Hochschule für
Wirtschaft und Recht Berlin
Berlin School of Economics and Law

Studienarbeit II

Erstellung einer C++ Bibliothek für Matrizen-Berechnungen mittels Taylor Polynomen

Name, Vorname:	Berth, Michael	Ritter, Marvin
Matrikelnummer:	603699	620373
Ausbildungsbetrieb:	Swisslab GmbH	Helmholtz-Zentrum Berlin
Studienjahrgang:	2010	
Fachbereich:	Duales Studium Wirtschaft • Technik	
Studiengang:	Informatik	
Modul:	IT3201 – Studienprojekt II	
Betreuer Hochschule:	Prof. Dr. Dagmar Monett Díaz	Wissenschaftliche Betreuung
Betreuer Unternehmen:	Olaf-Peter Sauer	
Anzahl der Wörter:	6152	

Vom Ausbildungsleiter zur Kenntnis genommen:

..... (Datum/Unterschrift)

..... (Datum/Unterschrift der/des Studierenden)

Kurzfassung

Dieses Dokument ist eine technische Dokumentation zur Erstellung einer C++ - Klassenbibliothek aus vorhandenem Quellcode. Weiterhin beschreibt sie den Vorgang des Portierens in andere Programmiersprachen und geht dabei vor allem auf die Entwicklungsmethodik des Test Driven Developments ein.
In der Dokumentation sind alle wesentlichen Schritte von der Analysephase, über das Zeitmanagement, bis hin zur Implementation und Erstellung von Beispielanwendungen erläutert.

Summary

This document is a technical documentation for the creation of a C++ - class library from existing source code. Furthermore the documentation describes the process of porting the class library to other programming languages and the process of the development method of test driven development.
In this documentation all necessary steps, from the analysis phase and time management until the implementation and creation of example applications will be explained.

Inhaltsverzeichnis

Abbildungsverzeichnis

Abkürzungsverzeichnis

Kurzform	Definition
DLL	Dynamic Link Library
FFI	Foreign Function Interface
LIB	Library
SWIG	Simplified Wrapper and Interface Generator
TDD	Test Driven Development

1 Einleitung

1.1 Allgemeine Hinweise zur Studienarbeit

Um ein besseres Verständnis und eine vereinfachte Darstellung zu ermöglichen, wurden folgende Formulierungen in der Studienarbeit genutzt:

- *Kursiv* geschriebene Begriffe werden im Glossar erklärt, die Fachbegriffe werden nur beim ersten Auftreten markiert
- (Runde Klammern) stehen für Abkürzungen des jeweiligen Fachbegriffes
- **[Eckige Klammern]** enthalten Verweise auf den Anhang
- Alle Abbildungen, bei denen keine Quellen angegeben sind, sind eigene Darstellungen

1.2 Ausgangssituation

Als Ausgangssituation für dieses Studienprojekt wurde durch Fr. Prof. Dr. Dagmar Monett Díaz ein Programm zur Verfügung gestellt („daeIndexDat“), welches unter anderem die Komponenten beinhaltet, die in einer eigenen *Klassenbibliothek* erstellt werden sollen. Dieses Programm wurde nicht in kompilierter Form, sondern als Quelltextdateien der Sprache C++ zur Verfügung gestellt. Die zu übernehmenden Klassen sind innerhalb dieses Projektes in eigenen Klassendateien ausgelagert und die Dateistruktur ist übersichtlich gehalten.

1.3 Aufgabenstellung

Ziel des Studienprojektes ist es, aus dem gegebenen Quellcode eine Klassenbibliothek in der Programmiersprache C++ zu erstellen. Vorrangig sollen die Klassen „Matrix“ und „TPolyn“ in die Klassenbibliothek übernommen werden. Weiterhin bietet die *Portierung* eine Möglichkeit, ein *Refactoring* durchzuführen.

Im Anschluss an die Erstellung der Klassenbibliothek in C++ soll von jedem Studenten eine Portierung in einer selbst gewählten Sprache erstellt werden. Deshalb wird die Klassenbibliothek zusätzlich in die Sprachen C# und Python portiert. Im Abschluss ist die Funktionalität der jeweiligen Klassenbibliotheken, anhand einer Beispielanwendung für jede Portierung, nachzuweisen und der gesamte Fortschritt und Zustand des Projektes in einer Dokumentation zu belegen.

2 Projektmanagement

2.1 Konzeptionierung

Aus der Aufgabenstellung ist als erstes ein Konzept zu entwickeln, welches den Ablauf und die benötigten Teilprojekte beschreibt. Dieses Studienprojekt wird in die folgenden Teilprojekte gegliedert: Vorbereitung, Klassenbibliothek in C++, Portierung der Klassenbibliothek nach C#, Portierung der Klassenbibliothek nach Python, Erstellung von Beispielanwendungen und Dokumentation des Studienprojektes.

2.1.1 Vorbereitung

Um die Zielsetzung optimal umsetzen zu können ist eine gründliche Vorbereitung notwendig. Daher wird zu Beginn der Umsetzung eine Analysephase durchgeführt. Die Analysephase umfasst die Erstellung einer Zielsetzung, die Einarbeitung in die Themen „Matrizen“ und „Taylorpolynome“, die Analyse des vorhandenen Quellcodes, die Kompilierung des gegebenen Programms und die Erstellung eines Zeitplanes, der die *Milestones* definiert. Weiterhin werden in dieser Phase *Coding Conventions* eingeführt, an die sich während der Entwicklung gehalten wird.

2.1.2 Klassenbibliothek in C++

Die Klassenbibliothek in C++ zu erstellen ist die Hauptaufgabe in diesem Projekt. Die Klassenbibliothek soll die Klassen „Matrix“ und „TPolyn“ aus dem gegebenen Projekt beinhalten und refactored werden.

Im Zuge der Vorbereitung wurde ein Aspekt der Entwicklung besonders berücksichtig, nämlich die Nutzung des „Test Driven Developments“ (TDD), die zu höherer Qualität beiträgt und auch sicherstellt, dass die Funktionalitäten korrekt sind.

Die Bereitstellung der Klassenbibliothek soll als windowskompatible „Dynamic Link Library“ (DLL) erfolgen. Für die Verwendung der DLL muss weiterhin beim kompilieren der Klassenbibliothek auch eine .LIB-Datei miterstellt werden, um die Nutzung der Klassenbibliothek in anderen Projekten zu ermöglichen.

2.1.3 Portierung der Klassenbibliothek nach C#

Nach Abschluss der Programmierarbeiten der C++ - Klassenbibliothek ist vom Studenten Michael Berth eigenständig eine Portierung der Klassenbibliothek in die Programmiersprache C# zu erfolgen.
Diese Klassenbibliothek soll nach Möglichkeit die gleichen Funktionalitäten der C++ Variante beinhalten und auch nach den Regeln des „Test Driven Developments“ entwickelt werden. Der Nachweis der Funktionalität ist ebenfalls zu erbringen.

2.1.4 Portierung der Klassenbibliothek nach Python

Nach Abschluss der Programmierarbeiten der C++ - Klassenbibliothek ist vom Studenten Marvin Ritter eigenständig eine Portierung der Klassenbibliothek in die Programmiersprache Python zu erfolgen.
Diese Klassenbibliothek soll nach Möglichkeit die gleichen Funktionalitäten der C++ Variante beinhalten und auch nach den Regeln des „Test Driven Developments“ entwickelt werden. Der Nachweis der Funktionalität ist ebenfalls zu erbringen.

2.1.5 Erstellung von Beispielanwendungen

Im Anschluss an die Erstellung der Klassenbibliotheken muss deren Funktionalität nachgewiesen werden, sowie eine beispielhafte Implementation der Bibliothek durchgeführt werden. So ist angedacht, dass es für jede Portierung der Klassenbibliothek, eine eigene Anwendung geben wird, die die Bibliothek nutzt und die Funktionalitäten nachweist. Die Anwendungen sind jeweils in der gleichen Programmiersprache geschrieben, wie die jeweilige Portierung.

2.1.6 Dokumentation des Studienprojektes

Zum Abschluss des Projektes und nachdem alle Arbeiten durchgeführt sind, ist eine Dokumentation zu erstellen, die den Fortschritt, die Bearbeitungsschritte und den Zustand des Projektes darstellt. Weiterhin sind in der Dokumentation Probleme und Techniken, die während des Projektes aufgetreten oder genutzt worden sind, zu beschreiben.

2.2 Zeitplan des Projektes

Die nachfolgende Auflistung stellt die zeitliche Planung und die Umsetzungsphasen des Studienprojektes SPI IT 3201 dar. Das Projekt wurde in mehrere Phasen unterteilt und folgendermaßen strukturiert:

- Phase 1 15.10.2012 – 04.11.2012
  Einarbeitung in die Themen „Matrizen“ und „Taylorpolynome“
  Analyse des vorhandenen Quellcodes
  Entwicklung von Coding Conventions

- Phase 2 03.11.2012 – 23.12.2012
  Erstellung der Klassenbibliothek in C++
  Einhaltung der getroffenen Coding Conventions

- Phase 3 07.01.2013 – 03.02.2012
  Erstellung der Portierungen
  C# - Version der Klassenbibliothek
  Python - Version der Klassenbibliothek

- Phase 4 04.02.2012 – 03.03.2013
  Dokumentation des Projektes
  Abschließende Arbeiten

- Abgabetermin 04.03.2012
  Termin für die Abgabe des Projektes im Hochschulbüro

Projektbegleitend wurde eine Präsentation zur Studienarbeit erstellt, welche am 14.12.2012 vor dem Kurs und dem Auftraggeber, in Person von Frau Prof. Dr. Monett, abgehalten wurde.

3 Mathematische Erläuterungen

Dieser Abschnitt gibt einen kurzen Einblick in die für das Projekt notwendigen mathematischen Grundlagen. Das Verständnis wie mit Matrizen und Taylor Polynomen gerechnet wird war essentiell für die Entwicklung der Klassenbibliothek.

3.1 Matrix

Als Schlüsselelement aus der Linearen Algebra taucht die Matrix in fast allen Gebieten der Mathematik und somit auch in vielen Bereichen der Informatik, z. B. 3D Grafiken und Künstliche Intelligenz, auf. Dabei handelt es sich meist um Matrizen von reellen Zahlen. Natürlich lassen sich auch mit anderen mathematischen Objekten, wie komplexe Zahlen oder Polynome, Matrizen bilden.
Allen gemeinsam ist eine rechteckige, durch Zeilen und Spalten gegliederte Anordnung der Elemente.

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \dots & a_{2n} \\ a_{31} & a_{32} & a_{33} & \dots & a_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & a_{m3} & \dots & a_{mn} \end{pmatrix}$$

Abbildung 3-1: Darstellung einer Matrix

3.2 Taylorpolynom

Taylorpolynome sind Potenzreihen zur Annährung von Funktionen in einer festgelegten Umgebung. Innerhalb dieser Umgebung ist die Annährung sehr gut, sodass mit dem Taylorpolynom gerechnet werden kann. Dies ist oft einfacher und schneller als die Benutzung der Originalfunktion.

