen dass Speichermanagement und geben von sich aus den Speicher frei, welches das Attribut elements nutzt. Wird also ein Matrix-Objekt zerstört, wird nicht nur der Destruktor der Klasse MatrixVector, sondern auch der der Klasse vector aufgerufen. Es ist also nicht notwendig im Destruktor etwas freizugeben. Im Gegenteil würde man dies versuchen käme es im geringsten Fall zu einem Fehler im Programm im schlimmsten Fall zu einem Blue-Screen, da man am Speicher rumwerkelt.

4. Attribute / Methoden

Fangen wir mit den Attributen an. Da gibt es nichts weiter zu sagen, denn wir haben alle Attribute der Klasse MatrixVector und deren Funktion schon kennengelernt.

Sollte es da noch fragen geben, dann schreibt es mir per E-Mail (juergen@jd-n.de) oder quatscht mich im Hörsaal oder so an.

Zu den Methoden möchte ich auch nicht näher eingehen. Ich will einfach nur eine Liste mit dem Bezeichner und einer kurzen Beschreibung an dieser Stelle bringen. Es wäre vom Umfang zu groß um hier alle Details zu besprechen, was aber auch nicht notwendig ist.

Ich werde die Liste aufgrund meines Zeitmangels unsortiert hier reinstellen. Bitte entschuldigt dies. Ich kann im Moment auch nicht unbedingt garaniteiren das alle Methoden ohne Probleme funktionieren. Also gilt ebenfalls dafür, wenn es Probleme gibt kontaktiert mich bitte, denn ich bekomme auch nicht jeden Fehler mit.

Methode (als Prototyp)	Beschreibung
[size_t index]	Dies ist ein Indexoperator den ich so überladen habe, dass man wie folgt auf die Elemente zugreifen kann: elements [z][s] Wobei z für die Zeile und s für die Spalte steht und ein nummerischer Wert größer 0 und kleiner rows bzw. columns sein muss.
void resize(size_t zeilen, size_t spalten	Wurde in den vorangegangen Kapaitel behandelt.
size_t getRows()	Gibt als Type size_type die Anzahl der Zeilen der Matrix zurück.
size _t getColumns()	Gibt als Type size_type die Anzahl der Spalten der Matrix zurück.
size_t size()	Gibt als Type size_type das Produkt aus Zeilen und Spalten zurück.
void setRows(size_t zeilen)	Verändert die Anzahl der Zeilen einer Matrix. Die Anzahl der Spalten bleibt erhalten. Vorsicht die Werte der Matrix gehen verloren, da intern die Methode resize aufgerufen wird.
void setColumns(size_t spalten)	Verändert die Anzahl der Spalten einer Matrix. Die Anzahl der Zeilen bleibt erhalten. Vorsicht die Werte der Matrix gehen verloren, da intern die Methode resize aufgerufen wird.
<<	Überladener shift-Operator <<. Wird beispielsweise in Zusammenhang mit cout aufgerufen (cout << m1;).
>>	Überladener shift-Operator >>. Dient der Eingabe der einzelnen Werte einer Matrix. Beispielsweise mithilfe von cin (cin >> m1;).

