

Aus dem Institut für Informationssysteme des Gesundheitswesens

Betreuer:

Hon. Prof. Dr. Roland Blomer

Modulare Delta Algorithmen für das praktische Versionsmanagement von Dokumenten

Masterarbeit

zur Erlangung des Titels

„Master of Science Biomedizinische Informatik“

der privaten Universität für Gesundheitswissenschaften, Medizinische Informatik und
Technik in Hall in Tirol

vorgelegt von

Matthias Gruber

aus

Feldkirch

Innsbruck, 2007



UMIT

private universität für gesundheitswissenschaften, medizinische informatik und technik
university for health sciences, medical informatics and technology

Zusammenfassung

In der vorliegenden Arbeit geht es um modulare Delta Algorithmen, die zum praktischen Versionsmanagement – wie beispielsweise in Dokumentenmanagement Systemen – eingesetzt werden. Diese Algorithmen können Speicher- und Transferkosten verringern, indem sie den Unterschied (Delta) zwischen verschiedenen Versionen einer Datei berechnen, so dass nicht jede Version der Datei vollständig auf Datenträgern gespeichert werden muss. Zu diesem Zweck wurde auch eigens ein gemanagtes Framework für Delta Algorithmen neu implementiert, und zwar vor allem weil vorhandene Softwarebibliotheken aufgrund von bekannt gewordenen Schwächen bei populären Hash Algorithmen keine kryptographische Sicherheit mehr bieten. Dies wird aber für den Einsatz insbesondere im medizinischen Umfeld vorausgesetzt. Das Framework wurde vom Autor mit dem Namen „SDelta“ (für „Secure Delta“) versehen.

In den einleitenden Kapiteln wird die Motivation, Problemstellung und Zielsetzung genauer erläutert. Im Anschluss werden die Techniken und Algorithmen erklärt, auf welchen die angefertigte Software beruht. Dazu gehören Hash Funktionen, Checksummen und zwei langjährig bewährte Delta Algorithmen. Der erste Algorithmus nach (Myers 1986) findet nachweislich stets ein kürzest mögliches Skript zur Überführung von einer Dokumentenversion in eine zweite. Dies ist gleichbedeutend mit dem Auffinden der längsten gemeinsamen Subsequenz und äußerst nützlich für von Menschen lesbare Daten, erfordert aber direkten (random) Zugriff auf beide Versionen und kann daher nicht effizient auf Streams operieren. Der zweite Algorithmus nach (Tridgell 1999) verarbeitet im Gegensatz dazu die Dateiversionen sequenziell und unabhängig voneinander, so dass er auf Streams und damit auch auf sehr große Dateien angewandt werden kann. Dies ist sogar noch praktikabel, wenn eine der Versionen nicht lokal vorliegt, also zum Beispiel nur über Netzwerk oder Internet erreichbar ist. Allerdings ist ein vom letzteren Algorithmus berechneter Delta gar nicht oder nur sehr schwer für Menschen lesbar.

Außerdem gehören zu den Grundlagen die Techniken zur Implementierung der Software, nämlich die Entwicklungsplattform .NET und die Programmiersprache C#, welche in Kapitel 2 beschrieben werden.

Anschließend werden die Methoden und Softwarewerkzeuge vorgestellt, mit deren Hilfe die beschriebenen Ziele erreicht wurden. In erster Linie sind das die Entwicklungsumgebung und eine Software Entwicklungsmethode, die vom so genannten „extreme programming“ abgeleitet wurde. Außerdem gehört Profiling sowie Dokumentations- und Testmethodik dazu.

Das SDelta Framework sowie dazu eigens angefertigte Demonstrations- und Testanwendungen werden im Folgenden unter „Ergebnisse“ detailliert beschrieben. Das SDelta Framework hat zum Zeitpunkt der Einreichung den Release Status erreicht und weist einsatztaugliche Performance auf, was am Ende des Kapitels im Abschnitt „Performance“ gezeigt wird. Die Ergebnisse werden anschließend unter „Diskussion“ kritisch beleuchtet und es wird ein Ausblick auf mögliche Erweiterungen und Verbesserungen gegeben. Die Software, Dokumentation und Testdaten können von <http://www.bluelogic.at> unter Downloads bezogen werden.

Am Ende folgen noch Verzeichnisse und Anhänge, sowie ein Glossar.

Summary

Topics of the presented work are modular delta algorithms, which are used in applied version management as in document management systems. These algorithms are used to save on memory and transfer costs by calculating the difference between two versions of a file so that only one version has to be saved fully on disk. Therefore a managed delta framework has been newly implemented. Main reason for the new implementation is that certain weaknesses have been discovered within popular hashing algorithms, which are used in current software. The consequence is that these products do not provide cryptographic security, which is required in a medical environment.

The presented framework will be referred to as „SDelta“ (for „Secure Delta“).

Motivation, problem definition and targets of this work are introduced in the starting chapters. These are followed by an explanation of required techniques and algorithms which are used in the software, including hash functions, checksums and two delta algorithms which are frequently and successfully applied in other software products.

The first of the two algorithms was invented by (Myers 1986) and has been proved to always find the shortest possible edit script for converting one dataset to another, which is equivalent to finding the longest common subsequence. The output is readable by humans and very useful in many cases but the algorithm requires random access to both versions of the data which prevents it from operating on streams.

The second algorithm by (Tridgell 1999) processes the files in a sequential order and in an independent way so that it can operate on streams and thereby also on very large files, while still being fast and efficient enough to operate over network or internet. Unfortunately, in this case the output is not readable by humans.

The last part of the second chapter covers the techniques used in the actual software implementation such as the programming language C# and the .NET framework.

In the following part the methods and software products which were used for the development of a solution are presented. These are mainly the development environment Microsoft Visual Studio and an excerpt of extreme programming methods. Furthermore, profiling, documentation and testing techniques are explained.

SDelta and the additional demonstration and testing tools are described in detail in the following chapter “Ergebnisse” (“results”). At the time of completion the software has reached release status and a practicable performance, which will also be described within this chapter. The results are further discussed in the following chapter “Diskussion” and finally there are some recommendations and ideas for further development of the software. The latest version of the software, including documentation and test material can be downloaded from <http://www.bluelogic.at> (download section).

The last chapters provide image and table indexes as well as a short glossary.

Inhalt

Zusammenfassung	II
Summary	IV
Inhalt.....	VI
1. Einleitung.....	1
1.1 Gegenstand	1
1.2 Motivation.....	2
1.3 Problemstellung	3
1.4 Zielsetzung.....	4
1.5 Fragestellung	4
1.6 Gliederung	4
2. Grundlagen.....	6
2.1 Hash Funktionen und Checksummen	6
2.2 Delta Algorithmen	12
2.3 Implementierung: CLI, .NET, C#	17
3. Methoden.....	26
3.1 Extreme Programming	26
3.2 Entwicklungsumgebung	31
3.3 Dokumentation	32
3.4 Testing	33
3.5 Profiling	42
4. Ergebnisse.....	44
4.1 Core Library	44
4.2 Delta Wizard	50
4.3 Diff Wizard	53
4.4 Performance	56
5. Diskussion	57

6. Ausblick	60
7. Danksagung	61
8. Verzeichnisse	62
8.1 Literatur.....	62
8.2 Abbildungen.....	65
8.3 Tabellen	66
8.4 Glossar.....	67
9. Lebenslauf.....	69
10. Anhänge	70
10.1 Anhang 1 – Text Delta	70
10.2 Anhang 2 – Binär Delta.....	72
10.3 Anhang 3 – Weitere Testergebnisse.....	75

1. Einleitung

1.1 Gegenstand

Ein Delta-Algorithmus ist eine Technik zur Erstellung eines gerichteten Transformationsskripts „Delta“ aus zwei verschiedenen Versionen eines Objektes, sodass die eine Version (Quelle) mit Hilfe des Delta in die andere Version (Ziel) überführt werden kann. Beispiele für Objekte sind Text- oder Codedateien, Binärdateien verschiedensten Ursprungs sowie genetischer Code aber auch komplexere Objekte im Sinne der objektorientierten Programmierung (OOP).

Das Skript besteht dabei aus einer Reihe von Anweisungen (Einfüge- und Kopieranweisungen bzw. Einfüge- und Löschanweisungen), die gegebenenfalls komprimiert werden. Es kann auch invertiert werden, was insbesondere für source code management (SCM) Software, aber auch anderweitige revision control Software, und generell Delta - basierte Dateisysteme wie XDFS (MacDonald 2000) wichtig ist, da in diesen Anwendungsfällen eine so genannte head revision vollständig gespeichert wird, während zu den älteren Versionen nur Deltas gehalten werden.

Spezielle Text - Deltas können dazu verwendet werden Unterschiede zu analysieren, wie dies in Team - Programmier - Projekten benutzt wird um Konflikte in gemeinsam bearbeiteten Code Dateien zu finden und zu bereinigen. Außerdem hilft die Methode auch Speicher- und Transferkosten zu verringern, was sie vor allem im Bereich der Versionierung (Branching, Tagging) von Dokumenten wie in Dokumentenmanagement Systemen und Dokumentationssystemen nahezu unverzichtbar macht. Auch bei der Distribution von Software Updates kann die Technik massiv Transferkosten einsparen, was sich zum Beispiel die Microsoft Windows Betriebssysteme zunutze machen. (Potter 2005)

1.2 Motivation

Gemessen an den zahlreichen Anwendungen gibt es nur eine sehr kleine Auswahl an patent- und lizenzenfrei verfügbaren Frameworks und Softwarebibliotheken zur Erzeugung, Speicherung und Verwaltung von Deltas. Besonders schwer verfügbar sind brauchbare Implementierungen für managed Plattformen wie Java oder .NET. In vielen Fällen ist es nur schwer nachzuvollziehen, welcher Algorithmus wie implementiert wurde, der Quellcode ist nicht erhältlich, unzureichend kommentiert oder schlecht implementiert und es mangelt an Dokumentation und Transparenz.

Eines der am weitesten verbreiteten, professionellen Pakete ist unter dem Namen xdelta (MacDonald 1998) bekannt. Dieses Produkt beruht auf rsync (Tridgell 1999) wobei der Algorithmus in erster Linie für ein anderes Anwendungsprofil optimiert, aber nicht grundlegend verändert wurde. xdelta (wie auch rsync) wird unter der so genannten „GNU General Public Licence“ (in Folge GPL) verbreitet (FSF 1991), was der wissenschaftlichen und kommerziellen Nutzung mehrere Steine in den Weg legt. Bekannte Kritikpunkte sind das Prinzip des „Copyleft“ an sich, der mangelnde Schutz der Urheberrechte und die Unvereinbarkeit mit Software oder Bibliotheken die anderen (auch freien) Lizenzen oder Patenten unterliegen. Insbesondere in Deutschland und Österreich kommen noch die effektive Unwirksamkeit des Haftungsausschlusses und die teilweise Gleichstellung mit „Allgemeinen Geschäftsbedingungen“ hinzu. (Sujecki 2005; Till Jaeger 2005)

Für xdelta wie für rsync sind aktuelle, gut kommentierte Quelltexte verfügbar. Die Pakete sind in C++ geschrieben, wobei eine Menge Pointerarithmetik und Template Einsatz ins Auge sticht. Deswegen bieten sie sich nicht gerade für eine Portierung nach Java oder .NET an. Außerdem würde eine solche Portierung auch wieder unter die GPL fallen.

Ein weiterer Punkt, der gegen sämtliche im Zuge dieser Arbeit begutachteten Delta-Pakete spricht ist die Abhängigkeit von bestimmten Hash Algorithmen. Häufig werden der MD5 Hash und der SHA-1 eingesetzt um das Ergebnis der Operation zu verifizieren. Jüngste Angriffe (Wang 2004; Wang 2005; Contini 2006; Stevens 2006) haben das Vertrauen in MD5 und SHA Hash Funktionen geschwächt, und es werden bereits Alternativen diskutiert (Bellovin 2005; Bellare 2006; Bentahar 2006; Gauravaram 2006; Gligoroski 2006; Halevi 2007). Keines der unten angeführten Pakete ist auf eine einfache Auswechselung des Hash Algorithmus ausgelegt.

Produkt	Algorithmus	Hash	Quelltext	Sprache	Plattform	Lizenz
rsync	Tridgell	SHA1	Ja	C	*	GPL
librsync	Tridgell	SHA1	Ja	C	*	GPL
xdelta	Tridgell	MD5	Ja	C	*	GPL
CVSup	Tridgell	-	Ja	Modula-3	Posix-OS	BSD
diff	Hunt / Tichy	-	Ja	C	*	Frei
diffutils	Myers	-	Ja	C	*	GPL
CSDiff	Text	-	Nein	?	win32	kommerziell
KDiff3	Text	-	Ja	C	*	GPL
windiff	Text	-	Ja	VC	win32	Frei
vdelta	Hunt / Tichy	-	Ja	C	*	Frei
vcdiff	binär	-	Ja	C	*	Frei
zdelta	binär	-	Ja	C	*	Frei
bdiff	Hunt / Tichy	-	Ja	C	*	Frei

Tabelle 1 – Verschiedene Delta Lösungen. Nur bei wenigen Produkten wird eine Hash Funktion dokumentiert, welche der Verifikation der Gesamtoperation dient. Dafür sind die meisten Produkte plattformunabhängig, insofern der zugehörige C Quellcode für die jeweilige Plattform kompiliert werden kann. Nur ein Teil der Pakete darf sowohl in Quelltext- als auch in -binärform frei weiterverbreitet werden. Mit „frei“ sind in diesem Fall Freeware oder nicht-restriktive Lizzenzen wie eine Creative Commons Lizenz (Commons 2007) gemeint.

1.3 Problemstellung

Ausgehend von diesen Vorüberlegungen ergeben sich folgende Problemstellungen:

- P1: Schlechte Verfügbarkeit von qualitativ hochwertigen, lizenzfreien Delta Implementierungen.
- P2: Schlechte Verfügbarkeit von managed Delta Implementierungen.
- P3: Veralterte Technologien: Binär - Delta Softwarepakete benutzen Hash Funktionen wie MD5 zur Verifikation der Rekonstruktionsoperation.
- P4: Mangelnde, effektive und einfache Austauschbarkeit von Hash Algorithmen bei den untersuchten Paketen.
- P5: Mangelnde Dokumentation, Erweiterbarkeit und Transparenz bei vielen der untersuchten Delta Paketen.

1.4 Zielsetzung

- Z1: Erstellen einer Übersicht über benötigte Algorithmen und Technologien.
- Z2: Implementierung eines O(ND) Text Delta nach (Myers 1986)
- Z3: Implementierung eines O(M+N) binär Delta Algorithmus nach (Tridgell 1999)
- Z4: Erstellen eines veröffentlichtbaren und lizenzfreien managed Delta Framework.
- Z5: Einfache Austauschbarkeit der eingesetzten Verifikations - Hashes.
- Z6: Testung und Validierung des Paketes.
- Z7: Analyse und Optimierung der Performance des Paketes.

1.5 Fragestellung

- F1: Welche Algorithmen und Technologien werden für die Implementierung benötigt?
- F2: Welche Rechenkomplexität besitzen die implementierten Algorithmen?
- F3: Welche Datenstrukturen werden für die Implementierung benötigt?
- F4: Was für eine Architektur soll für ein solches Framework gewählt werden?
- F5: Wie kann ein solches Delta Framework getestet und validiert werden?
- F6: Wie performant ist das Paket?
- F7: Welche Optimierungs- und Erweiterungsmöglichkeiten gibt es?

1.6 Gliederung

In weiterer Folge (Abschnitt 2) werden nun die Grundlagen erläutert, welche für das Verständnis der relevanten Techniken erforderlich sind. Dazu werden zuerst wichtige Begriffe aus dem Bereich der Hash – Funktionen und Checksummen erklärt. Im Anschluss wird kurz auf die Arbeiten von (Myers 1986) und (Tridgell 1999) eingegangen, welche die hier zum Einsatz gelangenden Delta Algorithmen behandeln. Im letzten Teil werden Grundlagen im Bereich der Implementierung (die Programmiersprache C# und das .NET Framework) besprochen.

Abschnitt 3 (Methoden) beinhaltet Details zur Implementierung und Testung. Es wird insbesondere auch eine genaue Beschreibung der Testdaten und -szenarien

herausgegeben. Außerdem werden die zum Einsatz gelangten Softwarepakete, Dokumentationsmethoden und Profiling Techniken vorgestellt.

Im folgenden Abschnitt 4 werden die Ergebnisse dieser Arbeit – eine Softwarelibrary und zwei Test- und Demonstrationsanwendungen – mit Hilfe von Klassendiagrammen und Quellcodeausschnitten vorgestellt und im Anschluss in Abschnitt 5 kritisch beleuchtet. Abschnitt 6 gibt einen Ausblick, welcher auch Optimierungs- und Erweiterungsmöglichkeiten beinhaltet. Nach der Danksagung (Abschnitt 7) folgen noch die Verzeichnisse und ein kurzer Lebenslauf.

In den Anhängen finden sich noch Beispiele zu den Delta Algorithmen und zusätzliche Testdaten.

2. Grundlagen

In diesem Kapitel werden wichtige Begriffe und Grundlagen erörtert, die für das Verständnis dieser Arbeit erforderlich sind. Nach einer kurzen Einführung in das Gebiet der Delta Algorithmen und der Hash Funktionen wird – soweit das für das Nachvollziehen der Arbeit nötig ist – noch die ausgewählte Programmiersprache C# und die Plattform .NET vorgestellt.

2.1 Hash Funktionen und Checksummen

Checksummen und so genannte “cyclic redundancy checks” (CRC), eine spezielle Form von Checksummen, unterscheiden sich grundlegend von kryptographischen Hash Funktionen. Für eine Checksumme ist eine gewisse Kollisionsresistenz ausreichend. Das bedeutet, es sollte schwierig sein, zwei Objekte o1 und o2 zu finden für die

$$\text{checksum}(o1) = \text{checksum}(o2)$$

gilt. An kryptographische Anwendungen werden allerdings höhere Anforderungen gestellt. Ein bekannter Fall, in dem ein CRC fälschlicherweise anstelle eines Hash eingesetzt wurde, ist der Wired Equivalent Privacy (WEP) Standard (IEEE 1999), welcher in Folge auch prompt „gekrackt“ wurde (Fluhrer 2001). Mittlerweile sind vollautomatische Softwarepakete erhältlich (zum Beispiel Airopeek NX) welche eine WEP „Verschlüsselung“ binnen Minuten knacken können.

In weiterer Folge werden kurz Grundlagen von Hash Funktionen erklärt. Danach wird noch eine spezielle Checksumme (Adler32) vorgestellt, die für den weiter unten beschriebenen binären Delta Algorithmus benötigt wird. Darüber hinaus wird nicht weiter auf Checksummen oder CRCs eingegangen, da dies nicht relevant für diese Arbeit ist.

2.1.1 Hash Funktionen

Neben der einfachen Anwendungsmöglichkeit als Checksumme haben Hash Funktionen einen wichtigen Anwendungsbereich in der digitalen Kryptographie; insbesondere sind elektronische Signaturen und Nachrichten - Authentifizierungsprotokolle (oft auch „message digest“ genannt) auf sichere Hash Funktionen angewiesen. Daher ist eine der wichtigsten wünschenswerten Eigenschaften für eine Hash Funktion, dass es schwierig sein soll, eine Kollision zu erzeugen. „Schwierig“ heißt in diesem Zusammenhang, es sollte genauso aufwändig sein, wie eine so genannte „brute-force“ Attacke, bei welcher einfach alle Möglichkeiten ausprobiert werden. (siehe Yuval 1979; van Oorschot 2004)

Es gibt keine formale, allgemein anerkannte Definition, welche die gewünschten Eigenschaften einer kryptographischen Hash Funktion festhält. Folgende Anforderungen werden aber im Allgemeinen als Mindestvoraussetzungen anerkannt:

- Collision Resistance (CR):

Es sollte schwierig sein, zwei Objekte o_1 und o_2 zu finden, für die

$$\text{hash}(o_1) = \text{hash}(o_2)$$

gilt. Eine „collision attack“ ist der Versuch zwei solche Objekte zu finden. Dies sollte bei einem CR Hash der Bitbreite n einen Aufwand der Größenordnung $2^{n/2}$ Operationen verursachen.

- Pre-image Resistance (PR):

Es sollte schwierig sein, zu einem Hash Wert h ein beliebiges Objekt o zu finden, so dass

$$h = \text{hash}(o)$$

gilt. Bei einer „pre-image attack“ versucht man genau dies. Ist der Hash PR, so sollte der Aufwand dafür $O(2^n)$ betragen. Aus der Sicherheitsperspektive ist dies noch ungünstiger als eine gelungene collision attack.

- Second Pre-image Resistance (SPR):

Es sollte schwierig sein, zu einem gegebenen Objekt o_1 ein weiteres Objekt $o_2 \neq o_1$ zu finden, so dass

$$\text{hash}(o_1) = \text{hash}(o_2)$$

gilt. Dies führt zu einer zweiten Variante der pre-image attack, die bei einem SPR Hash 2^n Operationen kosten sollte.