Die unendliche Taylorreihe ist dabei wie folgt definiert:

$$P_f(a) = f(a) + \frac{f'(a)}{1!}(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \ldots + \frac{f^{(n)}(a)}{n!}(x-a)^n$$
$$= \sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!}(x-a)^n$$

Abbildung 3-2: Taylorreihe

Ein Taylorpolynom ist eine nach n Schritten abgebrochene Taylorreihe für eine Stelle a. Je größer n ist, desto genauer wird die Annährung wie das Beispiel für die Wurzelfunktion an der Stelle 4 zeigt.

$$f(x) = \sqrt{x} = x^{\frac{1}{2}}$$
$$T_1(4) = 2 + \frac{1}{4}\frac{(x-4)}{1!}$$
$$T_2(4) = 2 + \frac{1}{4}\frac{(x-4)}{1!} - \frac{1}{32}\frac{(x-4)^2}{2!}$$
$$T_3(4) = 2 + \frac{1}{4}\frac{(x-4)}{1!} - \frac{1}{32}\frac{(x-4)^2}{2!} + \frac{3}{256}\frac{(x-4)^3}{3!}$$

Abbildung 3-3: Taylorpolynom

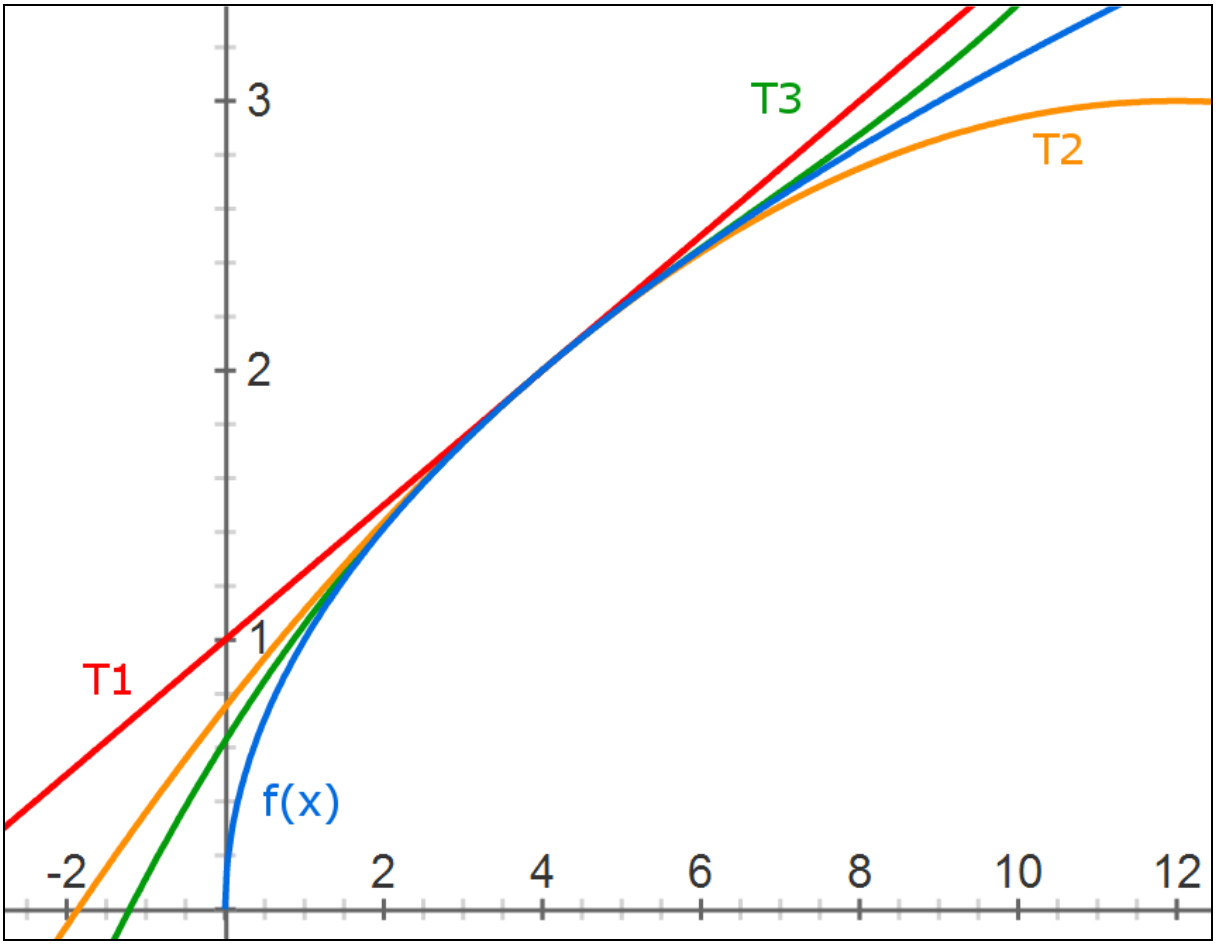


Abbildung 3-4: Wurzel-Taylorpolynom

4 Test Driven Development

4.1 Allgemeine Beschreibung

In den Coding Conventions wurde beschlossen, dass die Entwicklung dieses Projektes den Richtlinien der testgetriebenen Entwicklung („Test Driven Development“) folgt.

Bei dieser Form der Softwareentwicklung werden vor der Entwicklung einer Funktionalität, ein oder mehrere Teste geschrieben, die sicherstellen sollen, dass die Funktionalität gegeben ist. In den Softwaretests legt man *Assertions* (Annahmen) fest, die je nach Annahme und Test übereinstimmen müssen.

Die Softwaretests werden bei jedem Kompiliervorgang ausgeführt und sollten nach dem Schreiben fehlschlagen, da die Funktionalität noch nicht implementiert ist. Ist der Softwaretest fertig, folgt der Beginn der Implementierung der Funktionalität. Ist dieser Vorgang fertiggestellt kann neu kompiliert werden. Sollte der Softwaretest weiterhin fehlschlagen, gibt es dafür zwei Gründe. Entweder ist der Test falsch geschrieben oder die Funktionalität ist nicht korrekt implementiert. Hierbei muss die Funktionalität nun solange korrigiert werden, bis der Softwaretest erfolgreich ist. Als nächstes kann die Implementierung weiterer Funktionalitäten entsprechend diesem Modell erfolgen.

4.2 GTest Framework

Im Rahmen der Analysephase wurde beschlossen, dass das „Google C++ Testing Framework“ für die C++ Implementierung der Klassenbibliothek genutzt wird. Die Gründe hierfür sind die relativ einfache Integration und die Plattformunabhängigkeit des Frameworks. Zudem gibt es schon viele vorgefertigte Assertions, wie zum Beispiel „ASSERT_TRUE(value)“, in der angenommen wird, dass der Wert „value“ „true“ ist. Wird der Softwaretest durchgeführt und „value“ ist ungleich „true“, so führt dies zum Fehlschlag des Tests, wobei man bei C++ aufpassen muss, da ein boolscher Datentyp ein Integer-Datentyp ist und jeder Wert ungleich „0“ „true“ ergibt. Weitere häufig verwendete Assertions prüfen zum Beispiel auf Gleichheit oder Ungleichheit von Werten oder Objekten oder auch darauf, dass manche Funktionen eine *Exception* auslösen (vgl. http://code.google.com/p/googletest/).

4.3 Beispielhafte Implementation eines Tests

An dieser Stelle wird beispielhaft ein implementierter Softwaretest demonstriert, der für die Vergleichsoperatoren der Klasse „Polynomial“ geschrieben ist.

```
156 TEST_F(PolynomialOperator, comparsion)
157 {
158     // A
159     A = Polynomial(1);
160     double a[] = { 1, 2 };
161     A.setCoeffs(a);
162     // B
163     B = Polynomial(1);
164     double b[] = { 5, 2 };
165     B.setCoeffs(b);
166
167     ASSERT_TRUE(A == A);
168     ASSERT_FALSE(A == B);
169
170     ASSERT_FALSE(A != A);
171     ASSERT_TRUE(A != B);
172
173     Polynomial asA(A);
174     ASSERT_TRUE(A == asA);
175
176     Polynomial biggerA(2);
177     for (int i = 0; i < A.ncoeff(); i++)
178     {
179         biggerA[i] = A[i];
180     }
181     ASSERT_FALSE(A == biggerA);
182
183     ASSERT_EQ(A, asA);
184     ASSERT_NE(A, B);
185
186     ASSERT_TRUE( A < B );
187     ASSERT_FALSE( A < asA );
188     ASSERT_TRUE( A <= asA );
189     ASSERT_TRUE( B > A );
190     ASSERT_FALSE( asA > A );
191     ASSERT_TRUE( asA >= A );
192 }
```

Abbildung 4-1: GTest Beispiel – Vergleichsoperatoren

In Abbildung 4-1 ist der Test „comparison“ aus der Testgruppe „PolynomialOperator“ abgebildet. Dieser Test soll die Vergleichsoperatoren und den „Copy-Constructor“ der Klasse „Polynomial“ testen. In den Zeilen 158 - 165 werden die Polynome A und

B initialisiert und mit Werten vorbelegt. Im anschließenden Teil von Zeile 167 bis Zeile 171 werden die Assertions durchgeführt, die angenommen werden. So wird zum Beispiel in Zeile 167 angenommen, dass der Ausdruck „A == A“ den Wert „true“ annimmt. In den Zeilen 173 und 174 wird als erstes das Polynom „asA“ initialisiert, durch den „Copy-Constructor“ mit Übergabe des Polynoms „A“, und dann geprüft, ob der Ausdruck „A == asA“ den Wert „true“ ergibt. Es wird also davon ausgegangen, dass nach erfolgter Initialisierung durch den „Copy-Constructor“ beide Polynome gleich sind. Dies wird ebenfalls noch einmal in Zeile 183 mit dem Ausdruck „ASSERT_EQ(A, asA);“ überprüft, da die Funktion „ASSERT_EQ“ die Objekte „A“ und „asA“ ebenfalls auf Gleichheit prüft. In Zeile 184 wird geprüft, ob die Objekte „A“ und „B“ ungleich („ASSERT_NE“ entspricht dem Ausdruck „Assert Not Equal“) sind.

4.4 GTest Ausgabe

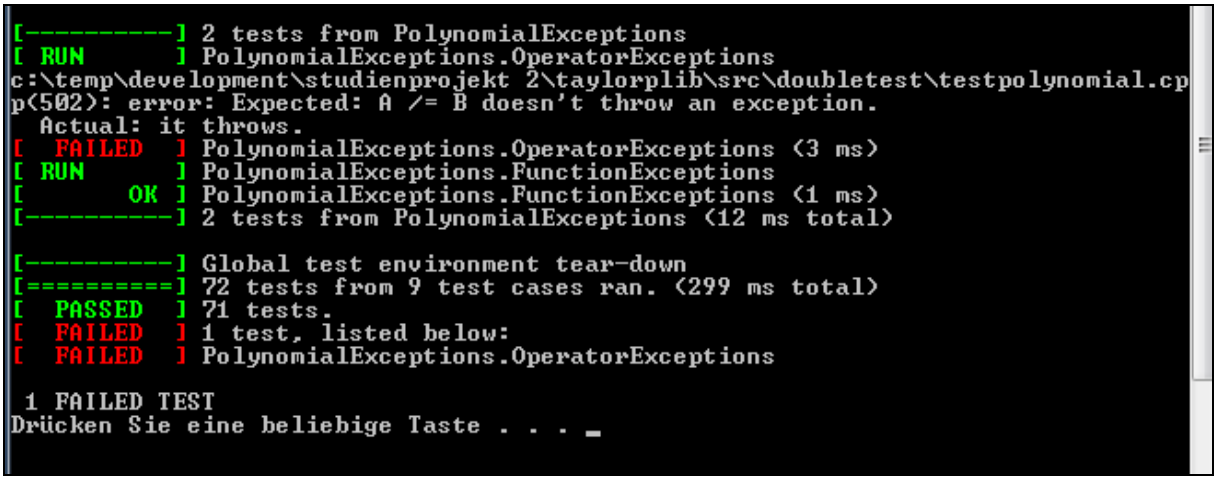


Abbildung 4-2: GTest Beispiel – Test Failed

In Abbildung 4-2 wird eine GTest Ausgabe nach einem Kompiliervorgang dargestellt, in dem ein Test fehlgeschlagen ist. In diesem Beispiel ist zu erkennen, dass in der Testgruppe „PolynomialExceptions“ im Test „OperatorExceptions“ der Test nicht erfolgreich durchgeführt werden konnte. Zudem gibt es noch weitere Informationen, an welcher Stelle der Test fehlgeschlagen ist. In diesem Beispiel wurde im Test erwartet, dass der Ausdruck „A /= B“ keine Exception auslöst und dass in diesem Test jedoch eine Exception ausgelöst wurde. So musste man die Funktionalität an dieser Stelle überarbeiten und neu kompilieren.