()	Überladener Index-Operator (). Dient dazu Einzelwerte aus der Matrix auszulesen. Zum Beispiel wie folgt. MatrixVector<double> m1; double zahl; zahl = m1(1,2); Hier wird der Einzelwert in Zeile 2 und Spalte 3 an die Variable zahl übergeben. Nicht zu vergessen, die Zählweise beginnt bei Null.
bool isZero()	Gibt true zurück wenn es sich um eine Nullmatrix handelt. Ansonnsten wird false zurückgegeben.
bool isSquare()	Gibt true zurück wenn es sich um eine quadratische Matrix handelt. Ansonnsten wird false zurückgegeben.
bool isDiagonal()	Gibt true zurück wenn es sich um eine Diagonalmatrix handelt. Ansonnsten wird false zurückgegeben.
bool isScalar()	Gibt true zurück wenn es sich um eine Skalarmatrix handelt. Ansonnsten wird false zurückgegeben.
bool isUnit()	Gibt true zurück wenn es sich um eine Einheitsmatrix handelt. Ansonnsten wird false zurückgegeben.
bool isUpper()	Gibt true zurück wenn es sich um eine rechte/obere Dreiecksmatrix handelt. Ansonnsten wird false zurückgegeben.
bool isLower()	Gibt true zurück wenn es sich um eine linke/untere Dreiecksmatrix handelt. Ansonnsten wird false zurückgegeben.
short isTriangular	Untersucht die Matrix auf Dreiecksmatrix und gibt einen entsprechenden Zahlenwert zurück: 0 – keine Dreiecksmatrix 1 – rechte/obere Dreiecksmatrix

	2 – linke/untere Dreiecksmatrix 3 – wenn nicht unterschieden werden kann, wie bei einer Diagonalmatrix. Könnte aber auch eine Einheitsmatrix sein, denn diese ist eine spezielle Diagonalmatrix, wie auch die Nullmatrix.
==	Vergleichsoperator ==. Gibt true zurück wenn die Matrix 1 und 2 die gleiche Zeilenzahl, Spaltenzahl aufweist und wenn die einzelnen Elemente an densleben Positionen den gleichen Wert aufweisen. Ansonsten wird false zurückgegeben.
!=	Vergleichsoperator !=. Hier gilt das Gleiche wie für den zuvor erwähnten Vergleichsoperator ==. Nur das statt true false und statt false true zurückgegeben wird.
void setElement(size_t row, size_t column, T val)	Diese Methode dient dazu einzelne Werte der Matrix zu überschreiben, also mit neuen Werten zu füllen. Die Parameter haben dabei folgende Bedeutung: row - Zeile column – Spalte val – Der zu an die Position (row,column) zu schreibende Wert. Dieser muss vom gleichen Typ wie der für die Matrix vorgegebene sein, daher als Typ T.
bool isColumnVector()	Gibt true zurück wenn es sich um einen Spaltenvektor handelt. Ansonnsten wird false zurückgegeben. Diese Methode dürfte sehr hilfreich sein, bei der Unterscheidung ob mit dem Operator * eine Matrixmultiplikation oder das Vektorprodukt (Kreuzprodukt) gemeint ist.
base_type getElements()	Gibt alle Elemente der Matrix als Type base_type zurück. Wobei base_type im Moment mit vector<vector>T> > definiert ist.
=	Zuweisungsoperator =. Dieser Operator dient dazu um in einem Programm folgendes möglich zu machen: MatrixVector<double> m1; MatrixVector<double> m2; cin >> m1; **m2 = m1;**

5. Fehlerbehandlung

Dieses Kapitel möchte ich mal von der praktischen Seite angehen. Schaut euch mal folgende Programmzeilen an.

MatrixVector<double> m1(3,3);
MatrixVector<double> m2(2,4);

m1 = m1 + m2;

Was wird hier passieren, wenn der Operator + aus folgenden Programmzeilen besteht:

```
my_type operator +(my_type &m)
{
	my_type tmp(rows,columns);
	for (unsigned r=0; r<rows; r++)
		for (unsigned c=0; c<columns; c++)
			tmp.elements[r][c] = (elements[r][c] + m.elements[r][c]);
	return tmp;
}
```

Das Problem entsteht, wenn r den Wert 2 annimmt, denn die Matrix m2 hat keine 3.Zeile. Es kommt also zum Programmabsturz.

Das problem ist die Addition selber, denn diese ist so definiert, das Elementweise addiert wird und um Diskussion wie Letztens zu vermeiden möchte ich den Papula im Band 2 auf Seite 12 in seiner Anmerkung zitieren:

„(…) Addition und Subtraktion sind nur für Matrizen gleichen Typs erklärt.(…)“

Was heißt dies für uns?