Um die Anforderungen für CR zu erfüllen, muss der Hash also doppelt so stark sein wie für PR und SPR. Von diesem Effekt stammt der alternative Name „birthday attack“, der in der Literatur oft für die collision attack benutzt wird¹. Balanciertheit ist eine weitere Anforderung, die sich aus den oben genannten ergibt (siehe auch Bellare 2002).

¹ Der Name stammt von dem Umstand, dass in einer zufällig gewählten Stichprobe von 23 Menschen die Wahrscheinlichkeit bereits über 50 Prozent beträgt, dafür dass zwei der Personen am selben Tag Geburtstag haben.

Auch wenn eine Hash Funktion diese Kriterien erfüllt, so ist noch nicht sichergestellt, dass keine anderen unerwünschten Eigenschaften vorliegen: Ein Beispiel dafür ist die „Length Extension Attack“, die benutzt werden kann um bestimmte naive Authentifizierungsschemas zu brechen: bei bekanntem Hash und bekannter Länge von einem unbekannten Objekt o wird nach einem o' gesucht, so dass ein Angreifer hash(o + o') berechnen kann, wobei „+“ hier eine Konkatenation darstellt.

In der aktuellen Literatur (siehe auch Gligoroski 2006; Hirai 2006) werden vor allem zwei Klassen von Anforderungsprofilen für Hash Funktionen betrachtet: Kollisionsresistente Hash Funktionen (kurz CRHF²) und universelle Einweg – Hash Funktionen (kurz UOWHF³). Für eine CRHF $h: \{0, 1\}^* \rightarrow \{0, 1\}^n$ muss gelten, dass die rechnerischen Kosten einer vollen Kollisionsattacke größer als $2^{n/2}$ sind. Die Hashes der SHA-2 Familie (SHA-224, SHA-256, SHA-384, SHA-512) fallen nach derzeitigem Wissensstand in diese Klasse. Leider ist es schwer die Sicherheit von CRHF für Signaturanwendungen zu beweisen und in letzter Zeit wurde für einige vermeintliche CRHF wie in der Einleitung bereits erwähnt das Gegenteil gezeigt. An UOWHF werden daher etwas weniger strenge Anforderungen gestellt, um ein beweisbares Signaturschema (Hirai 2006) zu erhalten. Es handelt sich dabei um Hash Funktionen, die mit einem zufälligen Initialisierungsvektor oder Key k initialisiert werden, so dass für ein beliebiges Objekte o1 schwer ein Objekt o2 $\neq o1$ zu finden ist, für welches

$$\text{hash}(k, o1) = \text{hash}(k, o2)$$

gilt. Nachteile sind der höhere Rechenaufwand (Performance) und die Key Größe. (Hirai 2006)

CRHF sollen nicht tatsächlich irreversibel sein, dies ist ein häufiges Missverständnis. Für kryptographische Sicherheit muss die Funktion eine Permutation sein, welche ihren Zustand bijektiv transportiert. Irreversibilität wäre ein Zeichen für lokale Kollisionen, welche Angriffe erleichtern können, wie dies im Fall der MDx und SHA Hash Familien passierte. Nach ersten frühen Warnungen (Bellare 2002) wurden fortlaufend bessere Methoden gefunden, Kollisionen bei einigen wichtigen Hash Funktionen zu erzeugen, darunter der stark verbreitete MD5 sowie MD4, SHA-0, HAVAL-128 und RIPEMED (Wang 2004). In weiterer Folge wurden die Angriffe erfolgreich auf SHA-1 (Wang 2005) erweitert. SHA-1, MD5 und RIPEMED-160 sind unter den am häufigsten benutzten Hash

² Aus dem Englischen „Collision Resistant Hash Function“

³ Aus dem Englischen „Universal One-Way Hash Function“

Funktionen (NIST 2006). Die folgende Tabelle gibt eine Übersicht über den Status zum Zeitpunkt der Verfassung dieser Arbeit (März 2007):

Hash	Bitbreite	Angriff	Operationen
MD4	128	Kollision	< 10
MD5	128	Kollision	$<< 2^{64}$
SHA-0	160	Kollision	2^{40}
SHA-1	160	Kollision	2^{63}
SHA-2	224-512	dzt. nur brute-force	$> 2^{112}$
RIPEMED	128	Kollision	$<< 2^{64}$
HAVAL	128	Kollision	2^6
HAVAL	160	Kollision	$<< 2^{80}$

Tabelle 2 – Verschiedene bekannte Hash Funktionen mit Bitbreite. Alle aufgelisteten Hash Funktionen außer der SHA-2 Gruppe weisen eine Sicherheitslücke auf, durch welche ein potentieller Angreifer zwei Objekte oder Nachrichten mit dem selben Hash Wert finden kann, ohne dazu die Rechenkosten einer vollen Kollisionsattacke (brute force) in Kauf nehmen zu müssen.

Eine Bitbreite von 64 Bits entspricht einer Anzahl von 2^{64} ($= 1.8 \cdot 10^{19}$) verschiedenen möglichen Hashes. Um alle Werte durchzuprobieren (PR oder SPR Angriff) würden bei 10^6 Hash Berechnungen pro Sekunde (entspricht in etwa der Leistung eines mittleren Firmennetzwerks von ca. 100 modernen Desktop PC's bei der herkömmlichen Berechnung von MD5 Hashes) beinahe dreihundert-tausend Jahre vergehen. Bei 10^9 Hashes pro Sekunde sinkt diese Zeitspanne auf nur ungefähr hundert Tage. Diese Leistung ist heute von einem Angreifer mit großen Ressourcen durch Supercomputing oder massives Clustering durchaus erreichbar. Auch eine low cost Attacke, die in der Leistung irgendwo zwischen den beiden Werten anzusiedeln ist wäre mit Hilfe von Grid Computing denkbar. Im Extremfall könnte dies von einer einzigen Person unter Einsatz eines speziellen Computerwurms nach dem Vorbild von SETI@home durchgeführt werden. Für eine 128 Bit breite Hash Funktion beträgt die Zeitspanne bei 10^9 Hash Berechnungen pro Sekunde bereits $5.4 \cdot 10^{18}$ Jahre, was das Alter des Universums nach derzeitigen Schätzungen um einen Faktor von ca. einer Milliarde übersteigt. (Haenni 2006) Solange Quantencomputer mit entsprechend großer Qubit Breite nicht praktikabel werden, sollte dies sicher genug sein.

Das NIST reagierte Ende Oktober 2005 mit einem Workshop um die Implikationen der Attacke nach Wang zu untersuchen. Es wurde festgestellt, dass die rechnerischen Kosten für eine Kollisionsattacke zwar sehr hoch, aber für einen Angreifer mit den

entsprechenden Ressourcen durchaus praktikabel sein könnten und gab eine neue Police bezüglich Hash Funktionen heraus. Darin wird verlautbart, dass die SHA-2 Funktionen für federal agencies weiterhin sicher genug sind (NIST 2006). SHA-1 soll jedoch bis spätestens 2010 nicht mehr für digitale Signaturen, digitale Zeitstempel und andere Anwendungen, die Kollisionsresistenz erfordern eingesetzt werden. Ausgenommen sind Hash basierende Nachrichten - Authentifizierungscodes („HMACs“), Schlüssel - Ableitungsfunktionen (key derivation functions, kurz KDFs) und Zufallsgeneratoren (random number generators, kurz RNGs) (siehe auch NIST 2006 online) Dieser etwas zögerlichen Reaktion schließt sich der Autor nicht an und empfiehlt stattdessen möglichst sofort von der Verwendung der erwähnten Hash Funktionen abzusehen. Auch die Funktionen der SHA-2 Familie stellen möglicherweise nur eine Übergangslösung dar. (Hawkes 2004)

2.1.2 Eine “Rolling Checksum”

An dieser Stelle soll kurz die so genannte Adler-32 Checksumme (benannt nach Mark Adler) besprochen werden, da sie essentiell für binäre Delta Algorithmen wie in (Tridgell 1999) und (MacDonald 1998) ist. Sie basiert auf der Fletcher Checksumme, einer schwächeren Variante, und wurde auch im SCTP (Stream Control Transmission Protocol) eingesetzt, bis klar wurde, dass Kollisionen bei kurzen Datensätzen (< 128 Byte) für diese Anwendung zu häufig sind. (Stone 2002)

Man erhält die Adler-32 Checksumme indem man zwei 16-Bit Checksummen A und B zu einer 32-Bit Integer Zahl aneinanderhängt. A ist die Summe aller n Bytes in Datenquelle D und B die Summe der individuellen Ergebnisse für A aus jedem Schritt, jeweils modulo 65512 (größte Primzahl < 2^{16}).

Der Adler-32 Algorithmus

$$A = (1 + D[1] + D[2] + \dots + D[n]) \bmod 65521$$

$$\begin{aligned}B &= ((1 + D[1]) + (1 + D[1] + D[2]) + \dots + (1 + D[1] + \dots + D[n])) \bmod 65521 = \\&= (nD[1] + (n - 1)D[2] + (n - 2)D[3] + \dots + D[n] + n) \bmod 65521\end{aligned}$$

$$\rightarrow \text{Adler-32}(D) = B \times 65536 + A$$

Tabelle 3 – Der Adler-32 Algorithmus. Dieser Algorithmus liefert eine Checksumme für ein Array von Daten (hier D). Die Checksumme ist für den binären Delta Algorithmus von besonderer Bedeutung, weil sie für aufeinander folgende, überlappende Datensegmente besonders einfach berechnet werden kann.

Die algorithmische Bestimmung kann durch Aufschieben der Modulo - Operation um 5552 Byte beschleunigt werden, wie (Deutsch 1996) und (Tridgell 1999) zeigen.

Wie man leicht sehen kann gilt für einen String S aus n Bytes, die von S[1] bis S[n] adressiert sind mit A und B wie oben:

$$A(S[2..n]) = A(S[1..n-1]) - S[1] + S[n] \bmod 65521$$

$$B(S[2..n]) = B(S[1..n-1]) - n * S[1] + A(S[2..n]) - 1 \bmod 65521$$

Somit kann man aus der Adler-32 Checksumme für die Bytes 1 bis n sehr leicht die Adler-32 Checksumme für die Bytes 2 bis n + 1 bestimmen, was für das binäre Delta Encoding im nächsten Kapitel (Abschnitt 2.2.2) sehr wichtig ist und wohl ausschlaggebend für die Bezeichnung „rollende“ Checksumme war.

Im Vergleich zur bekannten CRC32 Checksumme ist Adler32 etwas schwächer in Bezug auf Kollisionsresistenz, insbesondere bei Datenmengen < 1024 Byte, dafür aber wesentlich schneller zu berechnen. (siehe auch Jean-Loup Gailly 2006)

2.2 Delta Algorithmen

Wie bereits in der Einleitung (Abschnitt 1.1) erwähnt, dient ein Delta Algorithmus im Sinne dieser Arbeit dazu, ein gerichtetes Transformationsskript zwischen zwei in der Regel ähnlichen Datenquellen zu erzeugen, so dass aus der einen Version der Daten (Quelle) die andere (Ziel) erzeugt werden kann. Dazu werden je nach Anwendung zwei prinzipiell verschiedene Arten von Transformationsskripts eingesetzt, die in weiterer Folge als „Text - Delta“ beziehungsweise „Binär - Delta“ bezeichnet werden. Der Grund für diese Fallunterscheidung wird in den folgenden Abschnitten erklärt.

2.2.1 Text - Delta

Ein Text - Delta (auch „Edit Script“ genannt) besteht aus einer Folge von Einfüge- und Löschbefehlen. Die Erzeugung lässt sich am besten mit Hilfe eines Diagramms („Edit Graph“) veranschaulichen. Dazu werden die Quell- und Zieldaten orthogonal zueinander angeordnet, so dass die Untereinheiten der Quelldaten als N Spalten und die der Zieldaten als M Zeilen einer Tabelle dargestellt werden:

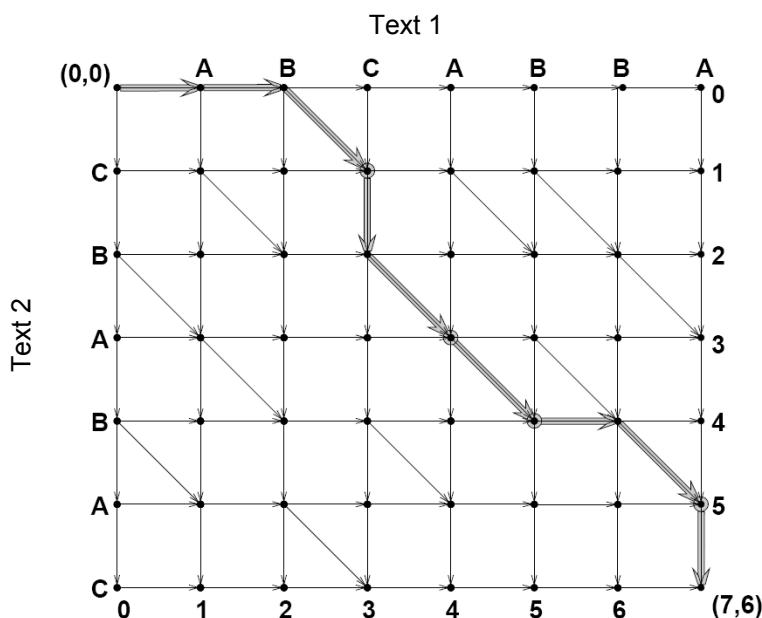


Abbildung 1 – Ein Edit Script aus (Myers 1986). Text 1 („ABCABBA“, horizontal) und Text 2 („CBABAC“, vertikal) spannen ein Raster auf, das auch als Baum interpretiert werden kann. Übereinstimmende Zeichen („Match Points“) sind durch diagonale Kanten im Graphen gekennzeichnet. (Myers 1986) zeigt auf, wie der kürzeste Weg durch einen solchen Graphen berechnet werden kann. Ein solcher Weg entspricht auch einer Reihe von Befehlen, mit welchen Text 1 in Text 2 überführt werden kann (indem horizontale Kanten als Lösch- und vertikale Kanten als Einfügeoperationen interpretiert werden) sowie umgekehrt.

Wie in der Abbildung zeichnerisch angedeutet, kann die Tabelle auch als gerichteter Graph mit den Tabellenfeldern als Knoten interpretiert werden. Horizontale Kanten können dann als Einfügeoperationen und vertikale Kanten als Löschoperationen für ein Edit Script interpretiert werden. Damit kann N aus M und einer Folge von Einfüge- und Löschoperationen rekonstruiert werden. Eine Einfügeoperation bedeutet, dass nach der angegebenen Position n das angegebene Zeichen z_n aus N in die Rekonstruktion von M eingefügt werden soll. Eine Löschoperation bedeutet, dass das Zeichen an der Stelle m in M nicht in die Rekonstruktion von N eingefügt wird. Die Positionsangaben beziehen sich dabei stets auf die Originaldaten und müssen so interpretiert werden, als ob sämtliche Operationen zur selben Zeit durchgeführt würden. Offensichtlich kann man das Edit Script auch invertieren, indem man einfach die Bedeutung von Lösch- und Einfügeoperationen umkehrt. (Siehe 10.1 – Anhang 1)

Die Felder, in denen Quelle und Ziel übereinstimmen werden als „match points“ bezeichnet. Diese verfügen über eine eingehende diagonale Kante, so dass zwischen einem match point (x, y) und dem Knoten $(x-1, y-1)$ keine Operation erfolgen muss. Der kürzeste Weg zwischen $(0, 0)$ und (N, M) stellt sich dann als der Pfad dar, welcher mit den wenigsten Operationen auskommt. Es bietet sich nun an, den Kanten entsprechend Kosten zuzuweisen, nämlich 1 für horizontale und vertikale Kanten und 0 für diagonale Kanten. Um ein Edit Script zu erzeugen, versucht man den kürzesten Weg zwischen $(0, 0)$ und (N, M) in Bezug auf die Kosten zu finden – also den Weg, der über möglichst wenige horizontale und vertikale Kanten führt. Eine solcher Weg wird in weiterer Folge als kürzestes Edit Script bezeichnet, wobei durchaus mehrere kürzeste Edit Scripts innerhalb eines Edit Graphen existieren können. In (Myers 1986) wird eine Methode vorgestellt, wie ein kürzestes Edit Script in $O((N+M)D)$ Zeit mit konstantem Speicherbedarf $O(N+M)$ gefunden werden kann. (Siehe 10.1 – Anhang 1)

Eine praktische Eigenschaft von zeilenweise berechneten Text – Deltas (wie für Quellcode Dateien in VCS Software üblich) ist, dass man diese anschaulich darstellen kann, indem man einfach die beiden Versionen eines Textes nebeneinander in zwei Textfeldern darstellt und die Zeilen entsprechend den match points zueinander anordnet.

```

File A: C:\Users\Matthias\Desktop\Development\test\v1\html.txt
File B: C:\Users\Matthias\Desktop\Development\test\v2\html.txt

1 offset: 479368 size: 28631
2 HTTP/1.1 200 OK
3 Date: Tue, 14 Aug 2001 01:06:09 GMT
4 Server: Apache (Debian Linux)
5 Cache-control: no-cache, age=0
6 Pragma: no-cache
7 Expires: Tue, 14 Aug 2001 01:06:09 GMT
8 Last-Modified: Tue, 14 Aug 2001 01:06:09 GMT
9 Connection: close
10 Content-Type: text/html; charset=US-ASCII
11
12 <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
13 <!-- $Id: clanek1.template,v 1.7 2001/04/30 13:52:41 enk Exp $ -->
14
15 <HTML>
16 <HEAD>
17 <TITLE>Novinky: Jak nas vidi pres ocean Jan Krondl z New Yorku</TITLE>
18 </HEAD>
19
20 <BODY bgcolor="#008000" link="#000080" vlink="#0000A4" text="#000000">
21
22 <!-- -->
23
24 <table width="610" border="0" cellspacing="0" cellpadding="0" bgcolor="#FFFFFF">
25

1 offset: 472391 size: 28791
2 HTTP/1.1 200 OK
3 Date: Sat, 27 Oct 2001 20:14:42 GMT
4 Server: Apache (Debian Linux)
5 Cache-control: no-cache, age=0
6 Pragma: no-cache
7 Expires: Sat, 27 Oct 2001 20:14:42 GMT
8 Last-Modified: Sat, 27 Oct 2001 20:14:42 GMT
9 Connection: close
10 Content-Type: text/html; charset=US-ASCII
11
12 <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
13 <!-- $Id: clanek1.template,v 1.7 2001/04/30 13:52:41 enk Exp $ -->
14
15 <HTML>
16 <HEAD>
17 <TITLE>Novinky: Jak nas vidi pres ocean Jan Krondl z New Yorku</TITLE>
18 </HEAD>
19
20 <BODY bgcolor="#008000" link="#000080" vlink="#0000A4" text="#000000">
21
22 <!-- -->
23
24 <table width="610" border="0" cellspacing="0" cellpadding="0" bgcolor="#FFFFFF">
25

```

Abbildung 2 – Zwei teils unterschiedliche HTML Dateien. Die Zeilen sind so angeordnet, dass übereinstimmende Zeilen nebeneinander stehen (grün markiert). Um Text 1 (linke Seite) in Text 2 (rechte Seite) zu überführen, müssen auf der linken Seite alle rot markierten Zeilen gelöscht werden und alle nicht markierten Zeilen der rechten Seite an den entsprechenden Positionen eingefügt werden.

2.2.2 Binär - Delta

Ein Binär - Delta ist eine Folge von Kopier- und Einfügebefehlen, die zur Rekonstruktion einer Datei aus einer anderen Version der Datei benutzt werden kann. Tridgell (Tridgell 1999) stellt eine Methode vor, wie ein solcher Binär - Delta in $O(N+M)$ Zeit erzeugt werden kann. Dies wird nun anhand des folgenden Szenario erklärt: Person 1 (im Folgenden „Empfänger“) verfügt über eine Datei A, aus welcher eine neuere Version – Datei B beim „Sender“ – erzeugt werden soll. Dazu teilt der Empfänger seine Version in nicht überlappende Blöcke von konstanter Größe S und übermittelt einen Hash (hier MD4) sowie eine rechnerisch wenig aufwändige Checksumme (hier Adler-32) an den Sender. Dieser erhält also Hash und Checksumme für $A[0..S]$, $A[S..2S]$, $A[2S..3S]$ und so weiter. Diese Daten werden in der zur Arbeit gehörigen Software als „BlockedHashSet“ (geblocktes Hash Set) zusammengefasst. Der Sender berechnet dann die Checksumme für alle Blöcke der Größe S für alle Offsets, also für $B[0..S]$, $B[1..S+1]$, $B[2..S+2]$ und so

weiter. Einer speziellen Eigenschaft von „rolling checksums“ (siehe Abschnitt 2.1.2) ist es zu verdanken, dass dies rechnerisch nicht sehr aufwändig ist: Wenn die Checksumme C für X[0..n] bekannt ist, so kann nur daraus und mit X[0] sowie mit X[n+1] auf einfache Art und Weise C für X[1..n+1] berechnet werden.