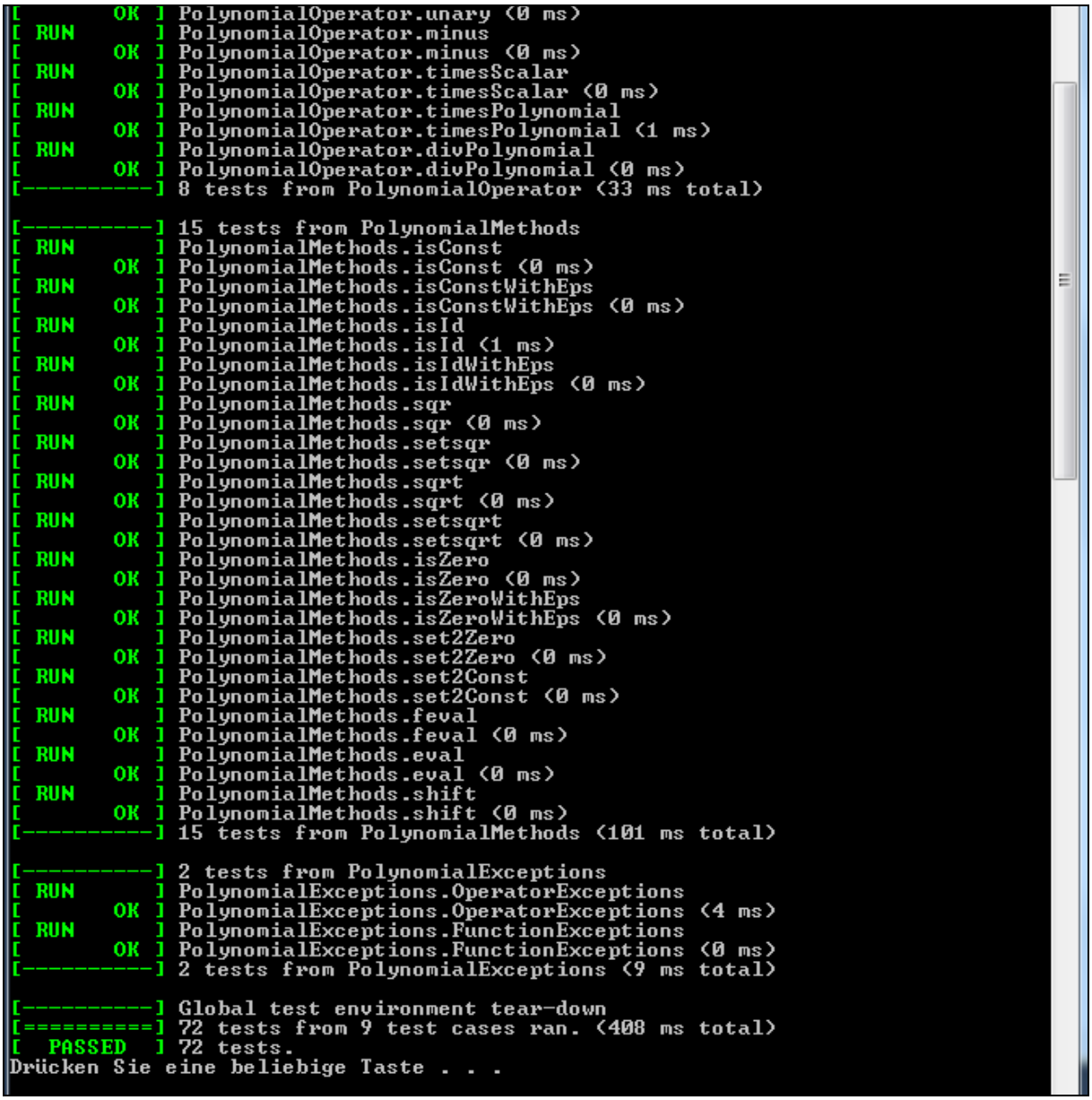


Abbildung 4-3: GTest Beispiel – Test Passed

In Abbildung 4-3 wird eine GTest Ausgabe nach einem Kompiliervorgang dargestellt, in dem alle Softwareteste erfolgreich durchgeführt sind. In der drittletzten Zeile kann man erkennen, dass 9 Testgruppen mit insgesamt 72 Testfällen durchgeführt worden sind. Die Gesamtzeit für die Durchführung betrug 408 ms.

Dies ist die erwartete Ausgabe, wenn die Softwareteste und Funktionalitäten korrekt implementiert sind.

5 Erstellung der Klassenbibliothek in C++

In diesem Abschnitt wird beschrieben, wie die Erstellung der Klassenbibliothek in C++ erfolgt ist. In der Analysephase wurde der gegebene Quellcode analysiert und die vorrangig benötigten Funktionen herausgesucht.

Die Implementationsphase wurde in vier Schritte eingeteilt. Für die erste Implementierung wurde die Umsetzung der Klasse Matrix zuerst für double Werte realisiert. Daraus ergeben sich die Phasen: Erstellung der Teste für Matrizen mit double Werten, Erstellung der Klassen „Matrix“, „Polynomial“ und „MathException“, Anpassung der Teste für Matrizen mit Polynomen und Anpassung der Klasse Matrix für Polynome. Diese Einteilung wurde getroffen, da es sich bei der Erstellung von Matrizen und den spezifischen Rechenoperationen relativ einfach ist, die Operationen per Hand durchzurechnen.

Bei der Erstellung der Klassen wurde der Quellcode des gegebenen Programms wiederverwendet und in einigen Fällen, in denen es sich anbot, der Code refactored.

Die Klassendiagramme sind im Anhang **[Anhang 1: Klassendiagramm Matrix]**, **[Anhang 2: Klassendiagramm Polynomial]** und **[Anhang 3: Klassendiagramm MathException]** zu finden und geben einen Überblick über die Eigenschaften und Methoden der Klassen.

Im nächsten Schritt kann nun die jeweilige Portierung in die Programmiersprachen C# und Python erfolgen.

6 Portierung der C++ Bibliothek nach C#

In diesem Abschnitt wird beschrieben, wie die Portierung der Klassenbibliothek nach C# erfolgt ist. Da die Sprachen C++ und C# sehr ähnlich sind, und sich C# aus C++ entwickelt hat, war ein großer Teil der Portierung durch einfaches Kopieren und Einfügen erledigt. Im Anschluss mussten nur die Syntax entsprechend angepasst werden. So wurde zum Beispiel aus:

```
_data[i][j] = A._data[i][j];
```

Abbildung 6-1: Code Snippet der C++ - Version

anschließend:

```
_data[i, j] = A._data[i, j];
```

Abbildung 6-2: Code Snippet der C# - Version

Weiterhin benötigt man bei der Programmiersprache C# zum Beispiel nur den „+ - Operator“, der dann auch gleichzeitig den „+= - Operator“ darstellt. Dies gilt analog zu dem „- - Operator“, dem „* - Operator“ und dem „/ - Operator“.
Im Vorfeld der Portierung wurde entschieden, dass Test Driven Development auch in der C# - Version der Klassenbibliothek anzuwenden. So ist bei der Portierung zuerst jede Funktion nach folgendem Beispiel implementiert worden:

```
public void function ()
{
   throw new NotImplementedException();
}
```

Abbildung 6-3: Anlegen einer neuen Funktion

Im nächsten Schritt wurden die Teste nach folgendem Muster angelegt:

```
[TestMethod]
public void FunctionTest_Function()
{
    Assert.Fail();
}
```

Abbildung 6-4: Anlegen eines neuen Testes

Dies führte zu insgesamt 72 Testen, die zu Beginn der Entwicklung alle fehlschlugen.

Im nächsten Schritt wurden nun die gesamten Teste in der C# - Version implementiert. Diese mussten alle neu geschrieben werden, da in der C# - Version das integrierte Testing Framework genutzt wurde. So musste jeder Test adaptiert und umgeschrieben werden.

Nach dem die Vorbereitungen durchgeführt sind schließt sich die eigentliche Portierung an. Die Portierung der Bibliothek lässt sich zu einem großen Teil durch Kopieren und Einfügen durchführen. Anschließend werden noch die entsprechenden Syntaxänderungen durchgeführt und circa 80 Prozent der Bibliothek sind erstellt. An manchen Stellen mussten jedoch strukturelle Anpassungen vorgenommen werden, da ein Teil der Logik aus der C++ Bibliothek nicht eins-zu-eins umzusetzen waren. Dies liegt darin begründet, dass im C++ Code vorrangig mit Pointern gearbeitet wird, dies in der C# - Variante jedoch nicht geht.

So wurde nun jede Funktion implementiert bis der erforderliche Test erfolgreich abgeschlossen wurde. Nach Abschluss der Portierungsarbeiten wurde im Visual Studio eine Codeanalyse auf die Testabdeckung durchgeführt. Diese ergab bei den ersten Analysen um die 75 Prozent Abdeckung, die oft darauf zurückzuführen waren, dass einige Exceptions nicht abgefangen wurden oder dass in manchen Codeteilen nicht alle Bedingungen getestet wurden, wie zum Beispiel bei der Klasse Polynomial im „- - Operator“. Im Anhang **[Anhang 5: Test - Nicht abgedeckter Code]**, ist eine farbliche Hervorhebung zu erkennen, welche Pfade nicht durchlaufen wurden.

Diese mussten im Anschluss an die Implementationstätigkeiten ebenfalls in die Teste integriert werden. So kam es noch zu einigen Fehlern, die durch diese Teste erst erkannt und behoben werden konnten.

So kann man im Anhang **[Anhang 6: Test - Code Coverage C#]** erkennen, dass abschließend alle Pfade der Bibliothek getestet sind. Dies entspricht dem Prinzip des Kantenüberdeckungstestes, dem sogenannten C1 Test (Semester 4: Vorlesung: Softwareengineering II, Dozent: Prof. Dr. Monett)

7 Portierung der C++ Bibliothek nach Python

In diesem Abschnitt wird die Portierung der Bibliothek nach Python beschrieben. Python ist eine dynamische, in der Regel interpretierte Programmiersprache. Sie unterstützt verschiedene Programmierparadigmen, wie Objektorientierung und funktionale Programmierung. Aufgrund der einfachen und übersichtlichen Syntax ist sie leicht zu erlernen und erfreut sich zunehmender Beliebtheit im Bildungs- und Forschungssektor.

Die Referenzimplementierung CPython, oft auch nur Python genannt, wird von der Python Software Foundation entwickelt und verfügt über ein Foreign Function Interface (FFI), welches auch den Aufruf von C Funktionen unterstützt. Darüber lassen sich Wrapper-Bibliotheken erstellen, welche die Funktionalität von C/C++ -Bibliotheken in Python zur Verfügung stellen. Dieser Weg, dessen zahlreichen Vorteile in Abschnitt 7.4 genauer erklärt werden, wurde auch für die Portierung der Bibliothek nach Python gewählt. Zuvor werden Wrapper allgemein, die nötigen Schritte für einen Python Wrapper von C++ Funktionen und die automatische Erstellung dieses erklärt.