Wir müssen solche Fehler, welcher ein Programmierer machen könnte, der unsere Klasse verwendet, abfangen.

Dies kann man mithilfe von Exceptions machen. Ich ergänze dazu die Funktion zum überladen des Operators + wie folgt:

```
my_type operator +(my_type &m) {
	if (!isSameType((*this), m)) throw MatrixException(105);
	my_type tmp(rows,columns);
	for (unsigned r=0; r<rows; r++)
		for (unsigned c=0; c<columns; c++)
			tmp.elements[r][c] = (elements[r][c] + m.elements[r][c]);
	return tmp;
}
```

Was geschieht in der gelb unterlegten Zeile?

Zunächst einmal wird über eine Methode isSameType geprüft ob beide Marizen vom selben Typ sind. Ist dies der Fall wird mit der Ausführung der Methode operator + fortgefahren. Ist dies nicht der Fall, wird die Ausführung abgebrochen und mithilfe des Befehls „throw“ eine Exception der Klasse MatrixException mit dem Parameter 105 ausgelöst.

Was bedeutet dies?

Auch dies möchte ich anhand eines Programmcodes erläutern.

```
try {
      m1 = m1 + m2;
}
catch (MatrixException err) {
      err.showMsg();
}
```

Es wird hier das Gleiche gemacht wie im Eingangsseitigen Programmbeispiel. Abgesehen davon, dass sich um Fehler berücksichtigt sind.

Zuerst wird mithilfe der Anweisung try versucht den im try-Block liegenden Code auszuführen. Tritt dabei ein Fehler auf wird nicht das Programm beendet, sondern die Ausführung des Codes innerhalb des try-Blocks.

Wir haben vorher gesehen, dass mithilfe der Zeile

```
if (!isSameType((*this), m)) throw MatrixException(105);
```

eine Exception ausgelöst wird. Diese kann nun mithilfe der catch-Anweisung abgefragt werden.