Der Sender vergleicht nun seine (schwachen) „rolling checksums“ mit denen vom Empfänger um Übereinstimmungen zu finden. Diese werden im Falle einer Übereinstimmung noch mit der Hashfunktion verifiziert, da die „rolling checksum“ nicht besonders kollisionsresistent ist. Mit diesen Informationen kann der Sender nun ein Delta Skript erstellen, mit welchem der Empfänger aus seiner Version A eine exakte Kopie von Version B erstellen kann. Die Blockgröße S ist dabei zugleich eine Grenze für die kleinstmöglichen erkennbaren Wiederholungen. (Siehe 10.2 – Anhang 2)

Die benutzte Hashfunktion muss nicht allen Anforderungen an eine kryptographische Hashfunktion gerecht werden, stattdessen ist in diesem Fall aufgrund der möglichen hohen Anzahl von Berechnungen wichtiger, dass sie rechnerisch effizient bestimmt werden kann. Das Gesamtergebnis sollte dann allerdings sehr wohl mit einem „sicheren“ Hash verifiziert werden, welcher zusätzlich übermittelt wird. Somit sollte bei einer Implementierung darauf geachtet werden, dass diese zur Unterscheidung von anderen Hash Funktionen in Folge „Digest“ genannte Hash Funktion leicht austauschbar gemacht wird. Dies gilt aber genauso auch für den anderen Hash und die Checksumme. So ließe sich mit einer 64-Bit rollenden Checksumme nach dem Muster der Adler-32 Checksumme die Blockgröße S verdoppeln, wenn kurze äquivalente Abschnitte in den Daten nicht relevant sind. Dies würde den Algorithmus beschleunigen, da weniger Modulo - Operationen benötigt würden. Auch für einen Austausch der MD4 Checksumme könnten sich Gründe ergeben, wie zum Beispiel bei einem erhöhten Auftreten von Kollisionen für ein besonders ungünstiges Datenformat. Auch eine schwächere Checksumme als der MD4 kann eingesetzt werden (Tridgell 1999), wenn beispielsweise die Rechenzeit sehr teuer ist, so dass eine einfachere Funktion gewählt werden muss.

Es sollte noch erwähnt werden, dass im Gegensatz zu Text Deltas nach Myers in diesem Fall keine für Menschen leicht erfassbare und interpretierbare Darstellung erzeugt werden kann, da man dazu die kopierten Blöcke anhand ihrer Offsets erkennen müsste. Bei sehr kleinen, wenig unterschiedlichen Dateipaaren könnte man vielleicht mit einer Farbcodierung der Adressen eine verständliche Visualisierung erzeugen, sofern das Dateiformat an sich für Menschen lesbar ist, aber bei zunehmender Datenmenge oder

Unterschiedlichkeit geht auch in diesem Fall die „Lesbarkeit“ verloren, so dass in der zu dieser Arbeit zugehörigen Software auf eine Visualisierung von Binär Deltas verzichtet wurde.

2.2.3 Gegenüberstellung Text und Binär Delta

Text - Deltas, bestehend aus Einfüge- und Löschoperationen, haben den großen Vorteil, dass sie ein für Menschen lesbares und leicht interpretierbares Output erzeugen. Außerdem findet der vorgestellte Algorithmus immer das kürzest mögliche Edit Script, gleichbedeutend mit dem besten möglichen „alignment“ zwischen zwei Datenabschnitten, was in der Bioinformatik von großer Bedeutung ist. Der Nachteil ist die inhärent höhere Rechenkomplexität (Hunt 1998). Außerdem sind Text - Deltas nicht optimal im Sinne des Speicherverbrauchs, da sie Einfüge- und Löschkosten gleich gewichten, obwohl Einfügeoperationen mehr Platz brauchen.

Binär - Delta Algorithmen benutzen Zeichenketten - Vergleiche um Stellen der Übereinstimmung zwischen Quell- und Zieldaten zu identifizieren. Für äquivalente Abschnitte werden entsprechende Kopierbefehle emittiert, für „neue“ Abschnitte Einfügebefehle. Der im obigen Abschnitt vorgestellte Algorithmus wird in verschiedenen Variationen vielerorts für Online Updates, Versionskontrolle und dergleichen eingesetzt und eignet sich grundsätzlich für Dateien aller Art. (MacDonald 2000)

Ein Text-Delta sollte also nur unter zwei Bedingungen eingesetzt werden:

1. Es liegen reine Textdaten vor.
2. Das Ergebnis muss für Menschen lesbar sein.

Andernfalls ist ein Binär-Delta zu benutzen, insbesondere wenn Prozessorzeit oder Arbeitsspeicher ein wichtiger Faktor sind. Diesen Umständen wird in VCS Systemen meist Rechnung getragen, indem ein Text - Delta erst bei Bedarf zusätzlich berechnet wird.

2.3 Implementierung: CLI, .NET, C#

Wie bereits erwähnt sollen die besprochenen Funktionen in managed code implementiert werden. Die Vorteile von managed code liegen vor allem in der hohen Produktivität und den eingeschränkten Fehlermöglichkeiten, welche Hauptkriterien bei der Erstellung von entsprechenden Programmiersprachen waren. Dazu kommt noch die Plattform – Unabhängigkeit, die erlaubt, ein und denselben Code für unterschiedliche Plattformen zu kompilieren. Nachteile sind eine teilweise geringere Performance, längere Startup Zeiten (Kompilierung der benötigten Klassen durch die „java virtual machine“ bzw. den „just-in-time compiler“ bei .NET) und die teilweise umständliche Interaktion mit native code, wofür das java native interface (JNI) nach Meinung des Autors ein gutes Beispiel darstellt.

Die Entscheidung zwischen Java und .NET wird oft leichtfertig als Geschmacksfrage abgetan. Davon wird dringend abgeraten, da die Unterschiede schon auf einer sehr grundlegenden Ebene beginnen und teilweise beträchtlich sind. In anderen Fällen wird die Diskussion zur philosophischen Grundsatzdiskussion erhoben, was genauso wenig produktiv ist und keinesfalls zu einer eindeutigen, allgemeingültigen Antwort führen kann. Oft spielen auch finanzielle Aspekte eine Rolle, da insbesondere Microsoft Produkte wie die Entwicklungsumgebung Visual Studio nicht gerade billig sind, während mit der Entwicklungsumgebung Eclipse eine recht ausgereifte Lösung für Java gratis verfügbar ist. Dies konnte hier vernachlässigt werden, da der Autor ohnehin über die benötigten Lizenzen verfügt. Im Falle der vorliegenden Arbeit gibt es eine Reihe von technischen Argumenten, die schließlich zur Entscheidung für die Sprache C# auf der .NET Plattform führten:

- Das .NET Framework bietet eine umfangreiche, Einsatzerprobte Kryptographielibrary, die einen großen Anteil der benötigten Hash Funktionen – insbesondere die SHA-2 Familie – bereits implementiert hat.
- Die reichhaltige und professionelle Entwicklungsumgebung Microsoft Visual Studio 2005 – hier um einige Plugins erweitert – ermöglicht komfortables und schnelles Arbeiten mit den verschiedenen .NET Sprachen.
- Die Sprache C# benutzt eine C-ähnliche Syntax, die auch für C++ und Java Programmierer verständlich ist.

- Das Generics Konzept ist in C# nach Meinung des Autors besser implementiert worden als in Java. (Dafür stellt Java allerdings im Gegensatz zum .NET Framework eine generische Hashtable zur Verfügung.)
- C# verfügt im Gegensatz zu Java über Events und Delegaten sowie Properties, was nach Meinung des Autors enorm zur Übersichtlichkeit und Verständlichkeit des Codes beiträgt.

Im Folgenden wird also das .NET Framework und im Anschluss die Programmiersprache C# kurz vorgestellt.

2.3.1 Das .NET Framework und die Common Language Infrastructure

Das .NET Framework von Microsoft ist die bekannteste Implementierung der Common Language Infrastructure (kurz CLI, aktuell ist Edition 4 seit Juni 2006), ein ISO/IEC/ECMA Standard (ECMA 335), der Systeme spezifiziert, die sprach- und plattformneutrale Softwareentwicklung und -ausführung ermöglichen sollen. Es gibt aber auch CLI-Implementierungen für Unix/Linux-Systeme, Mac OS X und BSD-Varianten. (Novell 2007)

Der Standard gliedert sich in sechs Teile:

Partition I: Konzepte und Architektur – Beschreibt die Gesamtarchitektur der CLI, spezifiziert Common Type System (CTS), Virtual Execution System (VES) und die Common Language Specification (CLS).

Partition II: Metadatendefinition und Semantik – Enthält Informationen über Metadaten: Das physische Layout der Dateien, die logischen Inhalte und deren Struktur.

Partition III: Beschreibt die Instruktionen der Common Intermediate Language (CIL).

Partition IV: Bibliotheken – Enthält eine Spezifikation von Klassen und Klassenbibliotheken, die als Teil der CLI standardisiert sind.

Partition V: Beschreibt das einheitliche Debuggingformat.

Partition VI: Anhänge.

Eine Reihe von namhaften Firmen und Universitäten haben an der Entwicklung des Standards mitgewirkt, darunter Borland, Microsoft, Intel, IBM, ISE, Sun Microsystems, die IT Universität in Kopenhagen, Monash University, Plum Hall und die University of Canterbury. (ECMA 335)

.NET ist eine Entwicklungsplattform, die verschiedene Betriebssystem-Funktionen zusammenfasst und anbietet und als veraltet geltende Technologien und Vorgehensweisen, wie zum Beispiel COM- oder API - Aufrufe im Programmcode minimieren soll. Sie besteht neben einer Laufzeitumgebung aus einer Sammlung (Framework) von Klassenbibliotheken (API) und aus angeschlossenen Diensten. Die .NET-Plattform stellt mit der Common Language Infrastructure (CLI) eine Basis zur Ausführung von Programmen zur Verfügung, die mit den verschiedenen kompatiblen Programmiersprachen erstellt wurden. Dies wird durch die Verwendung einer objektorientierten virtuellen Maschine wie in Java und der Framework Class Library (FCL, die gemeinsame Klassenbibliothek) erreicht. Die virtuelle Maschine nennt sich in diesem Fall Common Language Runtime (CLR) und kann als Gegenstück zur Java Virtual Machine gesehen werden. Die VM führt den Zwischencode der Common Intermediate Language (CIL) aus, ähnlich wie Java Bytecode ausführt. Die Common Language Specification (CLS) definiert eine gemeinsame Untermenge in CIL, der von der virtuellen Laufzeitumgebung (VM) in den Maschinencode der Zielmaschine übersetzt und ausgeführt werden kann. Dadurch ist es möglich, in .NET mit verschiedenen, an die CLR angepassten Sprachen zu programmieren. Je nach Distribution werden in Microsofts Entwicklungsumgebung Visual Studio Sprachen wie C#, C++, Visual Basic .NET, sowie J#, eine alternative Java-Implementierung von Microsoft, mitgeliefert. Um ein reibungsloses Zusammenspiel beim Aufruf von in einer anderen Sprache geschriebenen Komponenten zu gewährleisten existiert ein vereinheitlichtes, Sprachübergreifendes System von Objektbasierten Typen (Common Type System), das eine gemeinsame Schnittmenge an Datentypen beschreibt. Neben den von Microsoft für die .NET - Plattform angepassten Sprachen werden weitere .NET - Sprachen von Drittanbietern zur Verfügung gestellt, wie zum Beispiel Delphi von Borland. Dieses von Beginn an im Gesamtkonzept berücksichtigte Nebeneinander mehrerer Programmiersprachen unterscheidet .NET von anderen auf Zwischencode basierenden Laufzeitumgebungen. Die Vorteile der Unterstützung gemischtsprachiger Programmierung von .NET sind allerdings umstritten, weil sich die Wartbarkeit einer Anwendung durch den Einsatz verschiedener Sprachen prinzipiell verschlechtert.

Die oben erwähnte Framework Class Library (FCL) umfasst einige tausend Klassen, die in

Namespaces unterteilt sind. Die Klassen erfüllen Aufgaben wie zum Beispiel das Formatieren von Text, Kommunikation und Interaktion sowohl lokal als auch über Netzwerk und Internet, das Generieren von Code oder die in diesem Zusammenhang sehr nützlichen Kryptographiefunktionen. Die Dokumentation der Klassen wird in einem so genannten Software Development Kit (SDK) mitgeliefert.

Plattformunabhängigkeit

.NET ist prinzipiell plattformunabhängig. Von Microsoft wird die CLI allerdings nur für Windows und in eingeschränkter Form für FreeBSD und Mac OS X angeboten. Weiters gibt es für mobile Geräte mit Windows CE sowie für die XBox 360 Spielkonsole das „.NET compact framework“, welches große Teile der Funktionalität des kompletten Framework abdeckt, aber um Klassen welche für mobile Endgeräte unnötig sind oder zu viel Speicherplatz benötigen gekürzt ist. Eine Laufzeitumgebung für das Betriebssystem Linux und verschiedene Derivate steht in Form vom Mono (Novell 2007) zur Verfügung, das ursprünglich von der Firma Ximian als open source Projekt initiiert wurde. Das Mono-Projekt gilt als freie .NET-Alternative. Seit Herbst 2004 stehen endgültige Versionen des Produktes zur Verfügung. Bis auf eine handvoll Windows-spezifische Funktionen sind alle Komponenten von .NET innerhalb dieses Projektes vorhanden, sogar Windows Forms und ASP werden emuliert. Auch Portable.NET (DotGNU Project 2007), ein Teil des DotGNU Projekts, ist eine CLI Implementierung, welche eine Vielzahl von Prozessoren unterstützt und bereits beachtliche Anteile der Base Class Library implementiert.

Sicherheit

Eines der wichtigsten Konzepte ist das Sicherheitskonzept von .NET, das deutlich über das von Windows oder Java hinausgeht. Es existieren Mechanismen, die die Identität des Programmherstellers gewährleisten (Authentizität), Mechanismen die Programme vor Veränderung (zum Beispiel durch Programmaviren) schützen sowie Schutzmechanismen, die die Herkunft von Software und die Zone der Programmausführung (zum Beispiel Internet) einbeziehen. Technisch gesehen wird ein Codebasiertes (Code based Security) und ein Nutzerbasiertes (Role based Security) Sicherheitsmodell unterschieden. Negativ fällt allerdings bei der Web- und Datenbankprogrammierung unter .NET eine große Anzahl an alternativen und redundanten Sicherheitsmodellen von Betriebssystem, CLR, Datenbank und Webserver auf. Dies hätte nach Meinung des Autors insbesondere bei

Produkten aus ein und derselben Firma einheitlicher gestaltet werden können. Für .NET 3.5 sind in diesem Bereich jedoch Neuerungen geplant, welche diesem Missstand Abhilfe schaffen könnten. Ab .NET 3.0 und mit Windows Vista steht auch ein neuer Kryptographiedienst zur Verfügung, welcher allerdings leider keine neuen kryptographischen Hash Algorithmen beinhaltet.

Performance

Managed code wird wie oben erwähnt von der Laufzeitumgebung Common Language Runtime (CLR) verwaltet. Diese virtuelle Maschine übernimmt die Anforderung und Freigabe von Speicher und anderen Ressourcen (automatische Speicherbereinigung) und stellt sicher, dass geschützte Speicherbereiche nicht direkt adressiert oder überschrieben werden können. Zu beachten ist, dass die automatische Speicherbereinigung (durchgeführt vom Garbage Collector) Objekte nicht unbedingt sofort freigibt, wenn sie nicht mehr verwendet werden, sondern dies beispielsweise bei hoher Prozessorlast auch später durchführen kann. (Nur in sehr wenigen Fällen ist es sinnvoll und nötig, eine solche „Garbage Collection“ manuell anzustoßen.) Wie oben unter dem Abschnitt Sicherheit erwähnt, können auch Zugriffe auf Dienste, Dateisystem-Funktionen oder Geräte überwacht und von der CLR abgelehnt werden, wenn sie gegen Sicherheitsrichtlinien verstößen. Die automatische Ressourcenverwaltung und die verbesserte Sicherheit haben aber auch einen Preis – die Ausführung von managed code hat einen erhöhten Bedarf an Ressourcen und benötigt geringfügig mehr Zeit. Außerdem sind die Antwortzeiten wesentlich schwieriger zu kalkulieren und zum Teil deutlich größer, was die Anwendbarkeit für Echtzeitaufgaben unter Umständen stark einschränkt.

Für den Erfolg von .NET war es wichtig, die Entwicklergemeinde von C++ für .NET zu gewinnen. Daher war Geschwindigkeit bei .NET von Anfang an ein wesentliches Designziel. Durch verschiedene Techniken wurde versucht, den negativen Einfluss der CLR auf die Ausführungsgeschwindigkeit möglichst klein zu halten. Zum Beispiel wurde analog zu Java ein so genannter JIT-Compiler eingeführt, der vereinfacht ausgedrückt einen Mittelweg zwischen Interpretation und Kompilierung geht. Weiters kann man .NET Programme auch herkömmlich kompilieren, wodurch die erstmaligen Ladezeiten bei Programmen mit größeren Klassenmengen reduziert werden.

Dennoch wird .NET inzwischen auch bei performancekritischen Anwendungen, wie Beispielsweise 2D und 3D Simulationen und Computerspielen, Animationsprogrammen, Konstruktionsprogrammen und ähnlich, hochaufwendigen Anwendungen eingesetzt, was nicht zuletzt auch den stetig zunehmenden Prozessorgeschwindigkeiten und Arbeitsspeichervolumina zu verdanken ist. Diese Entwicklung ist nicht überraschend, da sich für viele Firmen und Projekte der Unterschied im Entwicklungsaufwand viel deutlicher auswirkt als der begleitende Performanceunterschied, sofern überhaupt vorhanden.

2.3.2 Die Programmiersprache C#

C# ist eine high level Programmiersprache der 3. Generation, die wie Java auf einer virtuellen Maschine ausgeführt wird. C# verwendet neben Konzepten aus Java auch Konzepte aus C++, Visual Basic und Delphi und beruht auf dem Standard (ECMA 334).

Microsoft reichte im August 2000 zusammen mit Hewlett-Packard und der Intel Corporation C# bei der Europäischen Standardisierungsorganisation European Computer Manufacturers Association ECMA zur Standardisierung ein. Im Dezember 2001 veröffentlichte die ECMA den Standard ECMA-334 mit dem Titel „C# Language Specification“. 2003 wurde C# auch von der ISO standardisiert (ISO/IEC 23270).

Im Juni 2005 genehmigte die ECMA die dritte Version (C# 2.0) der C# - Spezifikationen und aktualisierte den bisherigen Standard ECMA-334. Hinzu kamen die partiellen Klassen, anonyme Methoden, nullable types und Generics, ein Sprachfeature welches Typenabstraktion und -unabhängigkeit ermöglicht. (siehe unten) Im Juli 2005 übergab die ECMA die Standards an die ISO/IEC JTC 1. Die zweite Version von C# mit dem .NET-SDK 2.0 und Visual Studio 2005 wurde im November 2005 von Microsoft veröffentlicht.

Die ECMA Spezifikation 334 deckt nur die Sprache C# ab. Programme, die in C# geschrieben werden, nutzen gewöhnlich das im vorigen Abschnitt beschriebene .NET - Framework, welches durch die oben angeführten Spezifikationen beschrieben wird. Im Folgenden werden nun die wichtigsten Sprachfeatures beschrieben.

Events und Delegaten

Als .NET - Sprache verfügt auch C# über Sprachunterstützung für Events und Delegaten. Ein Delegat kann auf Methoden einer Klasse verweisen. Das Konzept lässt sich mit Funktionszeigern vergleichen. Im Unterschied zu Funktionszeigern enthalten Delegaten zusätzlich auch Verweise auf die zu den Methoden gehörenden Objekte. Ein Aufruf eines Delegaten ruft also Methoden auf, denen implizit ein Objektzeiger als Parameter mit

übergeben wird. Des Weiteren müssen Delegaten typensicher deklariert werden, was zur Folge hat, dass Inkompatibilitäten der Methodensignaturen zwischen aufrufendem Delegat und der aufzurufenden Methode schon während der Kompilation aufgelöst werden. Dies verhindert Probleme mit Pointern wie sie aus dem C++ bekannt sind und ermöglicht ein übersichtliches Event System.

Custom Attributes

Attribute erlauben es, Informationen über eine Klasse, ein Objekt, oder eine Methode zu speichern, die zur Laufzeit ausgewertet werden können. Man spricht hierbei auch von Metadaten, die beispielsweise im Rahmen der Komponenten basierten Programmierung Komponenteneigenschaften ausdrücken können. Es können dem Code also beschreibende Eigenschaften für die Verteilung, Installation, Konfigurierung, Sicherheit, Serialisierung, für Transaktionen und für viele andere Anwendungsmöglichkeiten hinzugefügt werden.

Innerhalb einer Anwendung kann mit Hilfe der Reflection-API auf die Attribute einer Assembly und die in ihr enthaltenen Elemente zugegriffen werden. Ein einfaches Anwendungsbeispiel findet sich im .NET Framework, welches einen XmlSerializer zur Verfügung stellt, der über diesen Mechanismus entsprechend markierte Elemente aus einer Klasse serialisiert und deserialisiert.