7.1 Wrapper

Wrapper (von engl. „wrap“ = einpacken) sind Softwarestücke, die andere Software umgeben und deren Funktionalität anpassen. Dabei kann je nach Anwendungsgebiet die Originalsoftware erweitert, eingeschränkt oder nur die neue Schnittstelle hinzugefügt werden. Die Möglichkeit der Einschränkung erfolgt häufig aus Sicherheitsgründen oder um die Komplexität der Original-Software zu verstecken und die Handhabung zu vereinfachen. Der umgekehrte Teil, der Anpassung und Erweiterung, wird immer wieder eingesetzt, um bereits vorhandene Software-Bibliotheken auch in andern, meist neueren Programmiersprachen nutzbar zu machen und sie dem typischen Programmierstil der Programmiersprache anzupassen.

7.2 Erstellen eines Wrappers von C++ nach Python

In diesem Abschnitt wird das Vorgehen für das manuelle Erstellen eines Wrappers von C++ nach Python anhand eines einfachen Beispiels beschrieben.

```
class Car
{
    public:
        void drive();
};
```

Abbildung 7-1: C++ Code

Über das FFI in Python lassen sich nur C-Funktionen ansprechen, sodass zunächst die Car Klasse mit Hilfe von Zeigern und Funktionen abgebildet werden muss.

```
Car* new_Car() { return new Car(); }
void delete_Car(Car* obj) { delete obj; }
void drive_Car(Car* obj) { obj->drive(); }
```

Abbildung 7-2: C++ Funktionen

Zusammen mit diesen wird dann eine DLL kompiliert und in das Python Programm eingebunden.

```
from ctypes import cdll
carLib = cdll.LoadLibrary(pathToCarLib)
```

Abbildung 7-3: Python Import

Das ctypes Modul ist Teil der Python-Standardbibliothek und ermöglicht das Laden von DLLs in Python. Darüber hinaus stellt es nach C kompatible Datentypen zur Verfügung. Um auch in Python objektorientiert programmieren zu können, muss noch eine Wrapper-Klasse geschrieben werden, die den Referenzzeiger verwaltet und ihn bei jedem Funktionsaufruf mit an die Hilfsfunktionen übergibt.

```
class Car:
      def __init__(self):
            self.obj = carLib.new_Car()
      def __del__(self):
            carLib.delete_Car(self.obj)
      def drive(self):
            carLib.drive_Car(self.obj)
```

Abbildung 7-4: Python Class

Nun lassen sich auch in Python Instanzen der Car Klasse erzeugen und benutzen.

```
car = new Car()
car.drive()
```

Abbildung 7-5: Python Aufruf

Das hier vorgestellte Beispiel lässt sich weitestgehend auf die im Rahmen des Projektes entwickelte C++ Bibliothek für das Rechnen mit Matrizen aus Taylor-Polynomen übertragen. Python unterstützt alle dafür relevanten C++ Features wie Operatoren und Exceptions, sodass die gesamte Funktionalität der Bibliothek auch in Python zur Verfügung steht.

7.3 SWIG

Der Aufwand für den in Abschnitt 7.2 beschriebenen Vorgang zum Erstellen eines Wrappers nimmt mit dem Umfang der Orignial-Software zu. Außerdem muss der Wrapper bei Erweiterungen der Software ebenfalls angepasst werden um die hinzugekommenen Funktionen zu unterstützen. Um den zukünftigen Aufwand bei der Weiterentwicklung der Bibliothek gering zu halten, wurde daher nach einer nachhaltigen und weitestgehend automatischen Lösung für das Erstellen des Wrappers gesucht.

Der Simplified Wrapper and Interface Generator, kurz SWIG, erfüllt diese Anfordungenen. Das mittlerweile seit 17 Jahren bestehende Open Source Projekt, erlaubt des Erstellen von Wrappern für C/C++ Programme. In der aktuellen Version 2.0.6 werden zahlreiche Zielsprachen unterstützt, darunter auch Java, PHP und Python. Dafür genüg SWIG eine Konfigurationsdatei, wie sie im Anhang zu finden ist, sowie kleine Modifikationen an der C++ Header-Datei für einige Operatoren. Außerdem mussten die print() Methoden in der Python Portierung umbenannt werden, da „print“ in Python ein reserviertes Schlüsselwort ist. Dies wird mit den „%rename“ Anweisungen in der Konfigurationsdatei bewerkstelligt.

7.4 Vor- und Nachteile des Wrappers

Die Portierung mittels Wrapper bietet zahlreiche Vorteile. Der für viele Python Entwickler wichtigste dürfte die Performance des kompilierten Bytecodes darstellen, weshalb es häufig vorkommt, das aufwändige Berechnung in Python in eine C/C++ - Bibliothek ausgelagert werden. Dieser Geschwindigkeitsgewinn wird mit dem Preis erkauft, dass das Python Modul für jede Plattform neu kompiliert werden muss, während reiner Python Code in der Regel plattformunabhängig ist. Da es sich bei dem Projekt um eine Bibliothek für mathematische Berechnungen handelt, ist dieser Austausch akzeptabel.

Ein weiterer Vorteil ergibt sich aus der gemeinsamen Codebasis mit der C++ Version. Zukünftig Änderungen, zum Beispiel Bugfixes oder neue Funktionen, müssen nur einmal implementiert werden und stehen dann sowohl in der C++ Variante als auch in der Python Portierung zur Verfügung. Dies gilt ebenfalls für die Test Cases, sodass die Richtigkeit der Python Portierung durch die C++ Test Cases abgesichert ist.

8 Beispielanwendungen

8.1 Nachweis der Funktionalität der C++ Klassenbibliothek

Die Klassenbibliothek der C++ - Variante kann durch zwei unterschiedliche Methoden eingebunden werden, die nachfolgend beschrieben werden. Für die Konsolenanwendung wurde die Variante mit dem Einbinden über die Projekteigenschaften gewählt.

8.1.1 Einbinden der C++ Bibliothek über Projekteigenschaften

In diesem Abschnitt wird beschrieben, wie die Klassenbibliothek über die Projekteigenschaften im Visual Studio (Version 2010) eingebunden werden kann. Vorteil bei dieser Variante ist, dass man die zugehörigen Bibliotheksdateien (TaylorPLib.lib, TaylorPLib.dll und TaylorPLib.h) an einem zentralen Ort aufbewahren kann und im Falle eines Updates nur diesen Ort mit der neuen Version der Klassenbibliothek aktualisieren muss.

Die Klassenbibliothek kann in vier Schritten in das Projekt eingebunden werden. Im ersten Schritt wird die Bibliothek, bestehend aus den Komponenten TaylorPLib.lib, TaylorPLib.dll und TaylorPLib.h, in einen Ordner kopiert. Im zweiten Schritt muss man die „Path“ *Umgebungsvariable* des Projektes entsprechend anpassen, wie es im **[Anhang 7: Einbinden der Bibliothek - Einstellung 1]** abgebildet ist. Dazu muss man mit einem Rechtsklick auf das Projekt klicken und den Menüpunkt Projekteigenschaften auswählen. Anschließend unter dem Punkt Konfigurationseigenschaften auf Debuggen wählen und unter dem Punkt Umgebung folgendes eintragen:

```
PATH = %PATH%; <Pfad zur DLL>
```

Abbildung 8-1: Einbinden der Bibliothek - Einstellung 1

Der Parameter <Pfad zur DLL> kann entweder eine absolute oder relative Pfadangabe sein und muss auf den Ordner verweisen, der die TaylorPLib.dll beinhaltet. Diese Einstellung sorgt dafür, dass während des Ausführens des Projektes aus dem Visual Studio heraus, die neu erstellte Anwendung Zugriff auf die Klassenbibliothek hat.

Im nächsten Schritt muss man in den Konfigurationseigenschaften unter dem Punkt C/C++ im Menü Allgemein den Pfad für die „Zusätzliche Includeverzeichnisse“ anpassen, wie es im **[Anhang 8: Einbinden der Bibliothek - Einstellung 2]** abgebildet ist. An dieser Stelle gibt man den Pfad zur TaylorPLib.h Datei an:

```
<Pfad zur Headerdatei>
```

Abbildung 8-2: Einbinden der Bibliothek - Einstellung 2

Hier kann ebenfalls wieder eine absolute oder relative Pfadangabe erfolgen und diese Pfadangabe sorgt dafür, dass die Headerdatei, wenn man sie über das Include-Statement in den Quelltext eingefügt, gefunden wird ohne dass man die Datei mit in das Projekt aufnehmen muss.
Im letzten Schritt müssen die zusätzlichen Abhängigkeiten hinzugefügt werden. Dies geschieht, wie in **[Anhang 9: Einbinden der Bibliothek - Einstellung 3]** abgebildet, über den Menüpunkt Konfigurationseigenschaften -> Linker -> Eingabe unter dem Punkt „Zusätzliche Abhängigkeiten“.

```
<Pfad zur LIB-Datei>/TaylorPLib.lib
```

Abbildung 8-3: Einbinden der Bibliothek - Einstellung 3

Hier ist es ebenso wie in den vorher genannten Punkten, mit der Ausnahme, dass an dieser Stelle der Ordnerpfad nicht reicht, sondern dass man den Dateipfad, inklusive Dateinamen angeben muss. Diese Eigenschaft bewirkt, dass der Linker bei der Erstellung des Projektes Platzhalter für die benutzten Komponenten der Klassenbibliothek, die er aus der LIB-Datei entnimmt, in den Programmcode einbaut.
Hat man diese Einstellungen vorgenommen und die Headerdatei inkludiert, kann man nun eine Anwendung schreiben, die die Klassenbibliothek nutzt. Nach Veröffentlichung der Anwendung ist jedoch darauf zu achten, dass die DLL dem ausführbaren Programm beigelegt wird, da es sonst zu Programmabstürzen kommen kann, die daraus hervorgehen, dass die Klassenkomponenten nicht gefunden werden.

8.1.2 Einbinden der C++ Bibliothek über Headerdatei

In diesem Abschnitt wird beschrieben, wie die Klassenbibliothek direkt in ein neues Visual Studio Projekt eingebunden werden kann. Dazu muss zuerst ein neues C++ Projekt erstellt werden und im Anschluss daran, die Dateien TaylorPLib.h und TaylorPLib.lib in das Projektverzeichnis kopiert werden. Anschließend müssen mit einem Rechtsklick auf das Projekt und dann unter dem Menüpunkt „Hinzufügen“ -> „Vorhandenes Element“ die beiden Dateien ausgewählt und hinzugefügt werden. Nun kann man die Headerdatei wie gewohnt über das Include-Statement einbinden und die Bibliothek verwenden. Zum Ausführen der Anwendung wird weiterhin die DLL benötigt, die dann ebenfalls in das Ausführungsverzeichnis der Anwendung kopiert werden muss. Dies sind beim Visual Studio, je nach Buildkonfiguration, die Verzeichnisse „Debug“ oder „Release“ innerhalb des Projektverzeichnisses.
Sollte die Klassenbibliothek aktualisiert werden, so müssen die Dateien im Projektverzeichnis mit den neueren überschrieben und das Projekt anschließend neu erstellt werden.

8.1.3 Konsolenanwendung (C++) zum Nachweis der Funktionalität

Zum Nachweis der Funktionalität der Klassenbibliothek der C++ - Variante ist eine Beispielanwendung entwickelt worden. Diese soll die Verwendung und Einbindung der Klassenbibliothek demonstrieren und weiterhin den Nachweis erbringen, dass die Funktionalitäten korrekt implementiert sind.
Da die Beispielanwendung nur zum Test der Bibliothek konzipiert ist, wurden die Projektabhängigkeiten über den beschriebenen Weg, das Einbinden der C++ Bibliothek über die Headerdatei, hinzugefügt. Die Beispielanwendung ist eine Konsolenanwendung, die die einfache Verwendung der Bibliothek demonstriert. So gibt es in ihr die Funktionen „createSimpleMatrixAndPrint()“, die zwei Matrizen instantiiert und auf der Konsole ausgibt. Weiterhin werden hier auch die Operatoren „+“, „-“, „*“, „==“ und „!=“demonstriert und getestet. Eine weitere Funktion ist „matrixMultiplication()“ in der einige Multiplikationsfunktionen, wie zum Beispiel „mmCaABbC()“ oder „mmCaAATbC()“, aufgerufen und die Resultate auf der Konsole ausgegeben werden. Abschließend gibt es noch die Funktion „checkError()“, welche

die Verwendung der Klasse „MathException“ demonstriert, in der zwei Matrizen unterschiedlicher Größe miteinander addiert werden und es zu einer erwarteten Exception kommt. Der Text der Exception wird auf der Konsole ausgegeben.