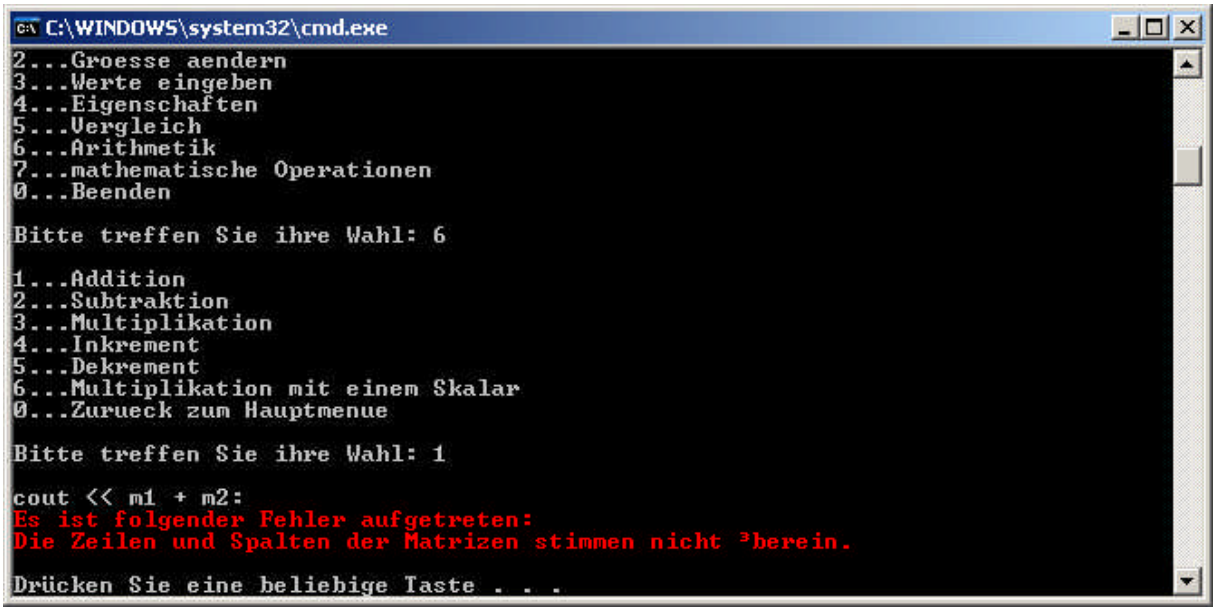
```
catch (MatrixException err) {
```

Dabei wird in meiner Zeile eine Variable err angelegt, welche vom Typ Matrixexception ist und beinhaltet nur den Fehler der im jeweiligen vorangegangen try-Block aufgetreten ist. Der Code innerhalb des catch-Blocks wird auch nur ausgeführt, wenn in dem vorangegangenen try-Block ein Fehler aufgetreten ist. Bei wird folgendes ausgeführt.

```
err.showMsg();
```

Es wird also die Memberfunktion showMsg der Klasse MatrixException aufgerufen. Diese macht nichts anderes als zum entsprechenden Fehler eine Fehlermeldung auszugeben.

Somit wird ein entsprechender Fehler in einem Programm wie folgt abgefangen:



Was steckt nun hinter der Klasse MatrixException?

Lediglich folgender Code, welcher recht simpel ist:

```
#include <iostream>
#include <windows.h>
#include <string>



#define msg0      "Kein Fehler.";
#define msg100    "Die Angabe der Zeilenanzahl und/oder Spaltenanzahl ist
kleiner als 1."
#define     msg101      "Sie müssen eine Anzahl Zeilen größer 0 eingeben."
#define msg102    "Sie müssen eine Anzahl Spalten größer 0 eingeben."
#define msg103    "Zeilen- und/oder Spaltenangabe lag außerhalb des
Definitionsbereiches der Matrix."
#define msg104    "Die Position an die der Wert geschrieben werden soll,
liegt außerhalb des Definitionsbereiches der Matrix."
#define msg105    "Die Zeilen und Spalten der Matrizen stimmen nicht
überein."
#define msg106    "Spalten der Matrix A und Zeilen der Matrix B stimmen
nicht überein."



class MatrixException {

private:
      int err;

      void textcolor(unsigned short color = 15) {
            SetConsoleTextAttribute(::GetStdHandle(STD_OUTPUT_HANDLE),
color);
      }

public:

      // Konstruktor/Destruktor
      MatrixException(int i) : err(i) {}
      ~MatrixException () {}

      //    Gibt Fehlernummer zurück
      int getErrorNumber() {return err;}

      //    Gibt die Fehlermeldung zurück;
      const char* getErrorMsg(){
            switch(err){
                  case 100    :     return msg100; break;
                  case 101    :     return msg101; break;
                  case 102    :     return msg102; break;
                  case 103    :     return msg103; break;
                  case 104    :     return msg104; break;
                  case 105    :     return msg105; break;
                  case 106    :     return msg106; break;
                  default           :     return msg0; break;
            }
      }

      // Gibt die Fehlermeldung aus.
      void showMsg(){
            textcolor(12);
```

```
            std::cerr << "Es ist folgender Fehler aufgetreten: " <<
std::endl;
            std::cerr << getErrorMsg() << std::endl << std::endl;
            textcolor(7);
      }
};
```

Man muss es nicht so machen. Es ist nur ein Vorschlag von mir.

Mehr kann man dazu im Buch C++ von A bis Z von Jürgen Wolf im Kapitel 6 auf den Seiten 661 ff lesen. Wenn es nur um die Klassenspezifische Exception geht, der kann dies im Kapitel 6.4.1 auf den Seiten 678 ff nachlesen.

Ich hoffe ich konnte ein wenig Licht ins Dunkel der Klassen bringen, vor allem fürs Projekt.

Jürgen