Das Konzept wurde auch in Java als so genannte „Annotationen“ übernommen. Damit wurden beispielsweise in der aktuellen Version des Standards für Enterprise Java Beans (DeMichiel 2006) die Entwicklung von EJBs vereinfacht.

Reflection, Emit

Die Reflection ist ein Objektmodell des jeweils zugehörigen Code Namespace. Über die Reflection ist es mit dem so genannten Emit - Mechanismus möglich, zur Laufzeit Programmcode über ein Objektmodell zu generieren und es im Arbeitsspeicher in eine lauffähige Assembly zu überführen. Diese Technik wird unter anderem von ASP.NET genutzt sowie einigen anderen Mechanismen wie zum Beispiel der Geschwindigkeitsoptimierung regulärer Ausdrücke. Zudem ist es damit auch möglich zur Laufzeit Skriptcode dynamisch zu kompilieren und einzubinden, was das Design von Code- und Formeleditoren, Makrofunktionen, genetischen Algorithmen und ähnlichen Anwendungen vereinfacht.

Generics

Generics fügen Klassen, Interfaces und anderen Sprachelementen Typenparametrierbarkeit hinzu. Das heißt, dass eine Klasse von einem oder mehreren Typen oder Typeninterfaces abstrahiert werden kann. Zum einen entfällt dadurch unnötiges Casting, was die Übersichtlichkeit erhöht und Fehler vermeiden kann. Zum anderen ermöglicht dies auch polymorphes Verhalten in Bezug auf Typenparameter, was in performancekritischen Anwendungen von Vorteil sein kann. Ein Typenparameter wird dem Klassennamen in spitzen Klammern angehängt und kann über das Schlüsselwort „where“ zusätzlich eingeschränkt werden. So kann beispielsweise sichergestellt werden, dass ein Typenparameter über einen parameterlosen Konstruktor verfügt. Zu den Einschränkungsmöglichkeiten („constraints“) gehören aber auch abstrakte Klassen und Interfaces.

Anonyme Methoden

Eine Anonyme Methode ist eine Methode ohne Methodename. Eine solche Methode kann im Code deklariert und zugewiesen werden. Im Wesentlichen wird dies für das Instanziieren von sehr einfachen Delegaten eingesetzt. Allerdings muss der Einsatz dieser Technik im Einzelfall gründlich abgewogen werden, da die Technik auch die Lesbarkeit und Wartbarkeit des Quellcodes beeinträchtigen kann.

Iteratoren

Über Iteratoren können C# Methoden vom Typ `IEnumerable` Elemente einzeln zurückliefern. Dies wird mit dem Schlüsselwort „`yield return ...`“ bewerkstelligt. Hauptvorteil der Technik ist die verbesserte Interoperabilität zwischen den verschiedenen enumerierbaren Typen und selbst definierten `IEnumerables` durch „`foreach`“ - Schleifen.

Partiale Klassen

Partiale Klassen sind Klassen, welche auf mehrere physikalische Codedateien aufgeteilt sind. Dies wird durch das „`partial`“ Schlüsselwort gekennzeichnet. Partiale Klassen werden beispielsweise vom Windows Forms Designer genutzt, um Designer - generierten Code sauber von händisch erstelltem zu trennen. Dadurch kann verhindert werden, dass der Programmierer Codeabschnitte verliert, weil eine automatisch erstellte Code-Datei neu

generiert wird. Dies wird auch in anderen Anwendungen mit automatisch erstellten Codedateien genutzt.

Generics, Iteratoren und partielle Klassen sind eigentlich Bestandteil des .NET-Frameworks ab Version 2.0 und stehen somit auch anderen .NET - Programmiersprachen, wie Beispielsweise VB.NET, zur Verfügung. Im Gegensatz dazu sind anonyme Methoden jedoch ein Bestandteil von C# und stehen ausschließlich in dieser Sprache zur Verfügung. Gleiches gilt für Covarianz beziehungsweise Contravarianz bei Delegaten.

Interop

Mit Hilfe der so genannten Interop - Technik lassen sich traditionelle COM - Programme mit .NET - Hüllen versehen und wie .NET - Klassen aufrufen. Umgekehrt lassen sich auch .NET - Klassen wie COM - Klassen aufrufen. Dies erleichtert die Umstellung von älteren Projekten auf .NET signifikant. Bislang sind nur wenige Sprachen auf dem Markt, die sowohl managed als auch unmanaged Code in einer einzigen Programmdatei unterstützen. Zu ihnen gehören C++/CLI (managed C++) sowie Delphi ab der Version 8.

Als nächstes wird nun im Detail vorgestellt, wie das Paket implementiert wurde.

3. Methoden

Die zur Arbeit zugehörige Software wurde vom Autor alleine implementiert. Daher kam eine herkömmliche Software Entwicklungs- und Projektmanagement Methode nicht zum Einsatz. Stattdessen wurde nach den wesentlichen Prinzipien des „Extreme Programming“ (XP) vorgegangen, welche in diesem Kapitel kurz erläutert werden. Weiters werden die Produkte und Arbeitsmethoden vorgestellt, die zur Implementierung der Software eingesetzt wurden. Neben der Entwicklungsumgebung wurde eine Reihe von Tools und Softwarebibliotheken eingesetzt, welche die Produktivität und Sicherheit steigern und Fehler vermeiden helfen. Das hier vorgestellte Setup beruht im Wesentlichen auf einer vereinfachten Version der „Best Practice“ und „Development Guidelines“ der Firma BlueLogic Software Solutions, und ist bereits über mehrere Jahre erfolgreich im Produktionsbetrieb eingesetzt worden.

3.1 Extreme Programming

Extreme Programming (XP) ist ein Vorgehensmodell zur Entwicklung von Software und gehört zu einer Reihe von Vorgehensmodellen, welche gerne unter dem Begriff „agiles Software- Development“ (kurz: agiles SWD) zusammengefasst werden. XP wurde in den Jahren 1995 bis 2000 von Kent Beck, Ward Cunningham und Ron Jeffries während der Arbeit an einer Lohnverrechnungssoftware bei Chrysler (Chrysler 1998) entwickelt. In (Kent Beck 2004) wird die Technik ausführlich beschrieben und erklärt.

Extreme Programming kann als Reaktion auf Schwierigkeiten mit traditionelleren Software Entwicklungsmethoden betrachtet werden: Unklare Problemstellung, spätes Verständnis der Anforderungen und ständige Änderungen lassen regelmäßig große Software Projekte scheitern. Hohe Flexibilität und adaptives, paralleles Arbeiten sind daher zentrale Konzepte dieser Methode, die bei oberflächlicher Betrachtung manchmal als rein philosophisches Theoriewerk missverstanden wird. Aber gerade in dieser strategisch unterschiedlichen Betrachtungsweise des Softwareentwicklungsprozesses liegt die große Stärke der Technik. Zum Beispiel wird „Refactoring“ als ganz normale Entwicklungstechnik betrachtet, nicht als eine Technik, die nur notfalls eingesetzt wird um veralteten, nicht mehr hinreichend funktionierenden Code zu „retten“. Darüber hinaus werden sogar Änderungen der Anforderungen an die Software als normaler SWD Prozess betrachtet. Dass dies nicht nur graue Theorie ist beweisen mittlerweile unzählige, erfolgreiche XP - Softwareprojekte in allen Größen und Variationen. Die Firma

Thoughtworks (Thoughtworks 2007) berichtet zum Beispiel über erfolgreiche verteilte, kommerzielle XP - Softwareprojekte in Teams mit sechzig und mehr Entwicklern.

Zu den zentralen Hauptbestandteilen der XP Technik zählen Werte, Prinzipien und Praktiken, welche im Folgenden kurz vorgestellt werden.

Werte

Als Werte werden Kommunikation, Einfachheit, Feedback, Mut und Respekt genannt. Diese beziehen sich – abgesehen von der Einfachheit – in erster Linie auf den Umgang mit Teammitgliedern und Kunden und werden aufgrund der geringen Relevanz für diese Arbeit hier nicht weiter erläutert.

Prinzipien

Die 14 Prinzipien des XP lauten:

- Menschlichkeit
- Verbesserungen
- Kleine Schritte
- Reflexion
- Wirtschaftlichkeit
- Beidseitiger Vorteil
- Selbstgleichheit
- Vielfältigkeit
- Lauffähigkeit
- Gelegenheiten wahrnehmen
- Redundanzen vermeiden
- Fehlschläge hinnehmen
- Qualität
- Akzeptierte Verantwortung

Diese Prinzipien sind etwas konkreter als die oben genannten Werte zu verstehen und in (Beck 2004) ausführlicher erklärt.

Praktiken

Über die traditionellen XP Praktiken gibt folgende Tabelle eine zusammenfassende Übersicht:

XP - Praktik	Beschreibung
Pair-Programming	Beim Pair - Programming arbeiten zwei Programmierer zugleich an einem Computer. Während einer schreibt, achtet der andere auf Fehler. Die Rollen werden regelmäßig getauscht. So werden nicht nur Fehler vermieden sondern es wird auch das Wissen schneller verteilt und Projekte sind nicht mehr so anfällig gegen den Ausfall eines einzelnen.
Kollektives Eigentum	Aktivitäten werden zunächst nicht an einzelne Personen verteilt, sondern an das ganze Team. Wissensmonopole werden vermieden und gemeinsames Verantwortungsgefühl gefördert.
Permanente Integration	Die einzelnen Komponenten eines Projektes werden in regelmäßigen Abständen zu einem lauffähigen Gesamtsystem zusammengesetzt. Auch dieser Schritt fördert das frühe Aufdecken von Fehlern und senkt außerdem automatisch die Kosten der Integration, die meist vollautomatisiert abläuft und schon zu einem frühen Zeitpunkt im Projektverlauf stabil ist.
Testgetriebene Entwicklung	Bei der Test - getriebenen Entwicklung werden erst die Modultests (Unit - Test) geschrieben, bevor die eigentliche Funktionalität programmiert wird. Dies vermeidet spätere Fehler vor allem wenn diese indirekt entstehen. Neben Unit Tests werden Integrationstests, Regressionstests, Performancetests und Akzeptanztests durchgeführt.
Kundeneinbeziehung	Enge Einbeziehung des Kunden, User Stories (Vereinfachte Use Cases), Akzeptanztests, Story-Cards.
Refractoring	Laufendes Refactoring, ständige Architektur, Design- und Code-Verbesserungen. In der XP Philosophie wird akzeptiert, dass Code nicht von Anfang an perfekt ist.
Keine Überstunden	Überstunden werden vermieden, weil darunter die Freude an der Arbeit, die Konzentrationsfähigkeit der Programmierer und somit auch die Qualität des Produktes leidet.
Iterationen	Kurze Iterationen in den Arbeitsprozessen, vor allem um Kunden in regelmäßigen Abständen einen lauffähigen Zwischenstand liefern zu können. Eine Iteration ist eine zeitlich und fachlich in sich abgeschlossene Einheit.
Metapher	Damit ist gemeint, dass in kommerziellen Produkten in der Fachsprache des Kunden kommuniziert wird. Die Programmierer müssen sich dazu auf bestimmte Metaphern einstellen.
Coding Standards	Das Team hält sich bei der Programmierarbeit an explizit festgelegte Coding Standards, was das nahtlose Weiterarbeiten an bestehendem Code für alle Teammitglieder erleichtert.

Einfaches Design	Auch in der Softwaretechnik ist Ockhams Rasiermesser bekannt. Das bedeutet, dass im Zweifelsfall die einfachere von zwei Lösungen die „richtigere“ ist.
Planning Game	Software Releases werden in einem so genannten Planning Game spezifiziert und in ihrem Aufwand geschätzt. Während dieses iterativen Vorgehens sind sowohl Entwicklungsmannschaft als auch Kunde anwesend.

Tabelle 4 – Traditionelle XP Praktiken. Extreme Programming (XP) ist ein so genanntes agiles Vorgehensmodell zur Entwicklung von Software und wird in (Kent Beck 2004) genau beschrieben. Für die Entwicklung der zur Arbeit gehörigen Software wurden Techniken und Prinzipien aus dieser Methode – soweit für Ein-Personen-Teams anwendbar – herangezogen.

Mittlerweile wurden diese traditionellen Praktiken durch die so genannten „evolutionären Praktiken“ ersetzt, welche zum Teil eine Umordnung oder Neuinterpretation der traditionellen Praktiken sind, zum Teil aber auch neue, verfeinerte Konzepte darstellen:

- Räumlich zusammen sitzen
- Informativer Arbeitsplatz
- Teamwork
- Pair-Programming
- Energievolle Arbeit
- Entspannte Arbeit
- User Stories
- Wöchentlicher Zyklus
- Quartalsweiser Zyklus
- 10-Minuten-Build
- Kontinuierliche Integration
- Test-First-Programmierung
- Inkrementelles Design

Zu diesen kommen dann noch Begleitpraktiken, auf die hier nicht mehr weiter eingegangen wird.

Anwendung

Im vorliegenden Projekt war daher das erste Ziel der Softwareentwicklung einen funktionierenden Prototypen für beide grundlegenden Algorithmen, nämlich der binäre Delta Algorithmus nach Tridgell und der Textdelta Algorithmus nach Myers und deren Anwendungen zu programmieren. Dieses Ziel wurde vor allem auch dank des umfangreichen .NET Framework binnen nur drei Tagen erreicht. Dazu muss allerdings erwähnt werden, dass dieser erste Prototyp wie zu erwarten enorm ineffizient und unzuverlässig arbeitete. Während der Text Delta Algorithmus Out - of - bounds Exceptions und in einigen Fällen sogar Stack Overflows durch Endlos - Rekursionen warf und in vielen Fällen schlichtweg falsche Edit Scripts erzeugte, konnte auch der Binäre Delta Algorithmus nicht durch Zuverlässigkeit oder Performance überzeugen: zu Beginn wurde nur der erste Adler32 Treffer ausgewertet und mit einem MD4 abgesichert. Wenn dieser keine echte Übereinstimmung anzeigte, wurde ein insert Befehl herausgegeben. Außerdem führte ein Fehler bei der Adaption des „SlidingStreamWindow“ (Originalversion von Bernhard Glück) dazu, dass ein fester Anteil der Rekonstruktionsoperationen – durch einen SHA-256 überprüft – versagte. Solche Fehler sind bei dieser Vorgehensweise aber zu erwarten und der psychologische Effekt einer (prinzipiell) funktionierende Software vorliegen zu haben überwiegt für den Programmierer – wie auch für den Kunden in kommerziellen Projekten – deutlich.

Als hauptsächliche Performance-Bottlenecks wurden dann zum einen die noch nicht optimierte Adler32 Berechnung und zum anderen die Xml - Serialisierung identifiziert. Diese Xml - Serialisierungstechnik ist zwar enorm praktisch – Xml Elemente müssen nur mit entsprechenden Attributen wie [XmlAttribute] oder [XmlElement(„Name“)] markiert werden – , ist aber dafür auch wenig performant, da die zu serialisierenden Daten vom System.Serialization.XmlSerializer über die Reflection herausgesucht werden müssen. Weiters wurden grundlegende Architekturschwächen offensichtlich, was der Hauptvorteil der XP Vorgehensweise ist: wäre zuerst versucht worden, alle Klassen möglichst optimal auszuprogrammieren, so wären Schwächen in der Architektur möglicherweise erst sehr spät erkennbar geworden und eine wesentlich größere Menge an Code hätte verworfen werden müssen.

Im Anschluss wurde in mehreren Refactoring Schritten („Iterationen“) eine immer zuverlässigere und effizientere Version der Software erzeugt. Bereits nach wenigen Manntagen Programmieraufwand konnte dann eine voll funktionsfähige Version am 18. April in der UMIT in Hall vorgestellt werden. Einzig die graphischen Tools waren zu

diesem Zeitpunkt noch unvollständig. Dies wurde nach einer weiteren Iteration korrigiert und am 1. Juni wurde ein Release Kandidat zusammengestellt. Im Anschluss erfolgten verschiedene Tests, welche in Kapitel 4 näher beschrieben werden. Nach einer letzten Iteration und abschließenden Tests wurde dann am 11. Juni 2007 die Version 1.0 eingefroren und in Folge dem Institut für Informationssysteme des Gesundheitswesens zur Überprüfung überlassen.

3.2 Entwicklungsumgebung

Als Entwicklungsumgebung wurde das Microsoft Visual Studio .NET 2005 Team Edition mitsamt Performance Tool, Unit Testing und Code Analyse (FxCop) benutzt. Das Produkt sowie seine Vorgänger (VS.NET 2002, 2003 und VS6) wurden vom Autor bereits in mehreren kommerziellen Projekten erfolgreich eingesetzt. Der große Funktionsumfang sowie eine Reihe von Features (wie zum Beispiel „IntelliSense“), welche die Produktion signifikant beschleunigen waren weitere Gründe für die Wahl dieses Produktes. Integriert in Visual Studio ist ein Designer für Windows.Forms, welcher das Modellieren, Anzeigen und die Steuerung von win32 Oberflächen ermöglicht, ohne dass dazu die win32 API (unmanaged) benutzt werden muss. Der Forms Designer wurde benutzt, um zwei kleine graphische Anwendungen zu erstellen, welche die grundlegenden Funktionen der Software demonstrieren können und teilweise zum Testen der Anwendung benutzt wurden. (siehe Abschnitt 4 – Ergebnisse)

Ergänzend wurde der „ReSharper“ von JetBrains eingesetzt, welcher die Produktivität mit einer Reihe von zusätzlichen Code-Assistance Features steigert. Dazu gehören erweitertes Fehler- und Syntax Highlighting, Quick-Fixes, Refactoring Funktionen und einige weitere Features, welche Visual Studio nicht in diesem Ausmaß zur Verfügung stellt.

Obwohl die Implementierung vom Autor alleine vorgenommen wurde, wurde das Projekt zuerst in ein VCS-System (Perforce) innerhalb des Unternehmens des Autors eingecheckt, um das im XP übliche gleichzeitige Diskutieren und Refactoring des Codes durch zwei oder mehr Entwickler zu ermöglichen. Der Geschäftspartner des Autors (Bernhard Glück) konnte so, ohne persönlich anwesend zu sein wertvolle Anregungen liefern und dem Autor bei der Auffindung von Schwachstellen und dem Identifizieren von möglicherweise aufkommenden Problemen behilflich sein.

Die wichtigsten zum Einsatz gelangten Namespaces aus dem .NET Framework Base Class Library beziehungsweise der Visual Studio Entwicklungsumgebung werden in folgender Tabelle aufgelistet:

Namespace	Verwendung
Microsoft.VisualStudio.QualityTools.UnitTestFramework	Unit Testing. Diese Referenz wird vom SDelta Software Library nicht benötigt.
System.Collections	Verschiedene (auch generische) Listen und Dictionaries werden stark eingesetzt, außerdem wird die System.Collections.Hashtable generisch erweitert.
System.Diagnostics	Eine spezielle Timer Funktion (Stopwatch) wird benutzt um Performancedaten zuverlässig zu bestimmen. Ein normaler Timer ist dazu nicht geeignet.
System.Drawing	Zeichnen von Edit Scripts über GDI+
System.IO	Datei Ein- und Ausgabefunktionen.
System.Reflection	Durchsuchen der Assembly nach Digest Funktionen, welche das entsprechende Interface implementieren.
System.Text	Reguläre Ausdrücke und andere String und Text Operationen.
System.Windows.Forms	Graphische Oberfläche. Die PictureBox und die RichTextBox werden überschrieben und erweitert.
System.Xml	Wurde vorübergehend zur Serialisierung von Hash- und Delta Daten eingesetzt und in weiterer Folge aus Performancegründen durch eine eigene XML Serialisierung ersetzt.

Tabelle 5 - .NET Namespaces und deren Verwendung im vorliegenden Softwareprojekt. System.Xml wurde nur vorübergehend eingesetzt, System.Windows.Forms, System.Diagnostics und System.Reflection werden nur in den Wizards benötigt und der Einsatz von Microsoft.VisualStudio.QualityTools.UnitTestFramework beschränkt sich rein auf das Testprojekt.

3.3 Dokumentation

Um die Quellcode Dokumentation zu vereinfachen und zu beschleunigen wurde Roland Weigelt's GhostDoc eingesetzt. Die Software unterstützt beim Erstellen von XML

Dokumentationskommentaren. Sandcastle (Version 2007 CTP) von Microsoft (Microsoft 2007) generiert daraus ähnlich dem bekannten JavaDoc Hilfedateien wahlweise im CHM oder HTML Format. Die .NET Framework Dokumentation wurde laut Angaben von Microsoft mit dieser Software erstellt. Diese Quellcodedokumentation liegt dem erstellten Software Library als CHM Datei sowie in Form einer Webseite (HTML) bei.