8.2 Nachweis der Funktionalität der C# Klassenbibliothek

8.2.1 Einbinden der C# Bibliothek

In diesem Abschnitt wird beschrieben, wie die Klassenbibliothek der C# - Variante direkt in ein neues Visual Studio Projekt eingebunden werden kann. Dazu muss ein neues C# Projekt erstellt werden, und im Anschluss daran ein Rechtsklick auf Projektmappe – Verweise, auf den Menüpunkt „Verweis hinzufügen“ getätigt und dann bei „Durchsuchen“ der Pfad zur aktuellen Klassenbibliothek der C# - Variante eingetragen werden.
Durch diese Einstellung kann man die DLL benutzen und sie wird beim Kompilieren automatisch in das Ausführungsverzeichnis kopiert.
Im nächsten Schritt kann man eine App.config Datei hinzufügen und unter dem XML - Knoten „configuration“ die folgenden Zeilen einfügen:

```
<runtime>
  <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
    <probing privatePath="LIB"/>
  </assemblyBinding>
</runtime>
```

Abbildung 8-4: App.config Einstellungen

Durch diesen Schritt wird bei der Kompilierung der Anwendung im Ausgabeverzeichnis eine Datei <Anwendungsname>.exe.config erstellt, die zusätzliche Informationen für die Programmausführung beinhaltet. Nun kann man im Verzeichnis der Anwendung einen Ordner „LIB“ erstellen und die DLL dort hinein verschieben. Beim Start der Anwendung wird die <Anwendungsname>.exe.config Datei verarbeitet und die zugehörigen Abhängigkeiten, wie zum Beispiel die entsprechende DLL im Ausführungspfad und im LIB Ordner gesucht.

8.2.2 Konsolenanwendung (C#) zum Nachweis der Funktionalität

Zum Nachweis der Funktionalität der Klassenbibliothek in C# wurde, wie schon bei der C++ - Version, eine Konsolenanwendung geschrieben, die durch einen Verweis auf die C# DLL, die Funktionalitäten der Klassenbibliothek implementiert.

Analog zur Beispielanwendung der C++ - Version wurden hier ebenfalls einige Funktionen zur Ausgabe auf der Konsole programmiert. So zum Beispiel „showPolynom()“, „showMatrix()“, „showSpecialMatrixMultiplication()“ und „showError()“.

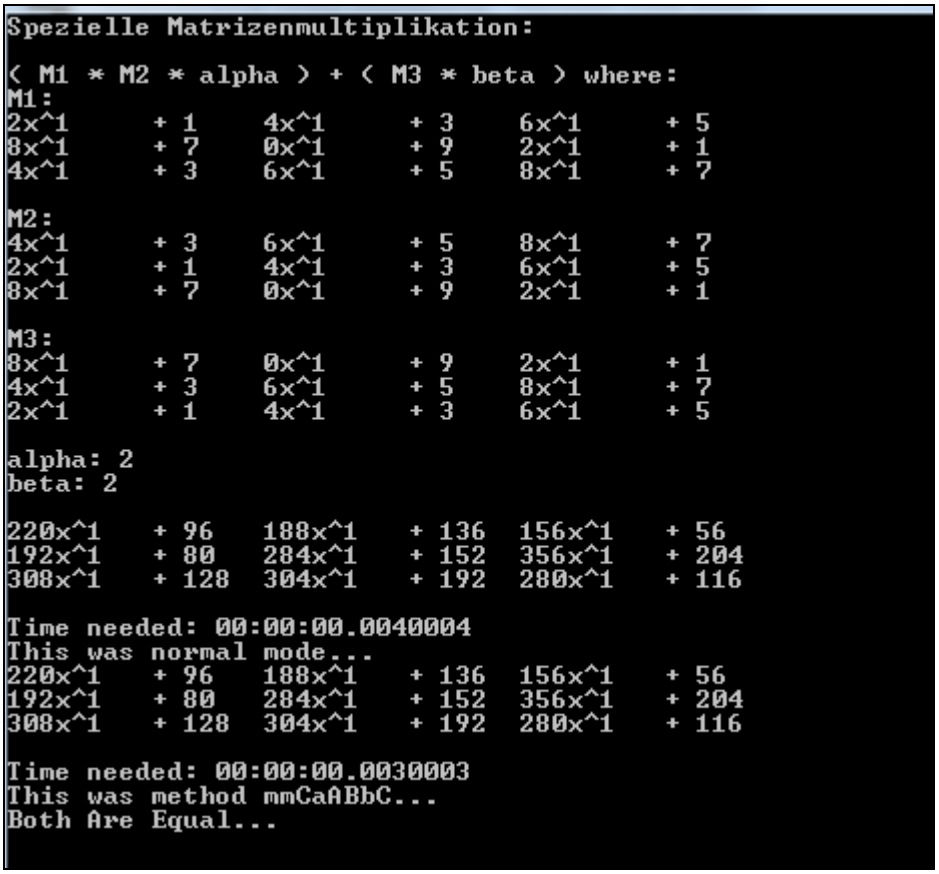


Abbildung 8-5: CSharp Beispielanwendung

In Abbildung 8-5 ist die Ausgabe der Funktion „showSpecialMatrixMultiplication()“ zu sehen, wo zuerst die initialisierten Matrizen M1 – M3 sowie die Werte für „alpha“ und „beta“ angegeben sind. Im nächsten Abschnitt wird die Matrizenmultiplikation auf herkömmlichem Wege durchgeführt, also durch Aufruf von „(M1 * M2 * alpha) + (M3 * beta)“. Anschließend folgt die Ausgabe nach Aufruf der Funktion „mmCaABbC()“. Die Resultate sind identisch.

8.3 Nachweis der Funktionalität der Python Klassenbibliothek

8.3.1 Einbinden der Python Bibliothek

In diesem Abschnitt wird beschrieben wie Klassenbibliothek in ein Python Programm eingebunden wird. Dazu werden die Dateien _TaylorPLib.pyd und TaylorPLib.py benötigt. Diese werden beim Kompilieren im Verzeichnis src/python/ erstellt. Im Verzeichnis bin/Python/Library finden sich zudem für Windows 7 64bit kompilierte Dateien. Beide Dateien müssen entweder im Pfad liegen oder im Hauptverzeichnis des Programmes. Bei der Auslieferung des Programmes müssen die Dateien ebenfalls mitgeliefert werden.

In allen Python Dateien, die auf Funktionen der Klassenbibliothek zugreifen, muss diese zunächst importiert werden.

```
import TaylorPLib
```

Abbildung 8-6: Python Import

Matrizen und Polynome lassen sich nun unter Angabe des Moduls erstellen:

```
M = TaylorPLib.Matrix(2, 3)
p = TaylorPLib.Polynomial(2)
```

Abbildung 8-7: Python Verwendung

Um die Schreibweise zu verkürzen können "from TaylorPLib import Matrix, Polynomial" statt „import TaylorPLib“ beim Import auch Referenzen im aktuellen Namensraum angelegt werden.

```
from TaylorPLib import Matrix, Polynomial
M = Matrix(2, 3, 2)
P = Polynomial(2)
```

Abbildung 8-8: Python Verwendung 2

8.3.2 Konsolenanwendung zum Nachweis der Funktionalität

Zum Nachweis der Klassenbibliothek in Python wurde ein kurzes Python Script example.py erstellt. Es demonstriert in einigen Schritten wie Matrizen und Polynome erstellt und mit ihnen gerechnet wird.

9 Erläuterungen zur Kompilierung des gesamten Projektes

9.1 Vorbereitungen

Um das gesamte Projekt zu kompilieren sind einige Vorbereitungen zu treffen. Es werden die Komponenten „Google C++ Testing Framework“, Python und SWIG (swigwin) benötigt.

Die Quellen sollten in einen Ordner (zum Beispiel: D:\Quellen\TaylorPLib) mit *git* geklont werden (Repository-Url: https://github.com/Marvin182/TaylorPLib.git). Anschließend muss das Google C++ Testing Framework in das Verzeichnis „D:\Quellen\GTest“ entpackt werden. Python wird durch eine Setup-Datei installiert und swigwin kann an einen beliebigen Ort extrahiert werden (Empfehlung: in einen Unterordner im Python Installationsordner zum Beispiel: C:\Python33\swigwin).

Nach Erstellung der benötigten Ordner müssen im Anschluss noch einige Umgebungsvariablen angepasst und angelegt werden. So muss die Umgebungsvariable „PATH“, zu finden unter Systemsteuerung -> System -> Erweiterte Systemeinstellungen -> Reiter Erweitert -> Umgebungsvariablen -> Systemvariablen um den folgenden Eintrag ergänzt werden:

```
C:\Python33\swigwin\;C:\Python33\;
```

Abbildung 9-1: Ergänzung der PATH - Umgebungsvariable

In der Abbildung 9-1 ist vorausgesetzt, dass Python in der Version 3.3 im Ordner C:\Python33 installiert wurde und die swigwin Dateien im Ordner C:\Python33\swigwin entpackt wurden.

Weiterhin müssen die Umgebungsvariablen „PYTHON_INCLUDE“, mit dem Wert „C:\Python33\include\“, und „PYTHON_LIB“, mit dem Wert „C:\Python33\libs\python33.lib“, angelegt werden, wie in Abbildung 9-2 zu erkennen ist.

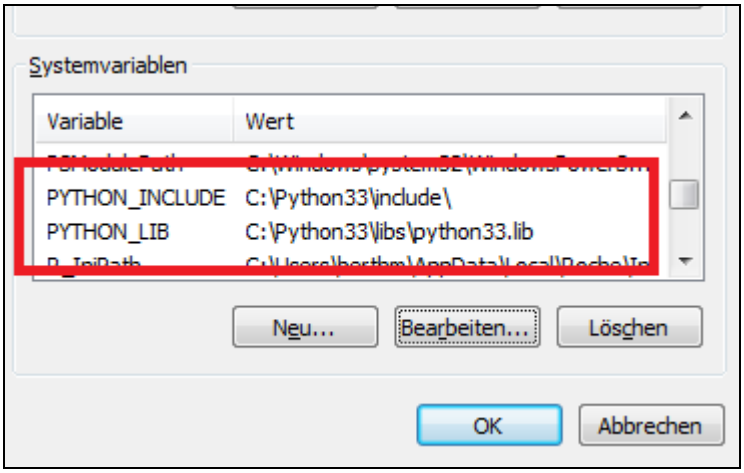


Abbildung 9-2: weitere Umgebungsvariablen

Nachdem diese Vorbereitungen abgeschlossen sind, muss im nächsten Schritt das Google C++ Testing Framework kompiliert werden. Dazu sind einige spezifische Schritte notwendig. Als erstes muss man den Schreibschutz für den Ordner „D:\Quellen\GTest\msvc“ aufheben, indem man einen Rechtsklick auf den Ordner macht und den Menüpunkt Eigenschaften auswählt. Nun muss man das Attribut „Schreibschutz“ deselektieren und mit einem Klick auf „OK“ die Änderungen für alle Dateien und Unterordner übernehmen. Anschließend kann man im Ordner „D:\Quellen\GTest\msvc“ die Projektmappendatei „gtest.sln“ mit dem „MS Visual Studio 2010“ öffnen (derzeit ist GTest mit dem MS Visual Studio 2012 noch nicht kompatibel). Im nächsten Schritt muss die Projektmappenkonfiguration zur Erstellung von Debug auf Release umgestellt werden.

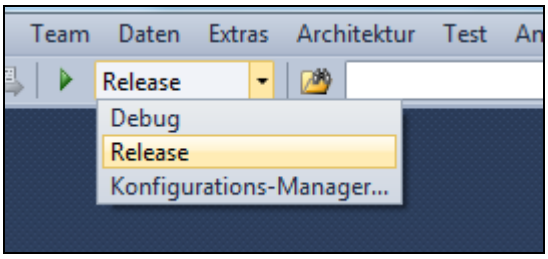


Abbildung 9-3: Projektmappeneigenschaften

Weiterhin müssen in den vier Projektmappen jeweils eine weitere Einstellung vorgenommen werden. Dazu muss auf jede Projektmappe mit einem Rechtsklick das Menü Eigenschaften gewählt werden, und dann unter Konfigurationseigenschaften -> C/C++ -> Codegenerierung der Eintrag „Laufzeitbibliothek“ von

„Multithreaded (/MT)“ auf „Multithreaded-DLL (/MD)“ geändert werden (abgebildet im **[Anhang 4: Einstellungen Google C++ Testing Framework]**). Jetzt kann das Projekt mit einem Tastendruck auf „F6“ kompiliert werden. Als Ergebnis der Kompilierung sollten im Ordner „D:\Quellen\GTest\msvc\gtest\Release\“ die Dateien „gtest.lib“ und „gtest_main.lib“ erstellt worden sein. Diese werden für das Hauptprojekt benötigt.

9.2 Hauptteil

Nach Abschluss aller Vorbereitungen kann im Ordner „D:\Quellen\TaylorPLib\src“ die Projektmappendatei „TaylorPLib.sln“ mit dem MS Visual Studio 2010 geöffnet werden. Hier sollte ebenfalls wieder die Projektmappeneigenschaft von Debug auf Release umgestellt werden. Sind alle Vorbereitungen richtig erfolgt, so kann ebenfalls das Projekt mit einem Tastendruck auf „F6“ kompiliert werden. Ist die Kompilierung ohne Fehler verlaufen, so befinden sich im Ordner „D:\Quellen\TaylorPLib\Release“ die Unterordner „C++“, „CSharp“ und „Python“, in denen jeweils ein Unterordner Library ist, der die Bibliotheken beinhaltet. Zudem sind in den 3 Unterordnern jeweils die kompilierten Beispielanwendungen zu finden, die entsprechend gestartet werden können.

9.3 Einzelne Projekte kompilieren

Für den Fall, dass nur einzelne Teile des Projektes kompiliert werden müssen, muss man nicht die kompletten Vorbereitungen treffen. Soll zum Beispiel nur das C# Projekt kompiliert werden, können die gesamten Vorbereitungen weggelassen werden und im Hauptprojekt muss man jeweils die nicht benötigten Projekte entladen. Dazu geht man auf die C++- und Python-Projekte mit einem Rechtsklick und wählt den Menüpunkt „Projekt entladen“. So werden die jeweiligen Projekte bei der Kompilierung nicht benötigt und es wird nur die Bibliothek und die Anwendung der C# Projekte erstellt.

Möchte man die Projekte anschließend doch kompilieren, so muss man mit einem Rechtsklick auf das jeweilige Projekt den Menüpunkt „Projekt erneut laden“ wählen, um die Projekte zur Erstellung hinzuzufügen.

10 Probleme (MB)

Während der Entwicklung der Klassenbibliothek kam es an einigen Stellen zu Problemen und Unklarheiten. Die größte Hürde in diesem Projekt war es den gegebenen Quellcode ohne die Zuhilfenahme von Beispielen in der Gesamtheit zu verstehen. Das Verständnis musste Funktion für Funktion erarbeitet und auch mithilfe von Literatur angeeignet werden. Dieser Punkt nahm dementsprechend einen gewissen Zeitraum in Anspruch, sodass die Umsetzung nicht zu 100 Prozent abgeschlossen werden konnte.

Ebenfalls an den zeitlichen Rahmen gekoppelt ist der Versuch, dieses Projekt unter den Rahmenbedingungen des Test Driven Development zu entwickeln. Die Qualität des Endproduktes ist durch diesen Ansatz merklich höher, die aufgewandte Zeit stieg dafür jedoch um einiges an. Um eine einhundert prozentige Codeabdeckung zu haben, kamen auf circa 2.600 Zeilen Quellcode ungefähr 2.900 Zeilen Testcode (in der C# Portierung). Dies ist durchaus ein erheblicher Schreibaufwand, der am Ende enorm Zeit gekostet hat.

11 Probleme (MR)

Anfänglich gab es Schwierigkeiten beim Verständnis des vorhandenen Quellcodes. Verstärkt durch den großen Umfang, war nicht klar welche Teile für die zu erstellende Klassenbibliothek relevant sind. Dazu kam, dass vorhandene Beispiele nicht erklärt und ohne das nötige Hintergrundwissen nicht verständlich waren.

Für weitere Verwirrung sorgten Optimierungen in den Algorithmen, die nur bei richtiger Parametrisierung zulässig sind, welche nicht dokumentiert waren. Nachdem die korrekte Funktionsweise mit Hilfe der Fachliteratur erarbeitet wurde, konnten passende Test Cases geschrieben und die Algorithmen anpasst werden. Allein für die 3.000 Zeilen große Matrix Klasse stellte dies einen enormen Arbeitsaufwand dar, weshalb am Ende auch einige wenige Hilfsfunktionen nicht implementiert werden konnten.

Gegen Ende kam es noch zu kleineren Problemen mit dem von SWIG erstelltet Python-Modul, welche auf die Unterschiede zwischen C++ und Python zurückzuführen sind. So ist beispielsweise das Wort "print" in Python ein

Schlüsselwort und darf nicht als Funktionsname verwendet werden. Entsprechende Funktionen mussten daher, durch Anpassungen der Interface-Datei für SWIG, für das Python-Modul umbenannt werden.

12 Fazit (MB)

Dieses Studienprojekt kann insgesamt als erfolgreich angesehen werden. Die Zielsetzungen, der Erstellung einer Klassenbibliothek und eine eigene Portierung durch jeden Studenten, wurden umgesetzt. Weiterhin ergab sich die Möglichkeit, im Rahmen des Studienprojektes, die Entwicklungsmethode des Test Driven Developments einzusetzen und kennenzulernen.

Aus meiner Sicht war es somit ein sehr spannendes und lehrreiches Projekt. Das Themengebiet musste zuerst vertiefend angeeignet werden und da der Quellcode vorhanden war, konnten wir uns vor allem auch auf Softwareteste fixieren und so in diesen beiden Aspekten einen großen Lernerfolg erzielen.

Weiterhin wurde dieses Projekt als Teamprojekt erarbeitet, in denen jeder seine vorher abgestimmten Aufgaben und Pflichten hatte und diese aus meiner Sicht sehr gut umgesetzt worden sind.

13 Ausblick (MB)

Im Ausblick auf die Weiterentwicklung dieses Projektes gibt es noch einige Ansätze, die zu implementieren wären. Wie schon erwähnt wurden nicht alle Funktionen des gelieferten Quellcodes implementiert, da die Entwicklungsmethode des Test Driven Developments enorm Zeit gekostet hat. Diese Funktionalitäten sollten noch implementiert werden. Weiterhin denke ich, dass es sinnvoll ist, eine Funktionalität zu implementieren, die Matrizen aus Taylorpolynomen aus zum Beispiel Textdateien einliest. Ich denke dies wäre eine weitere sinnvolle Ergänzung zu diesem Projekt.

14 Fazit (MR)

Zu Beginn des Projektes wurde ein Projektplan aufgestellt, welcher erfolgreich in die Tat umgesetzt werden konnte. Die Einarbeitung in das Thema hat sich wie erwartet als schwer und zeitintensiv herausgestellt, war aber auch sehr interessant und lehrreich. Der vorhandene Quellcode war gut dokumentiert und konnte an einigen Stellen weiter optimiert werden und ist nun eine eigenständige Bibliothek. Dabei konnte ich meine C++ Kenntnisse erweitern und lernte das Google Testing Framework kennen.

Durch die Portierung des Programmes nach Python, lernte ich zudem auch eine für mich neue aufstrebende Programmiersprache kennen.

Große Teile des Projektes entstanden in Teamarbeit mit Michael Berth, welche dank regelmäßiger und klarer Absprachen, in meinen Augen sehr gut funktioniert hat.

15 Ausblick (MR)

In Hinsicht auf die Zukunft des Projektes, wurde viel Wert auf übersichtlichen und gut dokumentierten Code gelegt. Dessen Richtigkeit ist zudem durch die im Zuge des Test Driven Development entstandenen Test Cases gesichert, sodass auch bei Änderung durch die Weiterentwicklung Fehler direkt sichtbar werden. Dies sollte auch bei den noch fehlenden Funktionen fortgesetzt werden.

Als eine interessante Erweiterung könnte ich mir das Erstellen von Benchmarks vorstellen. Diese würden helfen performance-kritische Abschnitte ausfindig zu machen und weiter zu verbessern.

Darüber hinaus wäre es natürlich auch wichtig auf die Wünsche der Nutzer der Klassenbibliothek einzugehen.

Glossar

- **Assertions**

  Assertions beschreiben bei Softwaretesten die Annahmen, auf die der Quellcode zu testen ist. Zum Beispiel könnte eine Annahme sein, dass ein bestimmter Wert nach Aufruf einer Methode den Wert True hat.

- **Coding Conventions**

  Coding Conventions beschreiben den Programmierstil, auf den man sich im Team vor Beginn der Programmierarbeiten einigt.

- **Dynamic Link Library**

  Eine "Dynamic Link Library" ist ein Modul, welches Funktionen und Daten beinhaltet, die von anderen Modulen (Anwendung oder DLL) genutzt werden können.[1]

- **Exception**

  „Exceptions […] stellen eine strukturierte, einheitlich und typsichere Art der Behandlung von Fehlerbedingungen auf System- oder Anwendungsebene dar."[2]

- **Git**

  Git ist eine freie Versionsverwaltung für Dateien.

- **Klassenbibliothek**

  Eine Klassenbibliothek beinhaltet Klassen, die man in mehreren Projekten wiederverwenden kann. Diese Klassenbibliothek kann in kompilierter Form (DLL) oder in Quellcode vorliegen und eingebunden werden.

[1] Vgl. http://msdn.microsoft.com/en-us/library/windows/desktop/ms682589(v=vs.85).aspx und Studienarbeit I
[2] Hejlsberg, Wiltamuth, Golde, München 2007, Die C#-Programmiersprache, S. 445

- **.LIB-Datei**

  Eine .lib-Datei ist eine statische Bibliothek, die es dem Linker erlaubt, Platzhalter für Funktionen aus der verwendeten Bibliothek einzubinden. Zur Laufzeit des Programmes wird dann die entsprechende DLL benötigt.

- **Milestone**

  Ein Milestone ist ein definierter Zeitpunkt, an dem ein Arbeitspaket fertig zu sein hat, oder der Zeitpunkt, an dem ein bestimmtes Arbeitspaket abgearbeitet wurde.

- **Portierung**

  Von einer Portierung spricht man in der Regel dann, wenn man ein Projekt von einer Programmiersprache in eine andere übersetzt.