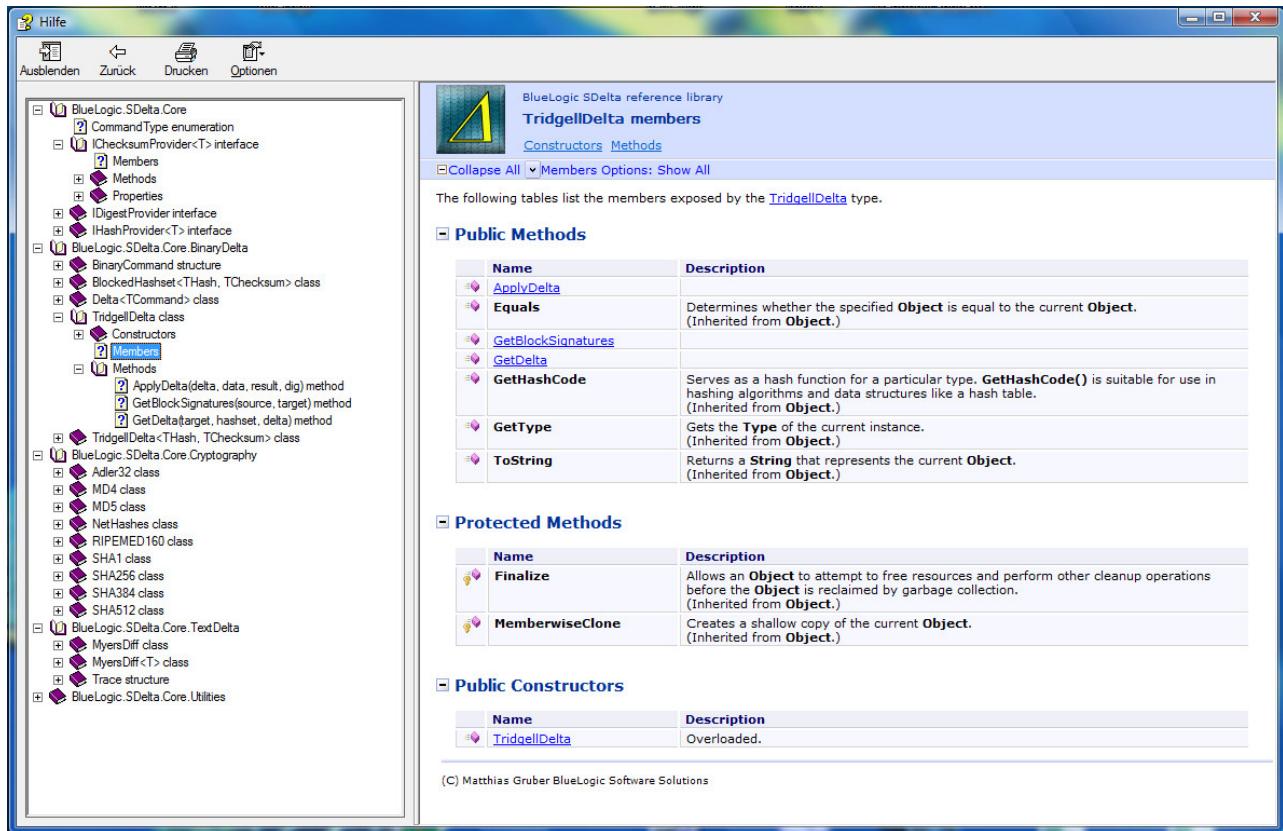


Abbildung 3 – Ein Auszug aus der Code Dokumentation, welche mit Hilfe der Sandcastle Software (Microsoft 2007) generiert wurde. Diese Dokumentation liegt der erstellten Software in HTML- und CHM-Form bei.

Weiters wurden UML Diagramme mit dem „Enterprise Architect“ von Sparx Systems erstellt. Die UML Dokumentation ist in Kapitel 4 dieser Arbeit enthalten. Neben der Architektur- und Quelltextdokumentation findet sich dort auch ein kurzes Manual zur erstellten Softwarebibliothek und den beigefügten GUI Anwendungen.

3.4 Testing

Zu allen Klassen, Funktionen und Properties der zur Arbeit gehörigen Software Library wurden soweit möglich und sinnvoll Unit Tests erstellt. Ausgenommen sind die beiden

graphischen Oberflächen, welche mit dem Windows Forms Designer erstellt wurden. Die Binär Delta Funktionen wurden in mehreren Testläufen an großen Sets von Dateien unterschiedlichen Typs validiert und getestet. Die Text Delta Funktionen wurden durch graphische Auswertung der Edit Scripts in einer zufälligen Auswahl an Webseiten und anhand einer Reihe von Sonder- und Grenzfällen validiert. Die zum Testen eingesetzten Funktionen sind Bestandteil der graphischen Oberflächen, welche vom Autor zusammen mit der Softwarelibrary zur Verfügung gestellt werden. Die folgenden Abbildungen zeigen die Code Abdeckung („Code Coverage“) durch Unit Tests.

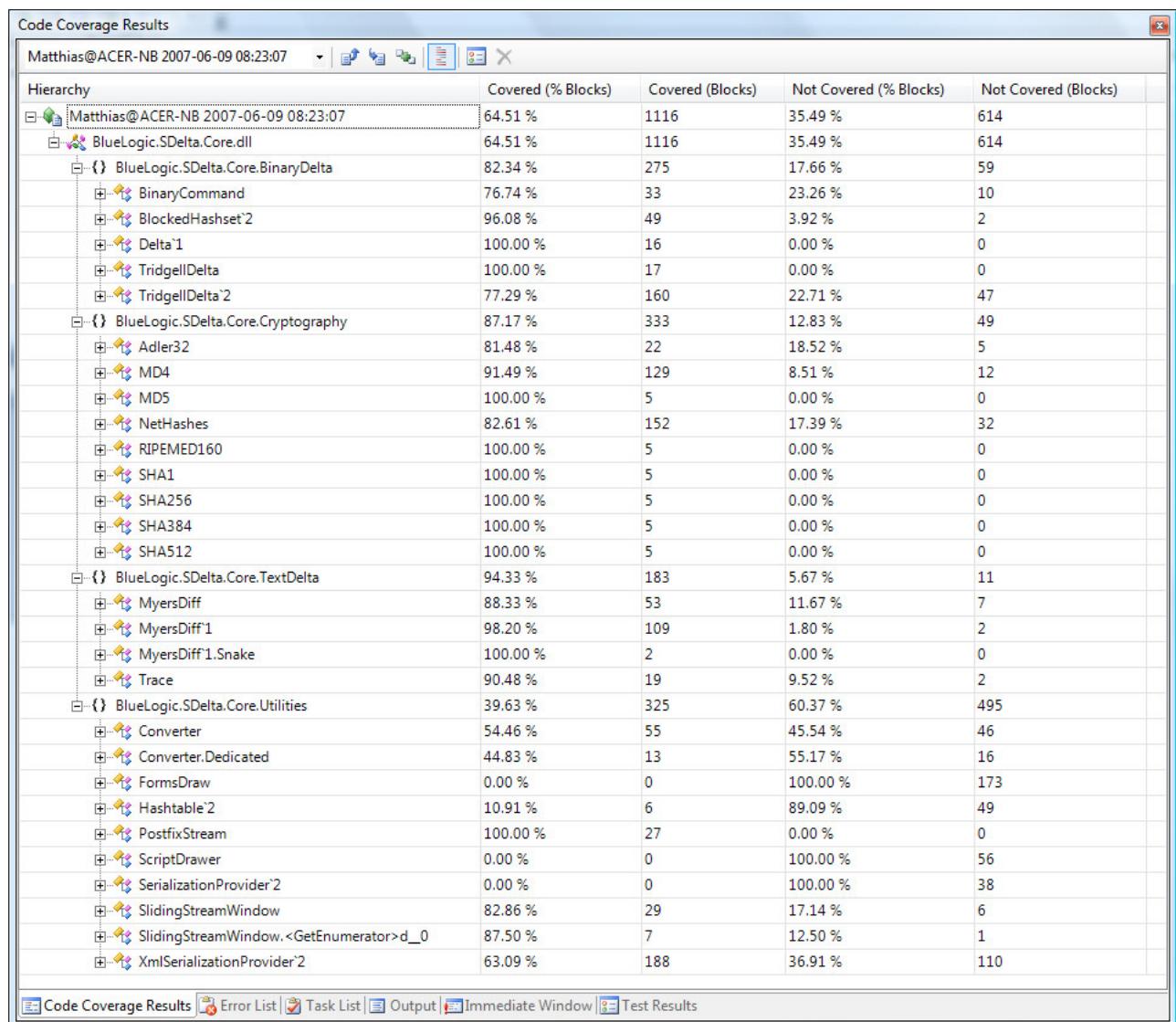


Abbildung 4 – Code Coverage Resultate. Die Abbildung gibt eine Übersicht über die absolute und relative Quellcode Abdeckung durch Unit Tests. Für die Klassen „FormsDraw“, „ScriptDrawer“ und „SerializationProvider“ wurden keine expliziten Unit Tests angelegt, diese wurden einem manuellen Test unterzogen.

Die Klasse „FormsDraw“ kommt aus einem länger zurückliegenden Projekt des Autors. Nur kleine Teile davon sind zum Einsatz gelangt. Die Klasse ScriptDrawer wird nur zur graphischen Ausgabe von Edit Scripts benutzt und der SerializationProvider ist als Prototyp zu verstehen. Daher wurden diese Funktionen händisch getestet und es wurden keine Unit Tests erstellt. Detaillierte Code Coverage Resultate für die Klassen TridgellDelta und MyersDiff befinden sich in Anhang 3 (Abschnitt 10.3).

In weiterer Folge werden nun besonders wichtige Unit Tests vorgestellt. Im Anschluss folgt eine genaue Dokumentierung einiger wichtiger manueller Tests und der zugehörigen Testmaterialien.

3.4.1 Unit Testing

Die Unit Tests für die zentralen Algorithmen der Software verdienen besondere Aufmerksamkeit. Beide Arten von Delta Algorithmen (Text- wie auch Binär-) sind relativ komplex und empfindlich. Ein unüberlegter Optimierungsversuch kann leicht zu einem heimtückischen Fehler führen, welcher nur sehr schwer erkennbar wäre.

Binär Delta Unit Tests

Der Binär Delta Test in seiner endgültigen Form ist eine Reaktion auf mühsame Suche nach Fehlerreproduktionen („Repros“) die im Laufe der Entwicklung und dem händischen Testen auftraten. Bei dem Test werden MemoryStreams als Datenquellen und -senken eingesetzt, um ohne zusätzliche Ressourcen auskommen zu können. Dazu werden zwei Strings in UTF8 Bytecodes kodiert und dem Algorithmus zugeführt. Es wird nur ein Beispiel vorgestellt, ein weiteres Beispiel mit unterschiedlichen Daten befindet sich im Anhang. Permutationen mit verschiedenen Datenmengen und Kombinationen sind mit dem beiliegenden Wizard (Kapitel 4.2) einfacher durchzuführen.

Als Blockgröße für den Test wurde 5 Bytes gewählt und die Länge des einen Strings so gesetzt, dass zum Ende ein Block mit einem einzelnen Byte zurückgeliefert werden muss. Dies hilft Indizierungsfehler schnell aufzudecken. Der zweite String wurde außerdem so gewählt, dass im zugehörigen Delta jede Art von Kommando vorkommen muss.

Quelldatenstring in Fünfergruppen:

```
01234 56789 abcde fghij kAAAAA AAAAAA BBBB BBCCCC BBBCC CCCCCC  
CCCCC 01234 56789 abcde fghij
```

Zieldatenstring in Fünfergruppen:

```
01234 56789 abcde fghij kAAAAA AAAAA BBBBB BBBBB BBBCC CCCCC CCCCC  
CCCCC 01234 56789 abcde fghij k
```

Der zugehörige Delta sieht in XML Form folgendermaßen aus:

```
<?xml version="1.0" encoding="utf-8"?>  
<Delta digestname="SHA256" hashname="MD4" checksumname="Adler32" blocksize="5">  
  <commands>  
    <copy>  
      <from>0</from>  
      <to>5</to>  
      <repeat>0</repeat>  
    </copy>  
    <copy>  
      <from>5</from>  
      <to>5</to>  
      <repeat>0</repeat>  
    </copy>  
    <copy>  
      <from>7</from>  
      <to>7</to>  
      <repeat>2</repeat>  
    </copy>  
    <copy>  
      <from>10</from>  
      <to>11</to>  
      <repeat>0</repeat>  
    </copy>  
    <copy>  
      <from>11</from>  
      <to>11</to>  
      <repeat>1</repeat>  
    </copy>  
    <copy>  
      <from>0</from>  
      <to>3</to>  
      <repeat>0</repeat>  
    </copy>  
    <insert>  
      <base64Binary>aw==</base64Binary>  
    </insert>  
  </commands>  
  <digest>8m0vZrPzuiY6MNPS9krB3Uou+QD/YT+Qvmm97+JK5EVCbHVMb2dpYW==</digest>  
</Delta>
```

Zentral für diesen Algorithmus ist die Adler32 Checksumme, für welche acht Tests angelegt wurden, die hier ebenfalls beschrieben werden sollen. Zum einen wird in „GetNameTest“ sichergestellt, dass der korrekte Name zurückgeliefert wird. In „GetChecksumTest“ und „GetChecksumTest1“ wird einmal über einen MemoryStream und einmal über ein Byte Array eine einfache Checksumme überprüft. In „GetChecksumTest2“ bis „GetChecksumTest6“ wird unter anderem mit Hilfe zweier Byte Arrays sichergestellt, dass die Funktion auch über mehrere Berechnungen richtig weiter permutiert. Die Daten aus „GetChecksumTest2“ werden mit einer Blockgröße von 4 Byte interpretiert:

```
ByteArray 1 = { 11, 22, 33, 44 };  
ByteArray 2 = { 5, 7, 9, 13 };
```

Die Adler Checksummen betragen 14680175 und 4849697 als kombinierte unsigned long Summen oder (111, 224) und (33, 74) in Summenpaaren s1 und s2. Über die folgende Befehlsfolge muss also aus der Adler Checksumme der ersten Bytegruppe die Adler Checksumme der zweiten Gruppe entstehen, was der Unit Test sicherstellt.

```
ulong a0 = 14680175;  
ulong a1 = target.GetChecksum(a0, 11, 5);  
ulong a2 = target.GetChecksum(a1, 22, 7);  
ulong a3 = target.GetChecksum(a2, 33, 9);  
ulong a4 = target.GetChecksum(a3, 44, 11);
```

Dabei ist der erste Parameter der Adler32 Wert eines Blocks A, der zweite Parameter der Wert des ersten Bytes in A und der zweite Parameter ist der Wert des ersten Bytes welches an A anschließt. Als Ergebnis erhält man dann die Adler32 Checksumme des A nachfolgenden, um eine Byteposition versetzten Blocks.

Im Anschluss folgt nun eine Beschreibung der Unit Tests für den Text Delta Algorithmus nach Myers. Die manuellen Binär Delta Tests werden im anschließenden Abschnitt beschrieben.

Text Delta Unit Tests

Neben den trivialen Extremfällen – zwei komplett unterschiedliche beziehungsweise zwei komplett identische Texte oder Dateien – existieren eine Reihe von Sonder- und Grenzfällen, die hier Berücksichtigung finden. Um eine einigermaßen durchschaubare Abdeckung der komplexen Funktionalität des Myers Algorithmus zu erhalten wurden die Unit Tests eng an den eigentlichen Code der verschiedenen, teils rekursiven Funktionen gehalten.

In „GetDiffTest1“ befinden sich Kombinationen von Strings, die über 1-Pfade, also Edit Scripts mit nur einem Befehl ineinander überführt werden können. Folgende Paare werden überprüft:

(ABC, ABCD) (ABCD, ABC) (XABC, ABC) (ABC, XABC) (ABC, ABDC) (ABDC, ABC)

In „GetDiffTest2“ werden entsprechend Stringpaare getestet, welche über 2-Pfade überführbar sind. Folgende Paare werden überprüft:

(ABC, ABCDD) (ABCDD, ABC) (XXABC, ABC) (ABC, XXABC) (XAXBC, ABC)
(ABC, AXBXC) (ABXXC, ABC) (ABC, XABCX)

In „GetDiffTest3“ werden komplexere Fälle überprüft. Bis auf den Fall von zwei identischen Strings werden auch hier alle Tests mit vertauschten Paaren zusätzlich überprüft.

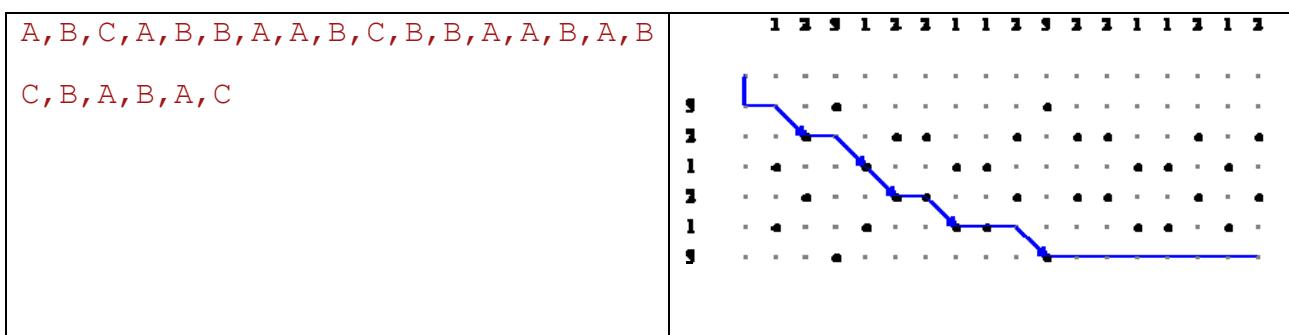


Abbildung 5 – Text Delta Unit Tests: Ein Test aus der Gruppe „GetDiffTest3“. In der linken Spalte finden sich jeweils die beiden Strings, die verglichen werden. Auf der rechten Seite sind die Spalten nach den Indices des ersten und die Zeilen nach den Indices des zweiten Strings beschriftet. Alle Zeichenketten oder Textzeilen werden in Form von Integer Werten verglichen, was den rekursiven Algorithmus performanter gestaltet.

Die restlichen Tests der Gruppe „GetDiffTest3“ werden in Anhang 1 (Abbildungen 21.1 bis 21.10) jeweils in einer Richtung dargestellt.

Schon beim Prototypen der Software wurde mit Hilfe der graphischen Ausgabe jedes berechnete Edit Script händisch kontrolliert. Um wirklich sicher zu gehen, dass der Algorithmus korrekt implementiert wurde, war es unumgänglich, eine sehr große Anzahl an graphischen Edit Scripts zu erzeugen und diese händisch genau zu überprüfen, denn auch das zweit kürzeste Edit Script würde zu einer korrekten Rekonstruktion führen, was einen Testmechanismus wie beim binären Delta unpraktikabel macht. Gesucht ist aber das kürzeste Edit Script. Zusätzlich zu diesen Unit Tests wurden eine Reihe größerer html Dateien mit zum Teil beträchtlichen Unterschieden aus dem Testdatenset B (im folgenden Abschnitt beschrieben) als Stichproben ausgewählt und mit Hilfe der GUI gründlich überprüft.

3.4.2 Zusätzliche Tests

Um den binären Delta Algorithmus ausführlicher zu testen wurde zu Beginn händisch eine Auswahl von 65 Dateipaaaren (Testdatenset A, 50 MB) auf dem Computer des Autors getroffen. Die Dateien sind von verschiedenen Formaten (Text, Office Dokumente, PDF Dateien, Bilder, Videoclips, komprimierte Dateien) und verschiedenen Größen (wenige Byte bis einige Megabyte). Auch die Unterschiede zwischen den jeweils zwei Versionen reichten von beinahe identischen Dateien bis zu völlig verschiedenen Dateien. Für jedes Dateipaar wurden in beide Richtungen Deltas berechnet. Außerdem wurde von jeder Datei ein Paar mit identischen Versionen getestet. (nicht in den folgenden Tabellen enthalten)

Erweiterung	Dateityp	Dateigrößen		Details
AVI	Microsoft Audio Video Interleave / PAL	5.57 MB	2.54 MB	Zwei kurze Videoclips die mit dem Windows Movie Maker verschieden geschnitten wurden
BMP	Microsoft Windows / OS/2 GDI Bitmap	16.3 KB 183 KB 375 KB	16.3 KB 183 KB 375 KB	Die Bilder wurden mit einer Bildbearbeitungssoftware verschieden stark bearbeitet
CHM	Microsoft Compiled HTML Help File	112 KB 145 KB 1.6 MB	120 KB 146 KB 1.43 MB	Verschiedene Versionen und Sprachen von Windows Hilfe Dateien
CS	C# Code Datei	21.8 KB	23.1 KB	Unterschiedliche Versionen von Klassen

CSS	W3C Cascading Style sheets	23.3 KB	34.7 KB	Zwei Stylesheets von verschiedenen Layouts einer Wikipedia Webseite
DLL	Dynamic Link Library	108 KB 632 KB 1.56 MB	120 KB 660 KB 1.69 MB	Verschiedene Programmversionen von GDI+, dem SDelta Framework und eines weiteren Projektes des Autors.
DOC	Microsoft Office Word Dokument	29 KB 281 KB 1.49 MB	31.5 KB 253 KB 1.49 MB	Verschiedene Versionen von Word Dokumenten
ENL	Thomson Endnote Library	8.32 KB	38 KB	Endnote Libraries von zwei verschiedenen Programmversionen
EPS	Adobe Encapsulation Postscript	34.1 KB	29 KB	Zwei verschiedene Versionen eines Firmenlogos
EVTX	Microsoft Windows Vista Ereignisprotokoll	68 KB 1.06 MB	68 KB 1.06 MB	Ereignisprotokolle mit verschiedenen Filtern und Datensätzen
EXE	Executable Datei	132 KB 196 KB 2.8 MB	140 KB 200 KB 2.8 MB	Verschiedene Versionen eines Uninstallers, des Delta Wizards und des .NET Framework 3
GIF	CompuServe Graphics Interchange Format	10.9 KB 69.8 KB 16.5 KB	13 KB 66.4 KB 17.1 KB	Die Bilder wurden mit einer Bildbearbeitungssoftware verschieden stark bearbeitet
JAR	Java Archive	3.64 KB 877 KB	2.7 KB 1.95 MB	Unterschiedliche Anwendungsversionen
JAVA	Java Code Datei	40.9 KB	40.8 KB	Verschiedene Versionen einer Java Klasse
JPG	Joint Photographic Experts Group ISO 10918-1 (JPEG)	7.16 KB 49 KB 15.9 KB	8.31 KB 52.5 KB 16.2 KB	Die Bilder wurden mit einer Bildbearbeitungssoftware verschieden stark bearbeitet
JS	JavaScript Code Datei	34.6 KB	31.6 KB	Verschiedene Versionen eines JavaScripts
MDB	Microsoft Office Access Datenbank	500 KB	548 KB	Verschiedene Versionen einer kleinen Kundendatenbank
MHT	MIME HTML (RFC 2557)	200 KB	212 KB	Zwei ähnliche Webseiten mit Bildern
MOV	Quicktime Multimedia Container Format	229 KB	235 KB	Zwei ähnliche Videos
MP3	MPEG Audio Layer 3 Musikdatei	2.92 MB	2.86 MB	Zwei verschiedene Lieder derselben Band.
ODP	OpenOffice Präsentation	14.2 KB	15.3 KB	Leicht modifizierte Präsentationen
ODS	OpenOffice Tabellenkalkulation	15.5 KB	10.8 KB	Leicht modifizierte Tabellenkalkulationen
ODT	OpenOffice Dokument	22.2 KB	27.4 KB	Leicht modifizierte Textdateien
PDF	Adobe Portable Document Format	83.8 KB 109 KB 718 KB	39.2 KB 106 KB 733 KB	Verschiedene Versionen von PDF Dokumenten
PNG	Portable Network (ISO/IEC	9.8 KB 150 KB	11 KB 150 KB	Die Bilder wurden mit einer Bildbearbeitungssoftware

	15948:2004)	29 KB	30.1 KB	verschieden stark bearbeitet
PPT	Microsoft Office Powerpoint Präsentation	100 KB	98 KB	Leicht unterschiedliche Powerpoint Präsentationen
PSPIMAGE	Jasc Paintshop Pro	18.9 KB 243 KB 240 KB 47.3 KB 49.1 KB	21.1 KB	Die Bilder wurden mit einer Bildbearbeitungssoftware verschieden stark bearbeitet
RAR	Komprimierter Ordner (Eugene Roshal)	94.4 KB	100 KB	Zwei teils unterschiedliche komprimierte Dateiordner
RB	Ruby Code Datei	47.6 KB	50.9 KB	Zwei ähnliche Ruby Quellcode Dateien
RESX	Microsoft Visual Studio Resource Datei	652 KB	710 KB	Ressourcen einer Windows Anwendung in zwei verschiedenen Versionen
TAR	Komprimierter Ordner (Tarball)	80.8 KB	77.7 KB	Version 2.0 und Version 2.1 des ZDelta Quellcodes
TIF	Adobe Tagged Image File Format (TIFF)	13.4 KB 217 KB 34.5 KB	15.3 KB 213 KB 35.7 KB	Die Bilder wurden mit einer Bildbearbeitungssoftware verschieden stark bearbeitet
VB	Visual Basic Code Datei	13.2 KB	12.8 KB	Unterschiedliche Versionen von Klassen
WAV	Microsoft / IBM Waveform Audio Format	97 KB	96.1 KB	Zwei ähnliche Klangeffekte
WMV	Microsoft Windows Media Video	79.1 KB	53.7 KB	Zwei kurze Videoclips die mit dem Windows Movie Maker verschieden geschnitten wurden
XLS	Microsoft Office Excel Sheet	29 KB	29 KB	Zwei leicht veränderte Tabellenkalkulationen
XML	W3C Extended Markup Language	123 KB	124 KB	Konfigurationsdaten für zwei verschiedene HP Drucker
ZIP	Komprimierter Ordner (Phil Katz)	95.4 KB	101 KB	Zwei teils unterschiedliche komprimierte Dateiordner

Tabelle 6 – Testdatenset für das manuelle Testing des binären Delta Algorithmus. Die Dateirekonstruktion wurde jeweils in beide Richtungen überprüft, außerdem wurden auch für identische Kopien der Dateien Deltas validiert. Art und Ausmaß der Unterschiede zwischen den jeweiligen Dateiversionen variieren und werden in der Tabelle genauer beschrieben.

Da der Prototyp zu Beginn nur ungefähr für die Hälfte der Dateipaare korrekte Deltas in beide Richtungen erzeugen konnte, wurden zuerst keine weiteren Testdaten eingesetzt. Nachdem das beschriebene Testset fehlerfrei verarbeitet wurde, mussten aber größere und statistisch aussagekräftigere Testdatensammlungen gefunden werden. Dazu wurden die Testdaten von (Trendafilov 2002) herangezogen, welche zum einen aus einer Sammlung von 10.000 HTML Dateien in vier Versionen (die Versionen 1 und 4 wurden als Testdatenset B gewählt, zusammen 271 MB) und zum anderen aus jeweils zwei Versionen der Quelltexte der Softwarepakete „emacs“ (v19.28 und v 19.29; ein Texteditor

mit über 1.400 Code Dateien; Testdatenset C, zusammen 67 MB) und „gcc“ (Gnu C Compiler v2.7.0 und v2.7.1, über 1.000 Dateien; Testdatenset D, zusammen 52 MB) bestehen.

Folgende Testläufe wurden mit der eingereichten Version der Software durchgeführt:

Datensets	Blockgröße	Kollisionen
A, B, C, D	256	0
A, B, C, D	512	0
A, B, C, D	768	0
A, B, C, D	1024	0
A, B, C, D	2048	0
A, B, C, D	3072	0
A, B, C, D	4096	0
A, B, C, D	5120	0
A, B, C, D	6144	0

Tabelle 7 – Testläufe mit dem Binär Delta Wizard. Die Ergebnisse wurden mit einem SHA256 Hash validiert. Auch bei weiteren, hier nicht angeführten Tests sind keinerlei Kollisionen des MD4 Hash Wertes aufgetreten.

Unterhalb einer Blockgröße von 2048 Byte steigt das Risiko für Kollisionen, so dass die Rekonstruktion versagen kann. (Tridgell 1999) Dieser Fall trat während der gesamten Arbeit an der Software nie auf. Dennoch wurde sicherheitshalber eine Blockgröße von 2048 Byte als Standardwert eingestellt. Über die Durchlaufzeit und den Datendurchsatz gibt der folgende Abschnitt 3.5 sowie Kapitel 4.4 (Performance) genauer Auskunft.

3.5 Profiling

Für das Profiling wurde wie erwähnt das integrierte Profiling Tool der Microsoft Visual Studio 2005 Team Edition eingesetzt. Nach mehreren Iterationen bestehend aus Profiling und Refactoring wurde ein zufrieden stellender Endzustand erreicht, der für die Anwendung mit Online Updates oder VCS Systemen völlig ausreichend ist. Für eine lokale Anwendung sollten allerdings noch weitere Optimierungen oder eine unmanaged Implementierung in Betracht gezogen werden.

Type/Allocating Function	Instances	Total Bytes Alloc...	% of Total Bytes
+ System.UInt64	9352959	149647344	68.453
+ System.String	145647	17874730	8.176
+ System.Byte[]	67765	40283935	18.427
+ System.UInt32[]	39952	2094360	0.958
+ System.Text.StringBuilder	21477	429540	0.196
+ System.Char[]	20675	2383236	1.090
+ BlueLogic.SDelta.Core.Cryptography.MD4	19895	557060	0.255
+ System.Collections.Generic.List`1	14716	353184	0.162
+ System.Int64[]	12437	826484	0.378
+ System.Collections.DictionaryEntry	12221	195536	0.089
+ System.Version	10726	257424	0.118
+ System.Object[]	5136	170460	0.078
+ System.Drawing.KnownColor	3643	43716	0.020
+ System.Collections.ArrayList	3470	83280	0.038
+ System.WeakReference	2846	45536	0.021
+ System.String[]	2007	190720	0.087
+ System.Reflection.RuntimeMethodInfo[]	1988	59128	0.027
+ System.Windows.Forms.ClientUtils.WeakRefCollection.WeakRefObject	1973	39460	0.018
+ System.Windows.Forms.LayoutEventArgs	1643	26288	0.012
+ System.Reflection.CerArrayList`1	1621	25936	0.012
+ System.RuntimeTypeHandle[]	1565	19900	0.009
+ System.Collections.Hashtable.bucket[]	1477	672312	0.308
+ System.Reflection.RuntimeMethodInfo	1453	81368	0.037
+ System.Xml.NameTable.Entry	1414	28280	0.013
+ System.Windows.Forms.Control.MultithreadSafeCallScope	1324	15888	0.007
+ System.Collections.Hashtable	1274	71344	0.033
+ System.Collections.ArrayList.ArrayListEnumeratorSimple	1032	28896	0.013
+ System.Security.PermissionSet	969	34884	0.016
+ System.Xml.Serialization.XmlSerializationPrimitiveWriter	927	63036	0.029
+ System.Xml.Serialization.XmlSerializationPrimitiveReader	927	407880	0.187
+ System.Int32	906	10872	0.005
+ System.Reflection.Runtime PropertyInfo[]	872	40992	0.019

Abbildung 6 – Profiling Ergebnisse. Die Liste zeigt die am häufigsten erzeugten Instanzen beim Verrichten einer ausgewählten Testaufgabe. Weitere Ergebnisse finden sich im Anhang.

4. Ergebnisse

Als Ergebnis der vorliegenden Arbeit kann das erfolgreiche Implementieren eines managed Text- und Binärdelta Library betrachtet werden. Dieses Kapitel beschreibt daher die Softwarelibrary und die zugehörigen graphischen Benutzerschnittstellen (GUIs) sowie Features und Performance der Anwendung. Außerdem wird beschrieben, wie und für welche Anwendungen die Software eingesetzt werden kann. Im Bereich der beiden graphischen Oberflächen, welche nur beispielhaft die Anwendung der Software demonstrieren sollen, wird nicht besonders auf technische Details (Klassendiagramme etc.) eingegangen.

4.1 Core Library

Kern der Software ist eine DLL Datei, welche den Namespace BlueLogic.SDelta.Core enthält. Dieser ist in weitere Namespaces aufgeteilt und beinhaltet alles, was zur Konfiguration, Berechnung und graphischen Darstellung von Text- und Binärdeltas benötigt wird.

Namespaces und Klassen

BlueLogic.SDelta.Core

- .BinaryDelta
 - TridgellDelta
 - TridgellDelta<THash, TChecksum>
 - BlockedHashSet
 - Delta
 - BinaryCommand
- .TextDelta
 - MyersDiff
 - MyersDiff<T>
 - Trace
- .Cryptography
 - Adler32
 - MD4

- MD5
- SHA1
- SHA256
- SHA384
- SHA512
- RIPEMD160
- NETHashes
- .Utilities
 - Converter
 - FormsDraw
 - Hashtable
 - PostfixStream
 - ScriptDrawer
 - SerializationProvider
 - SlidingStreamWindow
 - XmlSerializationProvider
- CommandType (Enumeration)
- IChecksumProvider (Interface)
- IHashProvider (Interface)
- IDigestProvider (Interface)

Direkt im Core Namespace befindet sich eine Enumeration zur Identifizierung von Befehlstypen (copy, insert u.a.) und drei wichtige Interfaces, nämlich IDigestProvider, IHashProvider und IChecksumProvider. Diese Interfaces stellen die Austauschbarkeit der verschiedenen kryptographischen Algorithmen sicher und sind sehr einfach gehalten, so dass zur Integration eines neuen Algorithmus nur sehr wenige Arbeitsschritte erforderlich sind.

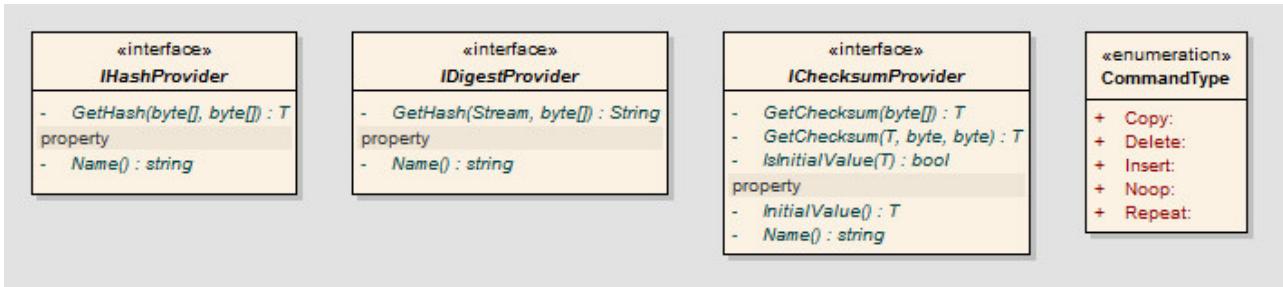


Abbildung 7 – BlueLogic.SDelta.Core. Dieser Namespace beinhaltet die zentralen Interfaces welche die Austauschbarkeit der verschiedenen kryptographischen Algorithmen sicherstellen.

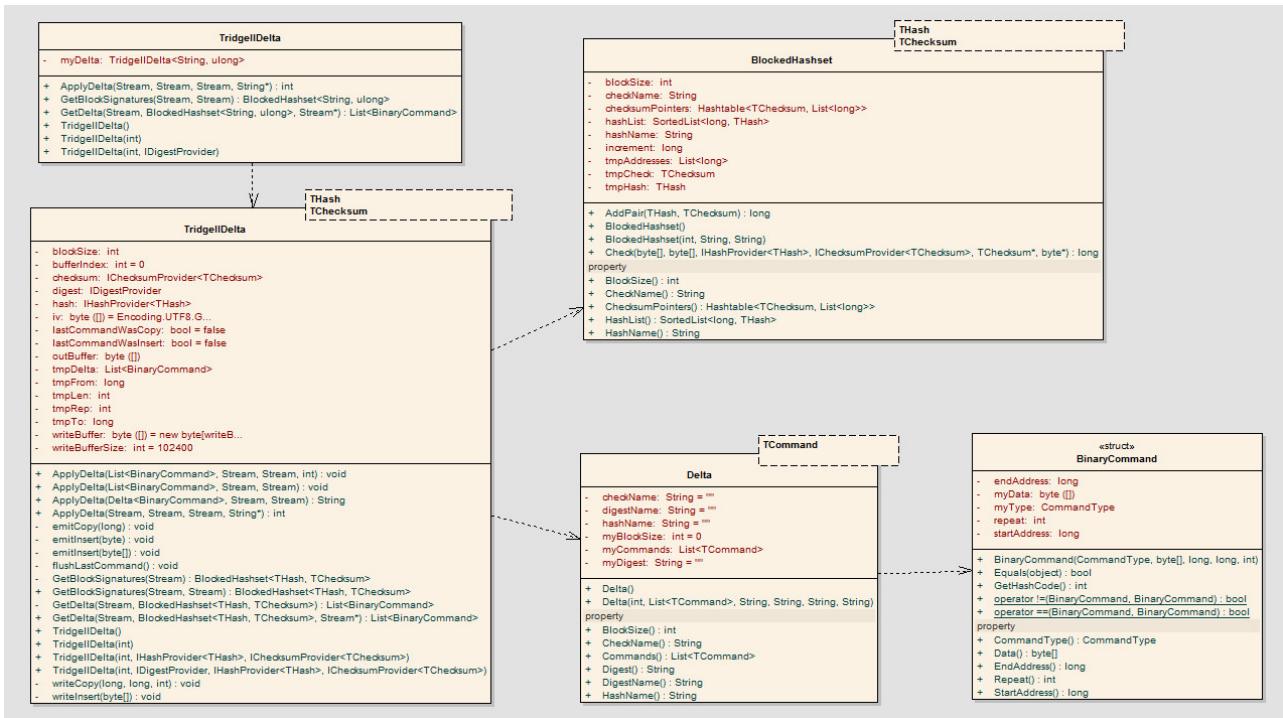


Abbildung 8 – Der Namespace BlueLogic.SDelta.Core.BinaryDelta beinhaltet Klassen, die zur Erzeugung, Anwendung und Speicherung von binären Deltas benötigt werden. Die generische TridgellDelta Klasse wird auch über einen nicht generischen Wrapper angeboten.

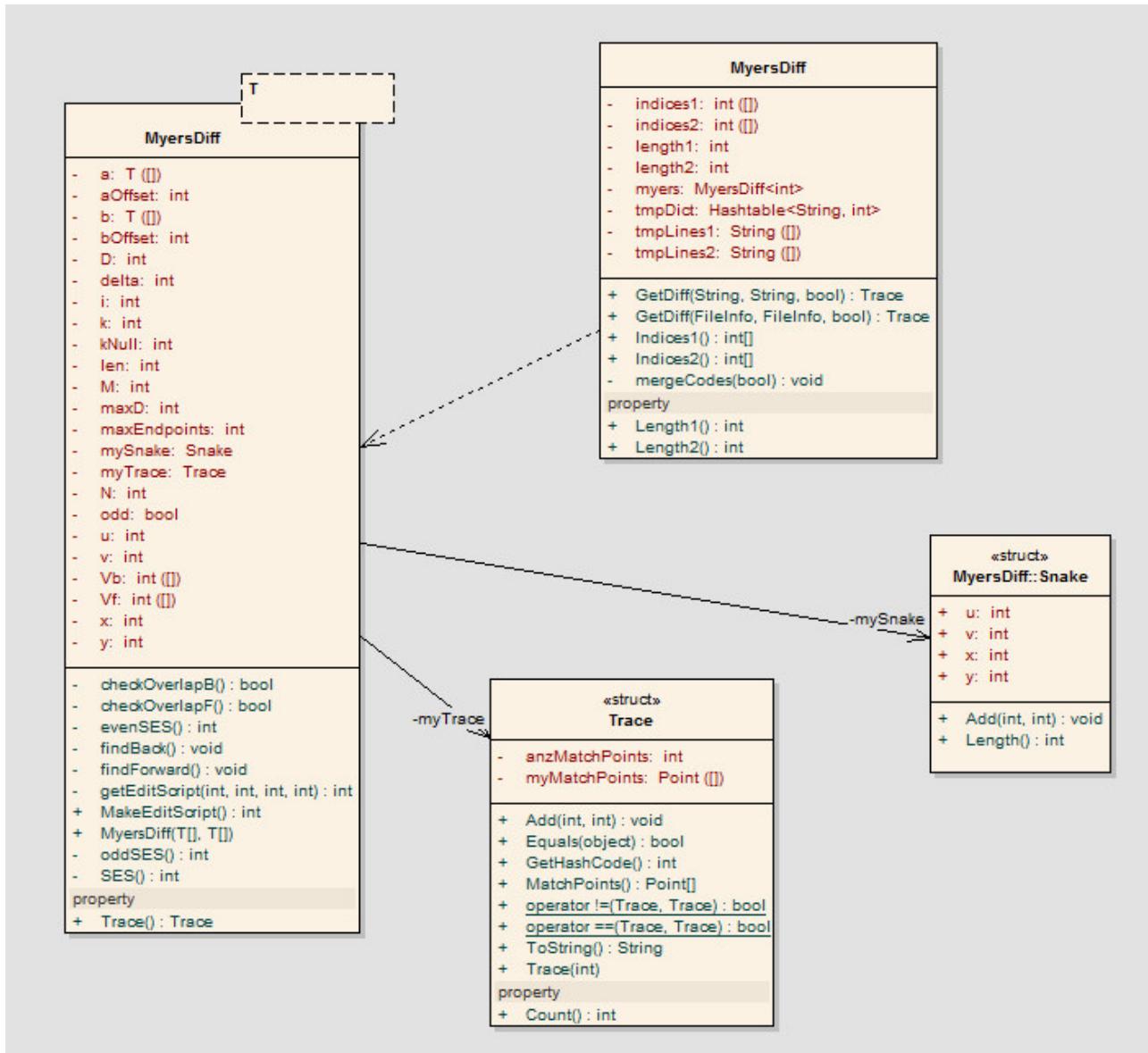


Abbildung 9 – Der Namespace BlueLogic.SDelta.Core.TextDelta enthält den Algorithmus nach Myers sowie die nötigen Erweiterungen, um den Algorithmus zeilenweise auf Textdateien anwenden zu können. Auch hier ist der zentrale Algorithmus generisch parametrierbar und wird zusätzlich über einen nicht-generischen Wrapper angeboten.