- **Refactoring**

  In der Softwareentwicklung spricht man vom Refactoring, wenn man zum Beispiel den Quellcode „aufräumt“. Zum Beispiel wenn man redundanten Code in eine eigene Funktion auslagert und an allen Stellen, die den Code nutzen, die Funktion benutzt.

- **Test Driven Development**

  Test Driven Development bedeutet Testgetriebenes Entwickeln. Bei dieser Methode der Softwareentwicklung werden für jede neue Funktionalität zuerst Softwareteste geschrieben, und anschließend die Funktionalität solange implementiert und verbessert, bis der Test nicht mehr fehlschlägt.

- **Umgebungsvariable**

  Die Umgebungsvariablen in Windows werden von bestimmten Programmen genutzt, um eine globale Einstellung zu haben. So wird zum Beispiel die Umgebungsvariable „JAVA_HOME“ vom Java Interpreter verwendet.

Literaturverzeichnis

1. Informationen zum Google C++ Testing Framework
   http://code.google.com/p/googletest/wiki/Primer
   Stand: 31.01.2012 10:00

2. Allgemeine Definitionen zur Programmierung
   Hejlsberg, Anders; Wiltamuth, Scott; Golde, Peter: Die C#-Programmiersprache, 2. Aufl., Addison-Wesley Verlag, München 2007

3. Informationen zu C++
   Jürgen Wolf: C++ von A bis Z: Das umfassende Handbuch, 2. Auflage, Galileo Computing, 2009

4. Informationen zu Dynamic Link Libraries
   http://msdn.microsoft.com/en-us/library/windows/desktop/ms682589(v=vs.85).aspx
   Stand: 19.02.2013 12:00

5. Informationen zu Python
   Lutz, Mark; Learning Python, 4. Aufl., O'Reilly Media 2009

6. Informationen zu SWIG
   Lucks, Julius B.; Python - All a Scientist Needs, Version 1, 12. März 2008
   http://arxiv.org/abs/0803.1838
   Stand: 27.02.2013 17:00

7. SWIG Project: SWIG-2.0 Dokumentation, Stand: 16. Dezember 2012, http://www.swig.org/Doc2.0/SWIGDocumentation.pdf
Stand: 27.02.2013 17:00

8. David M. Beazley: Interfacing C/C++ and Python with SWIG, November 1998, International Python Conference
http://www.swig.org/papers/PyTutorial98/PyTutorial98.pdf
Stand: 27.02.2013 17:00

Eidesstattliche Erklärung (MB)

Name: Michael Berth
Matrikelnr.: 603699

Hiermit erkläre ich an Eides Statt, dass ich die vorliegende Arbeit bis auf die offizielle Betreuung selbst und ohne fremde Hilfe angefertigt habe und die benutzten Quellen und Hilfsmittel vollständig angegeben sind.

Berlin, den 2. März 2013 .....

Unterschrift

Hochschule für
Wirtschaft und Recht Berlin
Berlin School of Economics and Law

Eidesstattliche Erklärung (MR)

Name: Marvin Ritter
Matrikelnr.: 620373

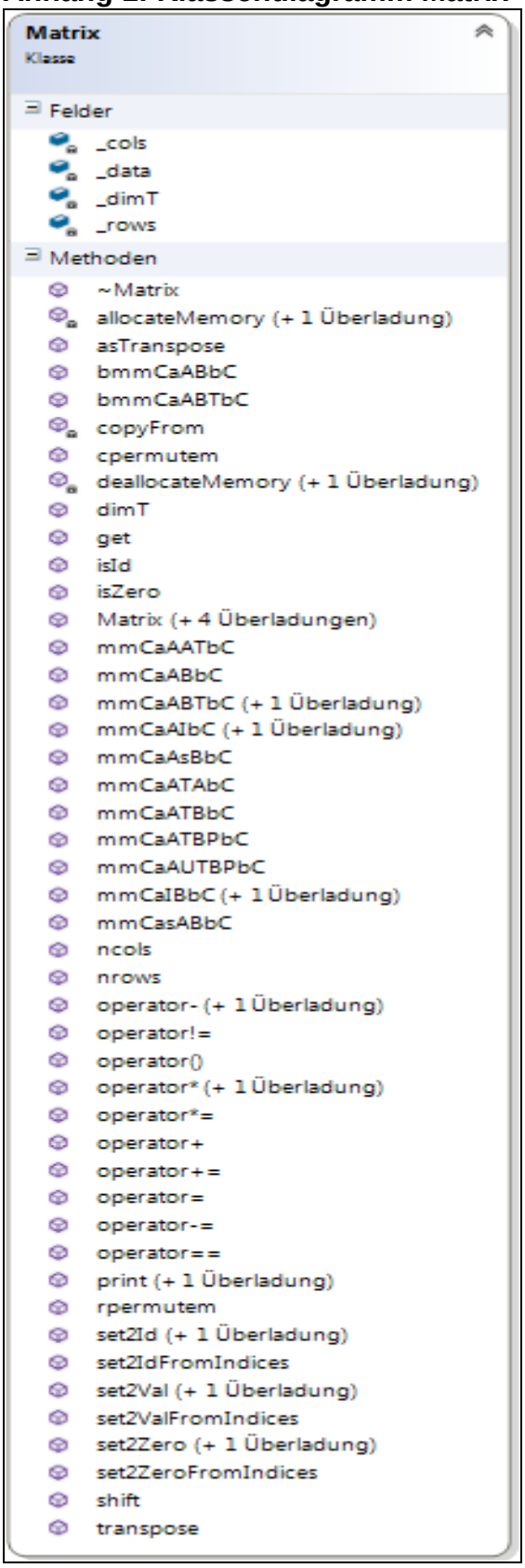
Hiermit erkläre ich an Eides Statt, dass ich die vorliegende Arbeit bis auf die offizielle Betreuung selbst und ohne fremde Hilfe angefertigt habe und die benutzten Quellen und Hilfsmittel vollständig angegeben sind.

Berlin, den 2. März 2013 .....