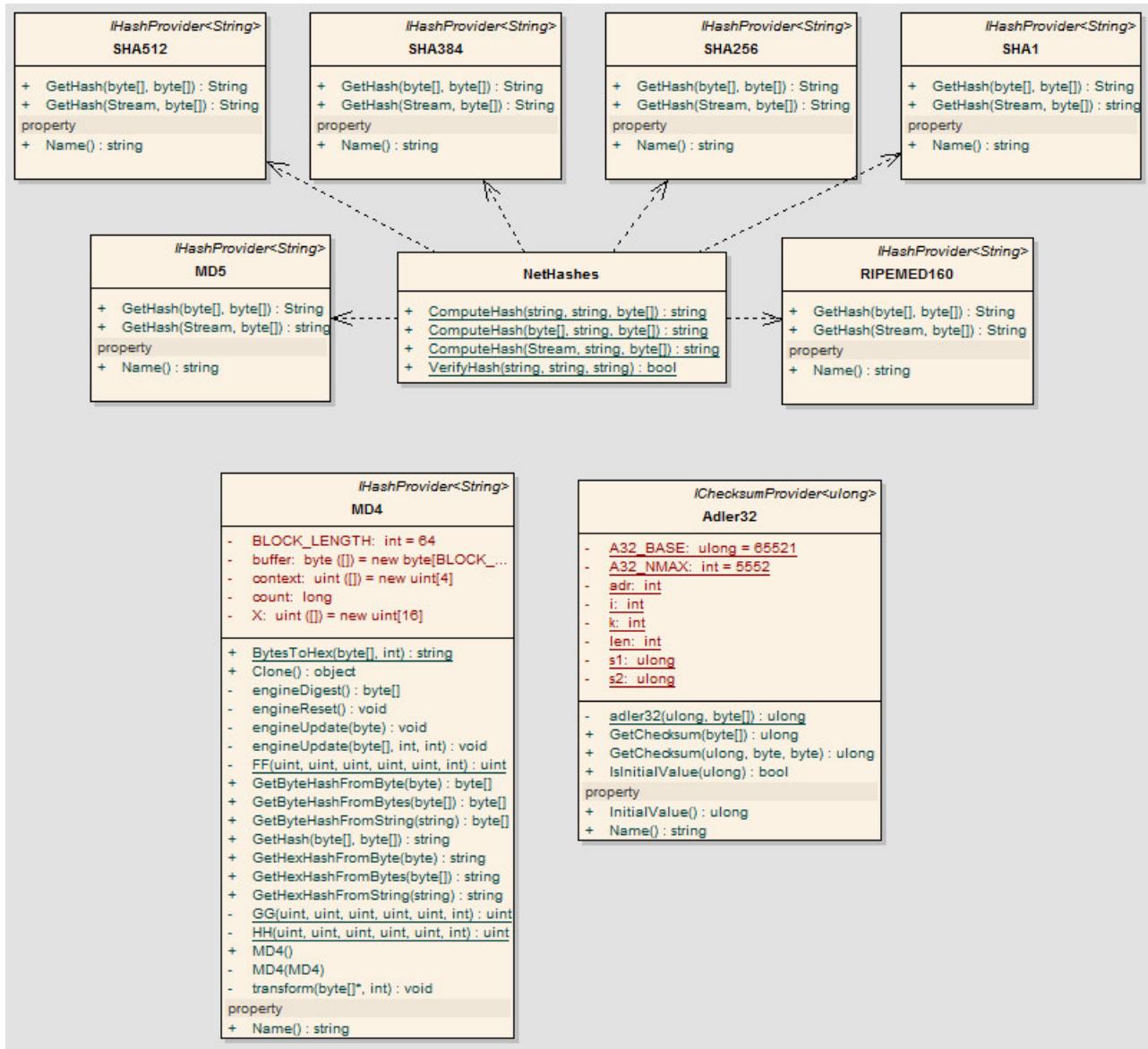


Abbildung 10 - Der Namespace `BlueLogic.SDelta.Core.Cryptography` enthält die benötigten Hash- und Checksummen Funktionen wie zum Beispiel `Adler32` oder `SHA256`. Diese werden bis auf `Adler32` und `MD4` über einen Wrapper (`NETHashes`) vom .NET Framework zur Verfügung gestellt. Die `Adler32` Checksumme wurde vom Autor nach (Deutsch 1996) neu implementiert. Der `MD4` Hash wurde von (Novotny 2000) modifiziert übernommen und wurde ursprünglich von der RSA Data Security, Inc. spezifiziert. (Rivest 1992)



Abbildung 11 – Namespace BlueLogic.SDelta.Core.Utilities: Zwei wichtige Klassen für den Umgang mit Streams. Das Sliding Stream Window stellt einen Datenstrom in Byte-Weise aufeinander folgenden Blöcken zur Verfügung. Dies wird bei der Erzeugung von binären Deltas verwendet. Der PostfixStream dient dazu eine Bytefolge an einen Stream anzuhängen. Dies wird benötigt, um einen Initialisierungsvektor für die kryptographische Hash Berechnung anhängen zu können.

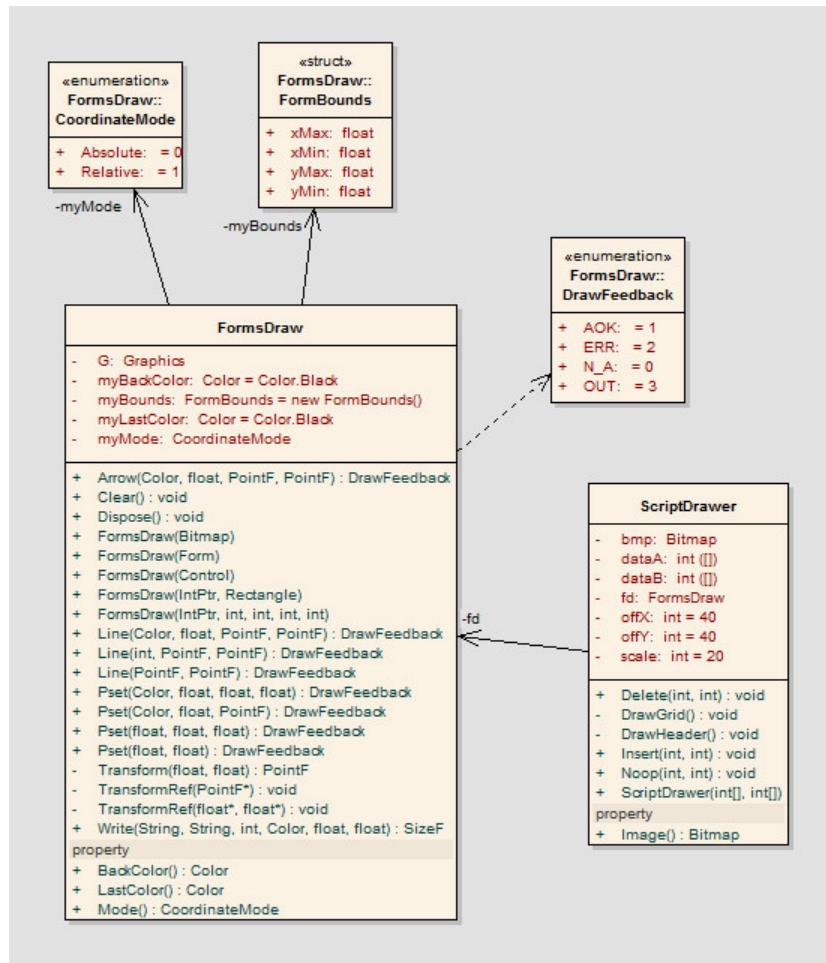


Abbildung 12 – Namespace BlueLogic.SDelta.Core.Utilities: Algorithmen und Datenstrukturen zur graphischen Ausgabe.

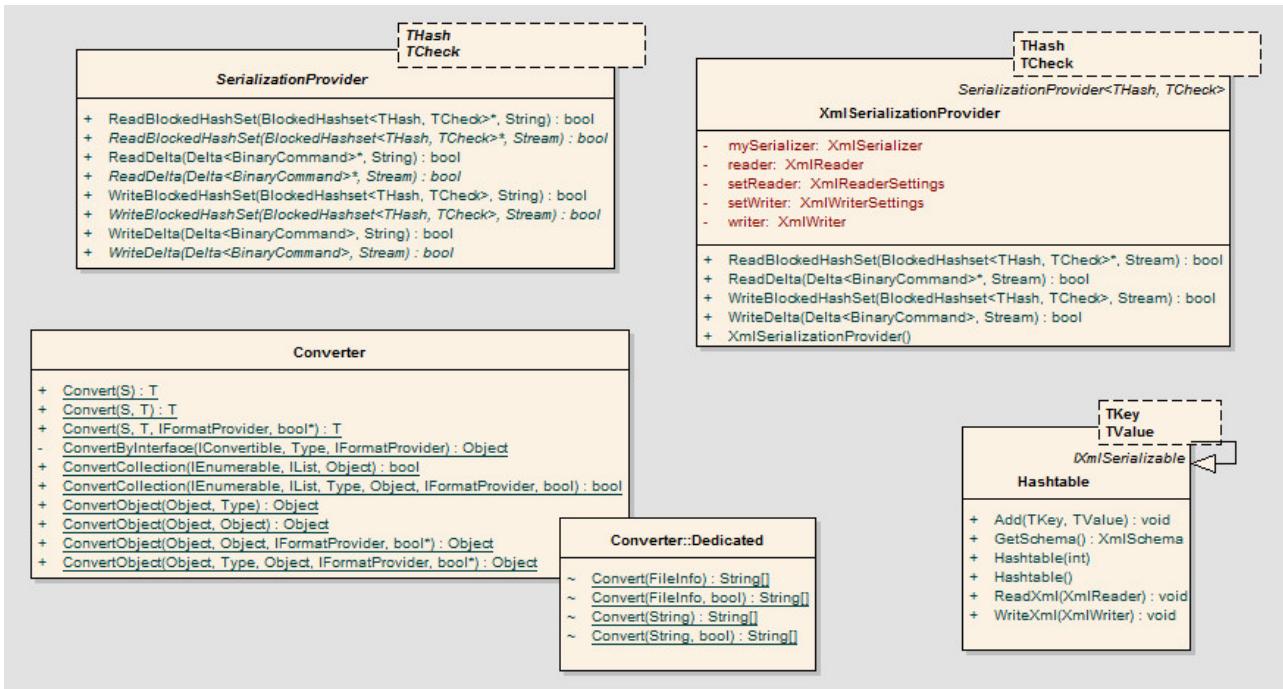


Abbildung 13 – Namespace BlueLogic.SDelta.Core.Utilities, restliche Klassen: Eine generische Hashtable Überladung, Serialisierungs-Provider und eine Konverter Klasse zur Konvertierung von verschiedenen Datentypen. Die Konverter Klasse wurde freundlicherweise von Bernard Glück zur Verfügung gestellt und vom Autor um die Funktionen in der Converter.Dedicated Klasse ergänzt.

Die genaue Verwendung der zentralen Funktionen wird anhand von Quelltextbeispielen in den folgenden beiden Abschnitten erläutert. Eine genauere Beschreibung der einzelnen Klassen, Funktionen und Eigenschaften findet sich außerdem in der Dokumentation welche der Software beiliegt.

4.2 Delta Wizard

Wie schon erwähnt dient der Wizard in erster Linie zur Demonstration und zum Testen der Software. Die manuelle, lokale Generierung von Binär Deltas wird nur selten auf einen Anwendungsfall treffen, außer eben zu Testzwecken. So kann damit nach der optimalen Blockgröße für eine Datei oder Dateimenge gesucht werden oder es kann mit verschiedenen Digest Algorithmen das für ein bestimmtes Anwendungsszenario beste Verhältnis zwischen Sicherheit und Rechenzeit gefunden werden. Im Folgenden wird nun Schritt für Schritt die Funktionsweise des Wizard beschrieben.

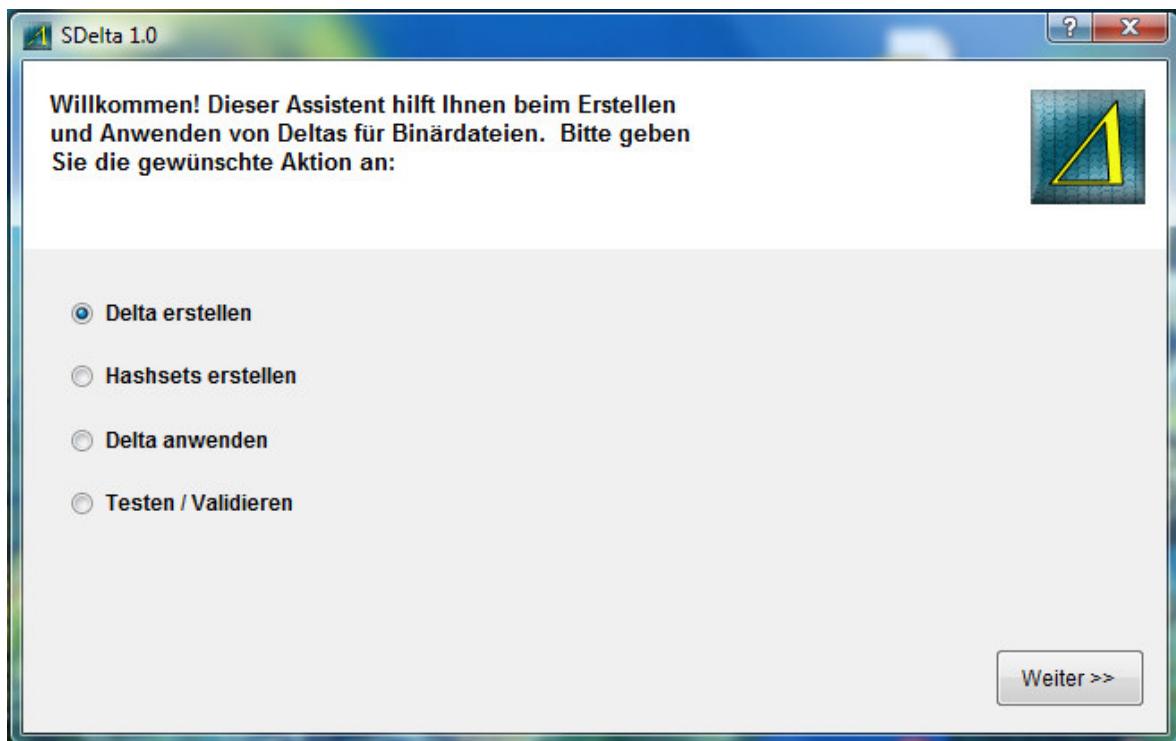


Abbildung 14 – Dialog 1: Aufgabe definieren. In dieser Ansicht kann zwischen den verschiedenen Anwendungsfällen gewählt werden. Außerdem kann über den Hilfe Button ein kleines Informationsfenster angezeigt werden.



Abbildung 15 – Dialog 2: Quell- und Zieldaten. Hier können die Datenquellen und -senken für die nachfolgenden Schritte der jeweiligen Aktion ausgewählt werden. Je nach Aufgabe müssen unterschiedliche Informationen eingegeben werden.

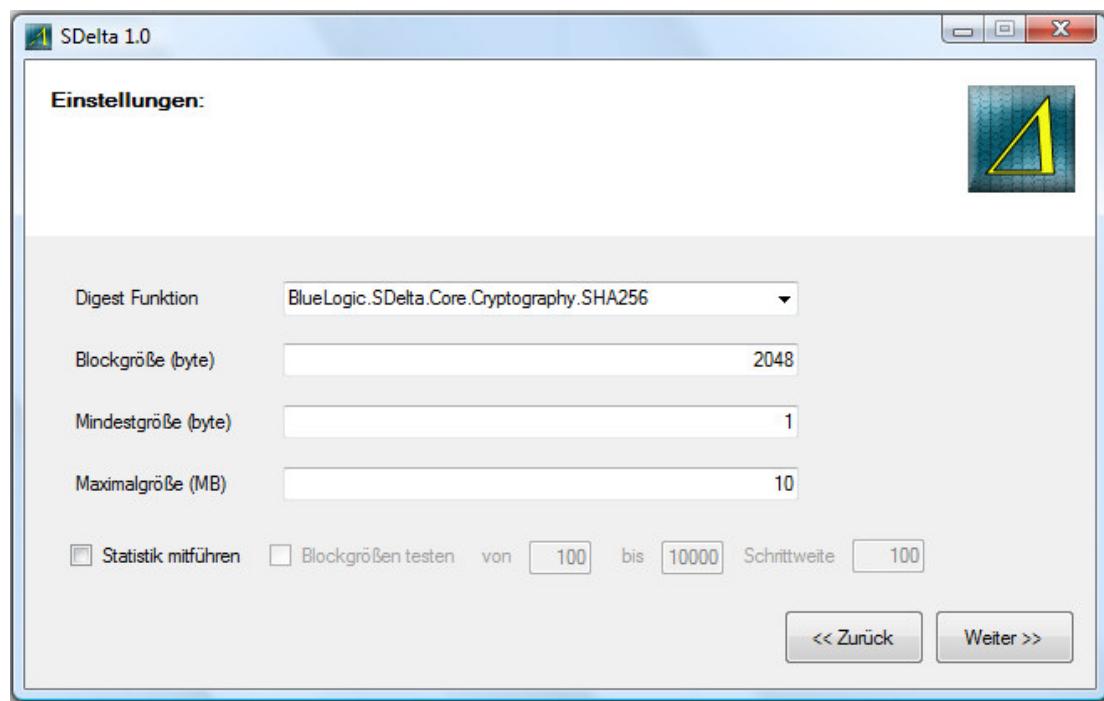


Abbildung 16 – Dialog 3: Einstellungen. In der Combobox zuoberst wird der gewünschte Digest Algorithmus ausgewählt. Standardmäßig wird hier der SHA256 Hash verwendet. Unter Blockgröße kann eine bestimmte Blockgröße eingestellt werden. Diese kann zu Testzwecken auch kleiner als auf das empfohlene Minimum von 2048 Byte eingestellt werden. Darunter können noch Mindest- und Maximalgröße für die Verarbeitung einer Datei angegeben werden. Im Testmodus sind außerdem noch die Schaltflächen für Statistik und Blockgrößentest wie oben sichtbar.

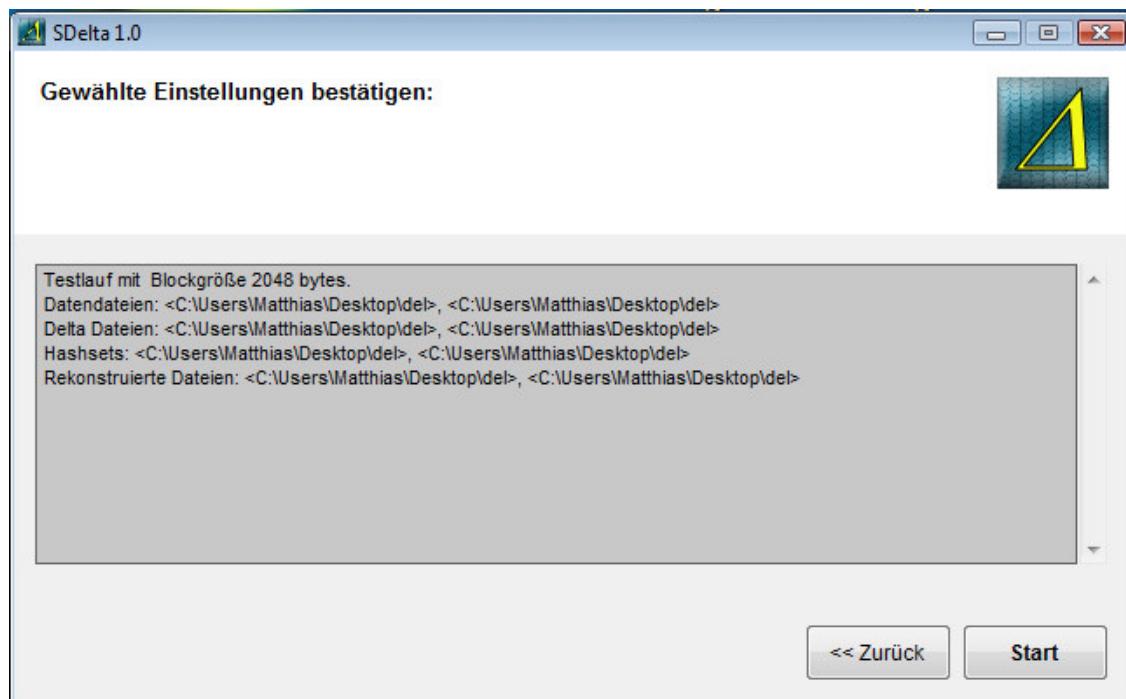


Abbildung 17 – Dialog 4: Bestätigung. Hier werden noch einmal die getätigten Einstellungen zusammengefasst und zur Bestätigung oder Korrektur angezeigt. Beim Betätigen der Start Taste wird der ausgewählte Vorgang gestartet.