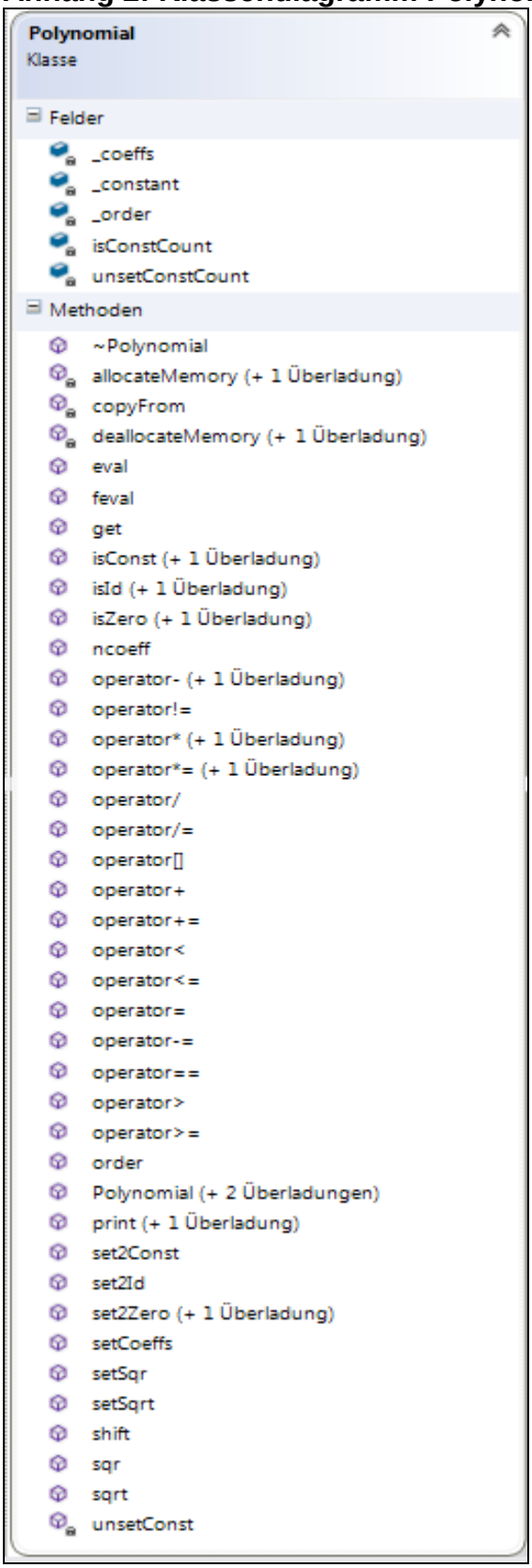
Unterschrift

Anhang

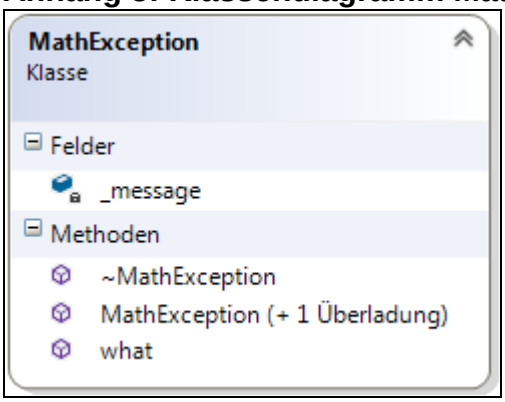
Anhang 1: Klassendiagramm Matrix



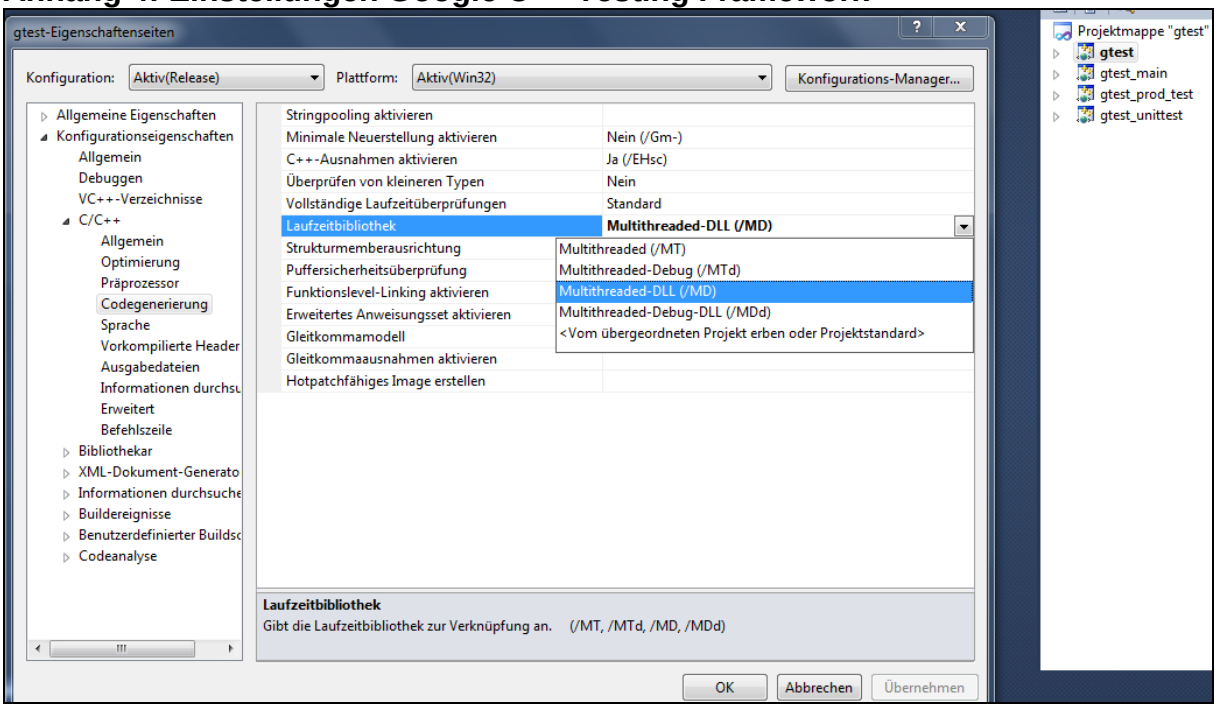
Anhang 2: Klassendiagramm Polynomial



Anhang 3: Klassendiagramm MathException



Anhang 4: Einstellungen Google C++ Testing Framework



Anhang 5: Test - Nicht abgedeckter Code

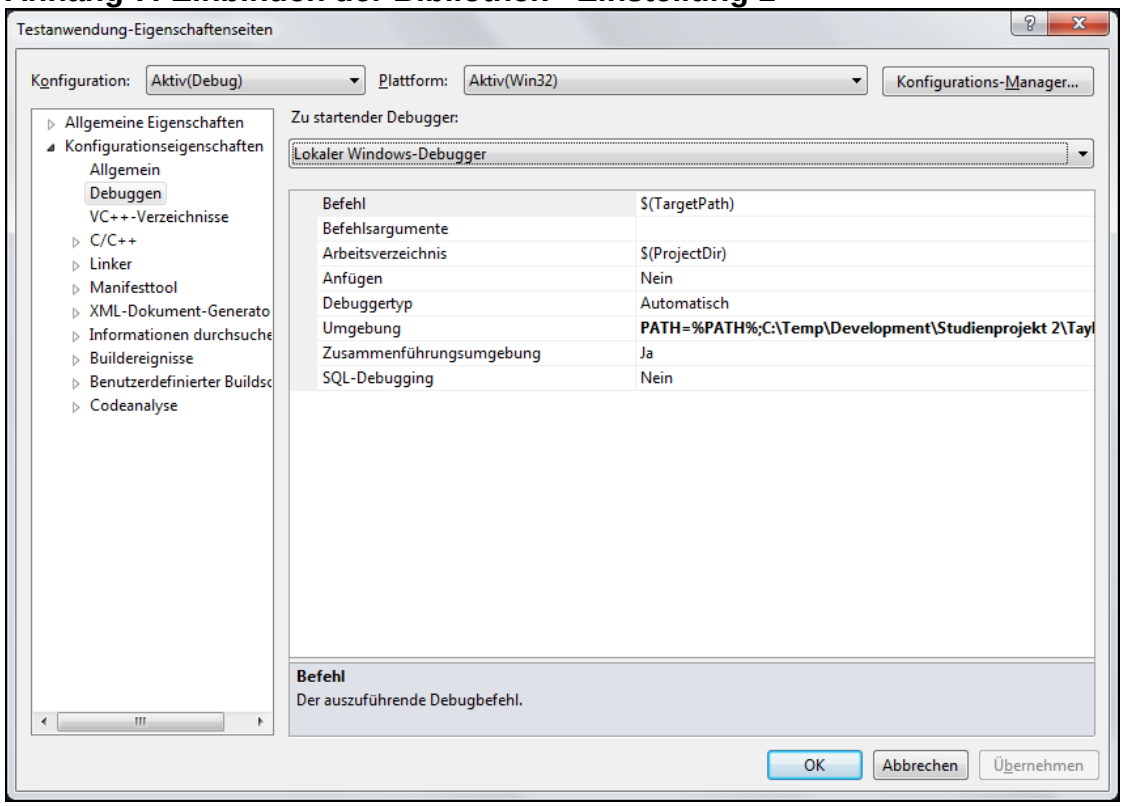
```
public static Polynomial operator -(Polynomial a, Polynomial b)
{
    if (a._order != b._order)
    {
        throw new MathException("The order of both Taylor Polynoms should match. ");
    }

    Polynomial retval = new Polynomial(a._order);
    if (a.isConst())
    {
        retval = new Polynomial(-b);
        retval[0] = a[0] - b[0];
    }
    else if (b.isConst())
    {
        retval = new Polynomial(a);
        retval[0] -= b[0];
    }
    // general case
    else
    {
        for (int i = 0; i <= a._order; i++)
            retval[i] = a[i] - b[i];
    }
    return retval;
}
```

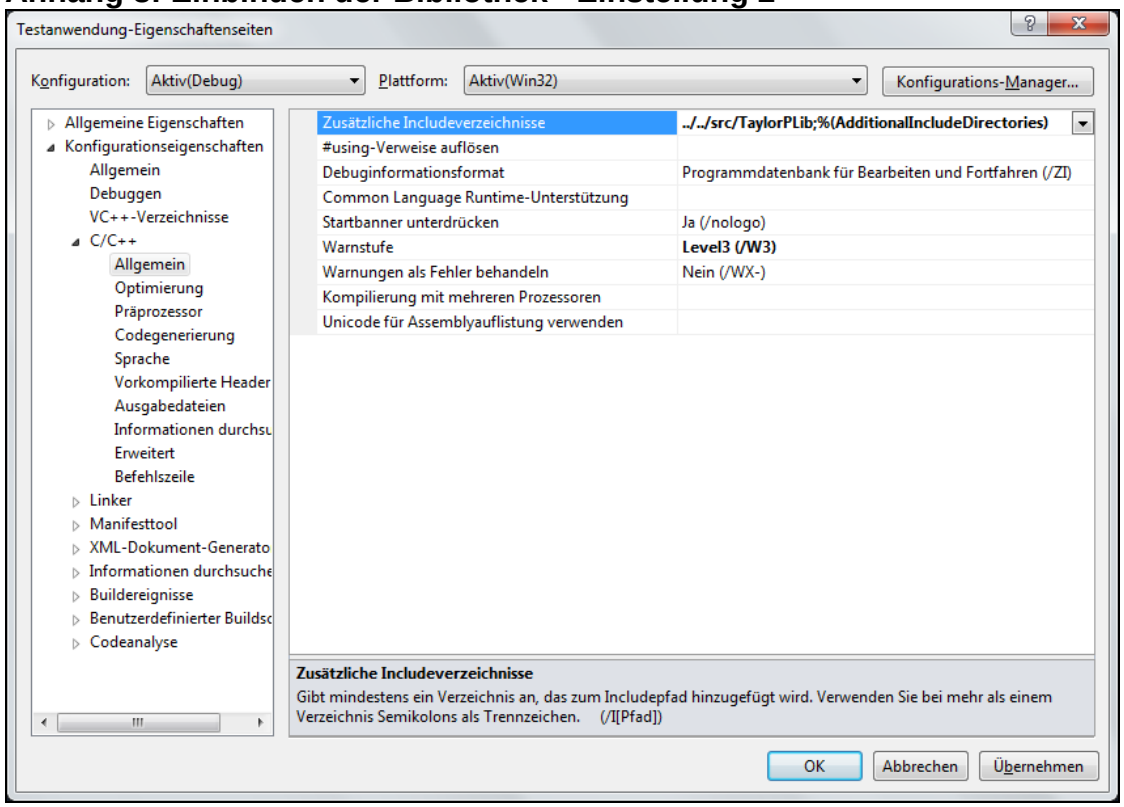
Anhang 6: Test - Code Coverage C#

Hierarchie	Nicht abgedeckt (Blöcke)	Nicht abgedeckt (% Blöcke)	Abgedeckt (Blöcke)	Abgedeckt (% Blöcke)
taylorplib_csharp.dll	0	0,00%	1336	100,00%
{ } LibMatrix	0	0,00%	1336	100,00%
MathException	0	0,00%	7	100,00%
MathException()	0	0,00%	2	100,00%
MathException(string)	0	0,00%	2	100,00%
what()	0	0,00%	3	100,00%
Matrix	0	0,00%	906	100,00%
Polynomial	0	0,00%	423	100,00%

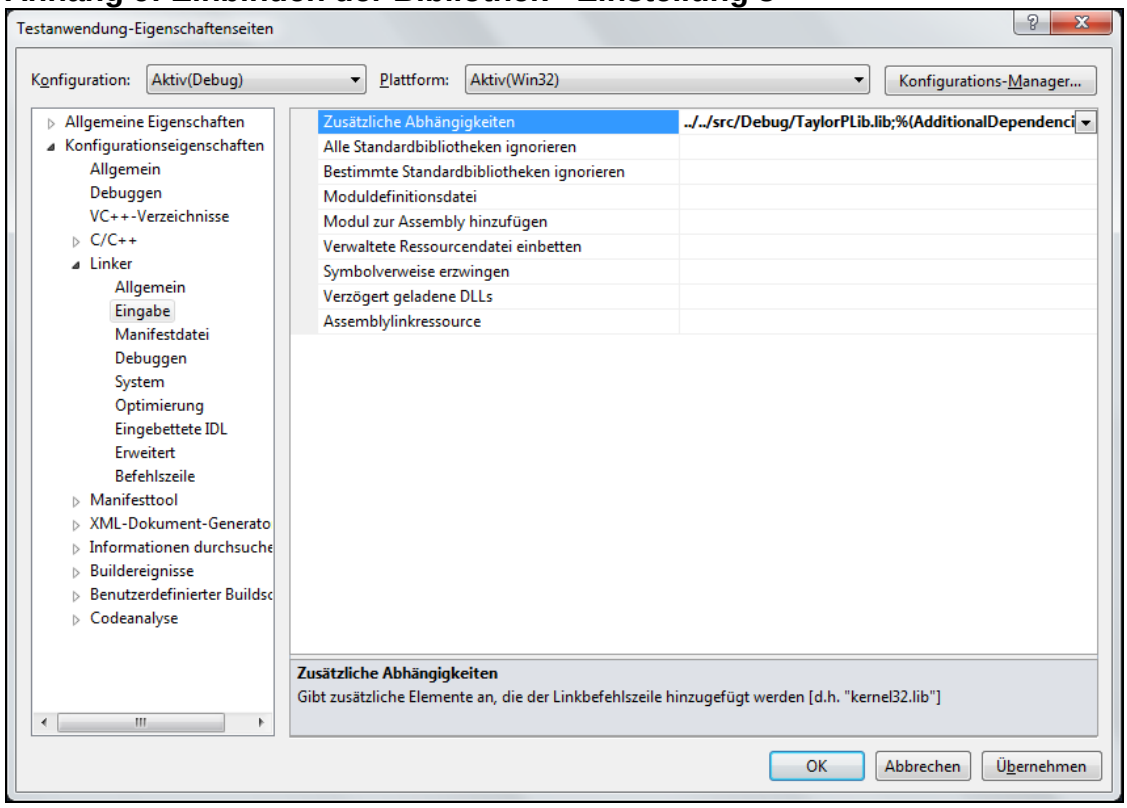
Anhang 7: Einbinden der Bibliothek - Einstellung 1



Anhang 8: Einbinden der Bibliothek - Einstellung 2



Anhang 9: Einbinden der Bibliothek - Einstellung 3



Anhang 10: TaylorPLib.i

```
/* File : TaylorPLib.i */
%module TaylorPLib

%{
#include "..\src\TaylorPLib.h"
%}

/* Support for vector<T> */

%include "std_vector.i"

namespace std {
    %template(DoubleArray) vector<double>;
    %template(IntArray) vector<int>;
    %template(PolynomialArray) vector<LibMatrix::Polynomial>;
}

%ignore LibMatrix::Polynomial::operator=(const Polynomial &p);

%rename (_print) print();
%rename (printToFile) print(FILE*);
%rename (printWithName) print(const char*);

%ignore LibMatrix::Matrix::operator=(const Matrix &p);
%ignore LibMatrix::Polynomial::operator[](int);

%ignore operator<<(std::ostream &out, const LibMatrix::Polynomial &p);
%ignore operator<<(std::ostream &out, const LibMatrix::Matrix &m);


/* include original header file */

%include "..\src\TaylorPLib.h"
```