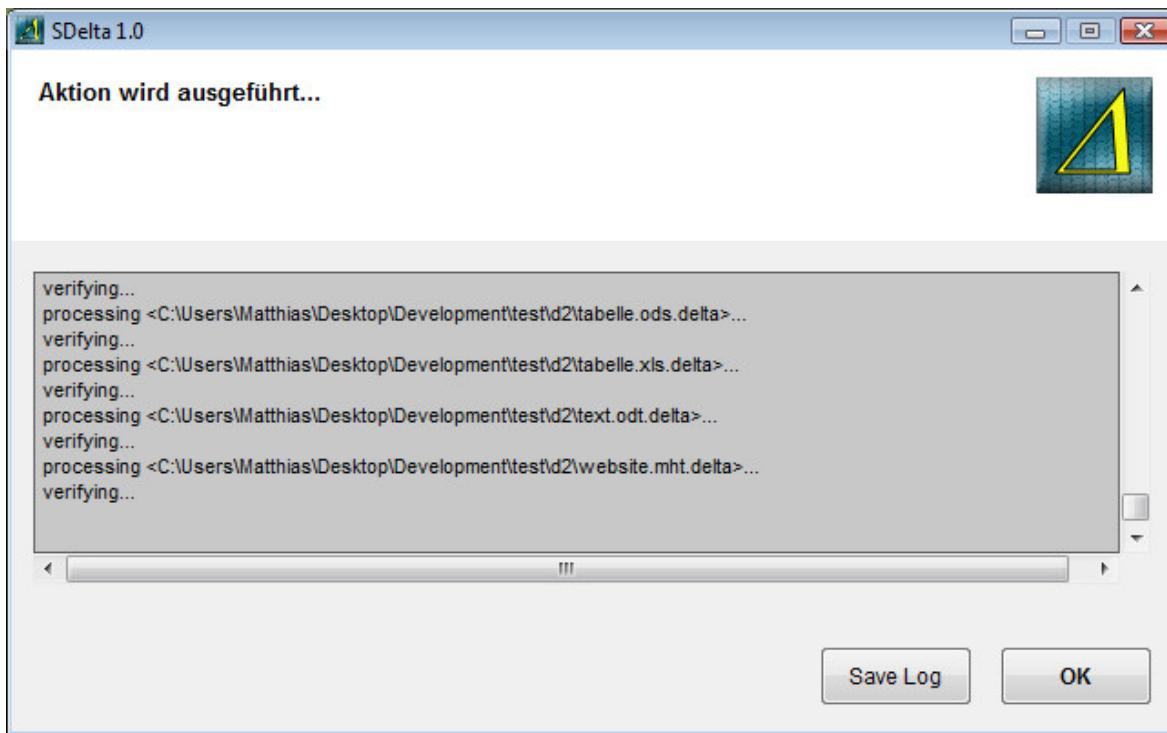


Abbildung 18 – Dialog 5: Testergebnisse. In diesem Dialog kann das ausgegebene Logfile gespeichert werden. Wenn die Checkbox „Statistik“ im Dialog 3 aktiviert wurde, wird nach Betätigen der OK Taste noch ein weiteres Fenster mit Ergebnissen angezeigt.

Schließlich soll hier noch der Code Abschnitt vorgestellt werden, welcher den zentralen Algorithmus parametriert und aufruft. Dieser Code wird im Delta Wizard benutzt, um den Algorithmus aufzurufen:

```
BlockedHashSet<String, ulong> bhs;
TridgellDelta td = new TridgellDelta(blockSize, digestProvider);
bhs = td.GetBlockSignatures(source, target);
List<BinaryCommand> delta = td.GetDelta(target, bhs, ref result);
```

Dabei wird nur einer von mehreren möglichen Konstruktoren benutzt. Weitere Informationen über die Verwendung der Klassen und Interfaces befinden sich in der Dokumentation.

4.3 Diff Wizard

Auch dieser Wizard ist eher zu Demonstrations- und Testzwecken gedacht. Wenn auch ein eigenständiges Programm zum Vergleich von Textdateien mitunter sehr nützlich sein kann, ist der vorliegende Wizard noch zu wenig ausgereift, um in dieser Art und Weise eingesetzt zu werden. Es fehlt an einem guten integrierten Text Editor und auch die Parametrierbarkeit mit Hilfe der GUI ist stark eingeschränkt. Außerdem wären Syntax

Highlighting und automatische Spracherkennung bei Code Dateien für ein solches Tool unverzichtbar. All diese Features hätten jedoch den Rahmen dieser Arbeit deutlich gesprengt und waren daher von Beginn an nicht vorgesehen.

Der Wizard verfügt grundsätzlich über zwei Funktionen zur Visualisierung der berechneten Edit Scripts: Zeichnerisch, wie in (Myers 1986), oder – wie in den meisten Text Delta Anwendungen üblich – anhand der Textdateien durch entsprechende Markierung der unveränderten, gelöschten und eingefügten Zeilen.

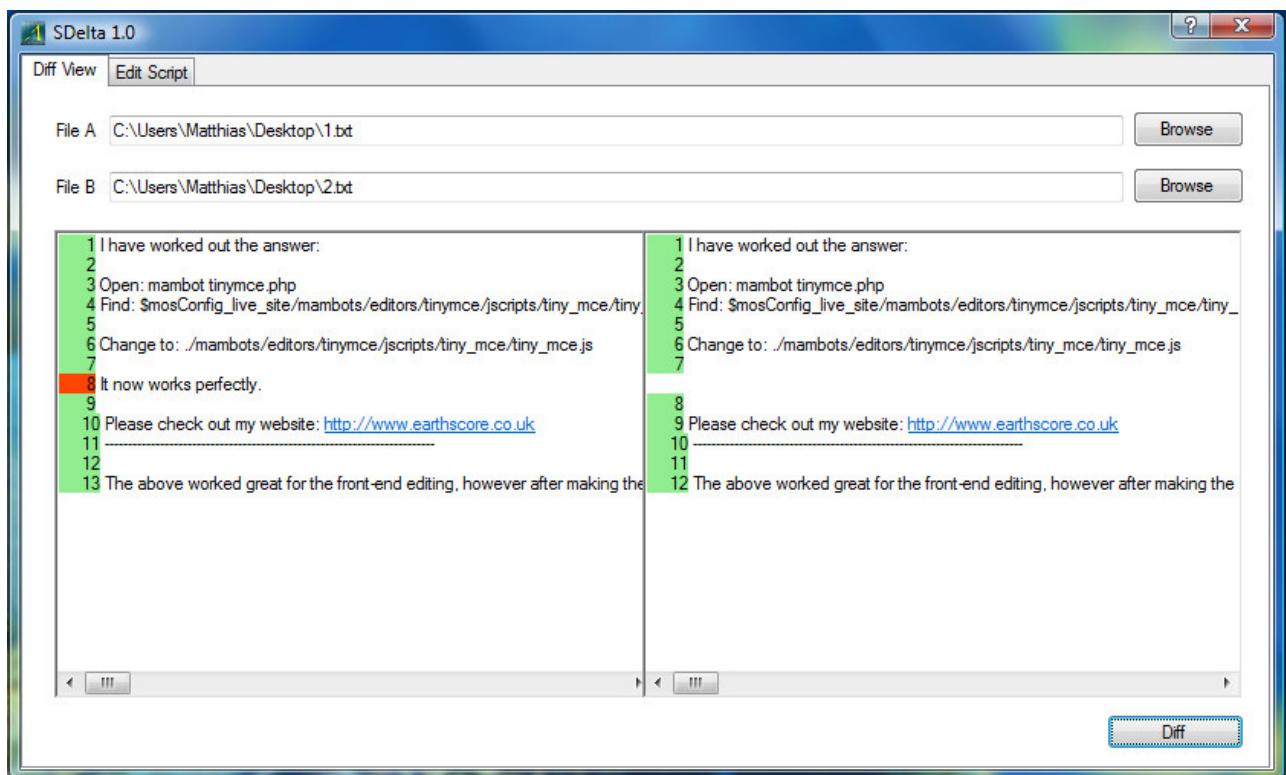


Abbildung 19 – Text Delta Wizard: Darstellung von unterschiedlichen Textversionen. Die beiden Texte in der Abbildung unterscheiden sich nur durch eine einzige Zeile, welche auf der linken Seite rot markiert ist. Dies bedeutet, dass aus dem Text auf der linken Seite nur diese eine Zeile entfernt werden muss, um daraus den Text auf der rechten Seite zu erhalten.

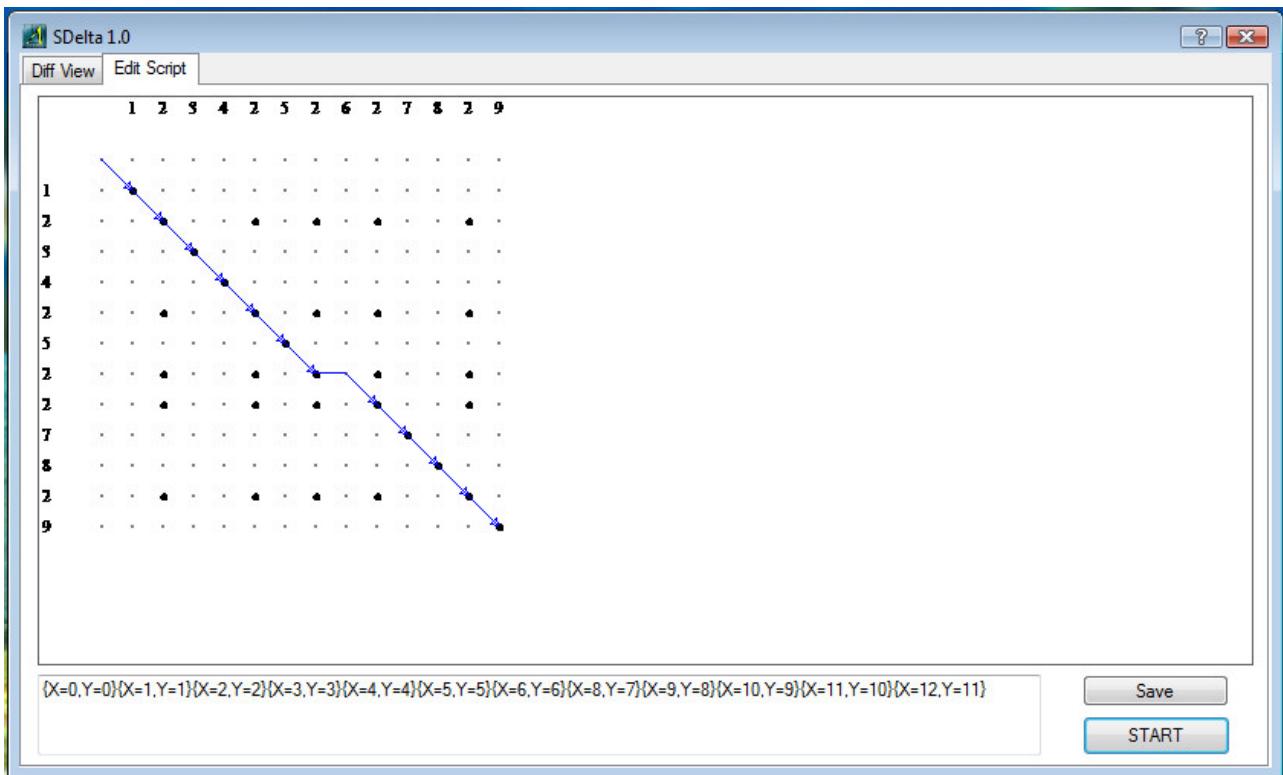


Abbildung 20 – Text Delta Wizard: Darstellung von Edit Scripts zu den gleichen Daten wie in Abbildung 18. Zusätzlich werden die Match Points, die auf dem Pfad liegen (Trace) in einer Textbox ausgegeben. Mit dem Save Button kann das Bild als PNG Datei gespeichert werden.

Zu beachten: der Aufwand zur Bestimmung der Match Points für das graphische Edit Script beträgt $O(N^*M)$, so dass dieses Feature bereits deutlich vor dem eigentlichen Algorithmus an seine Grenzen stößt. (Am PC des Autors beträgt diese Grenze ungefähr 500 Zeilen.) Für eine Text Datei mit 1.000 Zeilen muss eine Bitmap von 30.000 X 30.000 Pixel erzeugt werden und es müssen 1.000.000 Indexvergleiche durchgeführt werden, um die entsprechenden Match Points einzuziehen.

Abschließend soll hier noch der Code Abschnitt vorgestellt werden, welcher den zentralen Algorithmus parametriert und aufruft. Dieser Code wird im Wizard benutzt, um den Algorithmus aufzurufen:

```
MyersDiff myers = new MyersDiff();
Trace t = myers.GetDiff(file1, file2, true);
```

Anschließend wird aus der Trace eine Textanzeige oder ein graphisches Edit Script generiert. Auch hier kommt nur eine von mehreren möglichen Varianten zum Einsatz. Weitere Konstruktoren und Methoden finden sich in der Dokumentation.

4.4 Performance

Folgende Tabelle gibt Übersicht über einige grundlegende Performance Parameter, die von der Software selbst erfasst werden. Die Ergebnisse beziehen sich auf einen Intel Centrino Duo T2500 @ 2*2.00 GHz (Leistungsbewertung 4.8) mit 2048 MB RAM (Leistungsbewertung 4.9) und einer Toshiba MK1234GSX ATA Festplatte (Leistungsbewertung 4.3) unter Microsoft Windows Vista 32 bit Edition. Dabei ist allerdings zu beachten, dass die Software nur einen der beiden Prozessorkerne nutzen kann. Außerdem ist das Gerät ein Notebook, was bedeutet, dass die Prozessorleistung stärkeren Schwankungen unterworfen ist als bei einem Standgerät, obwohl die Tests bei Netzbetrieb durchgeführt wurden. Daher sind die Throughput Ergebnisse jeweils über drei Durchläufe gemittelt und auf Zehntel gerundet.

Datensets	Blockgröße	Throughput	CR
A	256	7,9	63
A	512	6,9	65
A	768	6	45
A	1024	4,7	35
A	2048	4,3	34
A	3072	3,6	13
B	256	11,7	17
B	2048	8,6	19
C	256	7	13
C	2048	4,8	13
D	256	6	31
D	2048	7,5	31

Tabelle 8 – Performance Ergebnisse mit dem Binärdelta Wizard. Die Blockgröße ist in Byte angegeben, der durchschnittliche Throughput (Datendurchsatz) in Megabyte pro Sekunde. Der Datendurchsatz bezieht sich auf die Generierung von Deltas und nicht auf die Rekonstruktionsoperation, welche wesentlich schneller läuft. CR steht für „compression ratio“, durchschnittliche Kompressionsrate = Größe_{unkomprimiert} / Größe_{Delta}.

5. Diskussion

Wissenschaftliche Software wird in vielen Fällen nicht über den Alpha- beziehungsweise Prototyp - Zustand hinaus entwickelt. Dies liegt daran, dass viele Implementierungen nur einen „proof of concept“ darstellen um dann oft von anderen, teils kommerziellen Anwendungen ersetzt zu werden. Manche Pakete befinden sich auch im ständigen Experimentierstadium oder sind für Hardware geschaffen, die noch nicht verfügbar ist. Nur verhältnismäßig wenige Pakete erreichen den Release Status. Das hier beschriebene Paket kann allerdings sehr wohl als Release bezeichnet werden. Innerhalb der beschriebenen Parameter ist das Paket sehr zuverlässig und hinreichend performant. Die Fehlerfreiheit wird zum einen durch die durchgeführten Tests und zum anderen durch kryptographische Sicherheit gewährleistet, so dass ein verantwortungsvoller Einsatz auch im medizinischen Umfeld oder in anderweitig juristisch sensiblen Bereichen möglich ist.

Allerdings gibt es auch eine Reihe von Einschränkungen, die hier besprochen werden sollen:

- Das Binärdelta Paket kann nur Dateien bis zu einer Größe von wenigen hundert Megabyte (je nach Konfiguration des eingesetzten Rechners) verarbeiten. Um noch höhere Skalierbarkeit zu gewährleisten muss die Software so umgeschrieben werden, dass Deltas und Hashsets nicht mehr im Arbeitsspeicher gehalten werden. Dazu ist allerdings eine andere Architektur insbesondere in Bezug auf das BlockedHashSet erforderlich, welches mit Caching ausgestattet werden müsste. Die Performance bei kleinen Dateien kann dadurch allerdings Einbußen erleiden, daher ist eine reine Stream Implementierung nur sinnvoll, wenn damit hauptsächlich große Dateien (über der Größe von gewöhnlichen Textdokumenten) verarbeitet werden sollen.
- Das Textdelta Paket kann nur Dateien bis zu einer Größe von wenigen hundert KB verarbeiten (je nach Ausstattung des benutzten Rechners). Um auch für sehr große Dateien ein Edit Script zu berechnen wird ein grundlegend anderer Algorithmus benötigt, beispielsweise aus dem dynamic programming. Möglicherweise lassen sich auch Wege finden, wie beide in dieser Arbeit vorgestellten Algorithmen geschickt kombiniert werden können.
- Die erreichte Performance beim binären Delta Algorithmus ist zwar für online Anwendung leicht ausreichend, für die lokale Anwendung wäre aber eine höhere

Performance wünschenswert, da SDelta diesbezüglich noch merklich hinter anderen etablierten Produkten steht.

- Die XML Serialisierung erzeugt durch die XML Tags einen gewissen Overhead, der bei einer binären Serialisierung auf einen Bruchteil der Größe sinkt. Damit könnte die Kompressionsrate des binären Delta Algorithmus weiter verbessert werden.
- Die generierten Datenformate sind einfache XML Dateien, welche nicht weiter komprimiert werden. Je nach Anwendungsszenario sollten zumindest zusätzliche Komprimierungsschritte in Betracht gezogen werden. Außerdem sollte ein binäres oder MIME basiertes Serialisierungsformat hinzugefügt werden.
- Für hohe Performance sollten die in der Profiling Session als Bottlenecks identifizierten Funktionen in einer nativen Sprache wie C++ neu programmiert werden. Die deutlichste Performance Engstelle die im Laufe des Profiling identifiziert werden konnte ist die Check(...) Funktion in der Klasse BlockedHashSet. Beim Text Delta ist dies schwieriger zu analysieren, da je nach Edit Script unterschiedliche Funktionen den größten Aufwand verursachen. Allerdings wird dieser Algorithmus ohnehin schon durch den Speicherverbrauch begrenzt, bevor die Rechenzeit auf einem aktuellen System auffällig lange wird.

Es muss an dieser Stelle auch noch darauf hingewiesen werden, dass die unter Kapitel 4.4 in Tabelle 9 aufgelisteten Kompressionsraten nicht mit den Kompressionsraten von herkömmlicher Komprimierungssoftware vergleichbar sind. Gerade bei kleinen Dateien generiert der Delta Algorithmus häufig Deltas, die in XML Form sogar größer sind, als die Originaldatei, was einer Kompressionsrate < 1 entspricht. Dies ist zum einen für Softwareimplementierungen wichtig, die einen Mechanismus benutzen sollten, der in ungünstigen Fällen die Delta Berechnung abbrechen kann um stattdessen die Originaldaten zu übermitteln. Zum anderen ist dadurch während der Testläufe aufgefallen, dass die Kompressionsrate nicht nur von der Blockgröße und dem Ausmaß des Unterschiedes zwischen den Versionen abhängt, sondern in starkem Maße auch vom genauen Dateiformat. Dies ist ein Umstand, der genauer ausgewertet werden sollte, bevor ein Delta Algorithmus wie der von Tridgell in eine Anwendung integriert wird, die hohe Skalierbarkeit und Performance benötigt.

Weiters möchte ich noch kurz auf die Benutzerfreundlichkeit („Usability“) der GUI Applikationen eingehen. Diese stehen nicht im Vordergrund der vorliegenden Arbeit und wurden daher noch nicht im selben Ausmaß getestet und optimiert. Es ist zu erwarten, dass Skalierungsprobleme oder leichte Fehler auftauchen, wenn die Software auf verschiedenen Computersystemen und unter verschiedenen Bedingungen eingesetzt wird.

Für beide Applikationen gibt es außerdem einen Fehlerfall, der absichtlich nicht abgefangen wird, um die Grenze der Skalierbarkeit auch auf besser ausgestatteten Systemen ausloten zu können:

- Beim Textdelta Wizard tritt ab einer bestimmten Dateigröße, abhängig von der Länge des kürzesten Edit Script und vom verfügbaren Speicher ein „freeze“ - Fehler auf, das heißt die Anwendung reagiert nicht mehr auf Benutzerinteraktion und muss terminiert werden.
- Beim Binärdelta Wizard tritt derselbe Effekt auf, wenn das BlockedHashSet eine bestimmte, systemabhängige Größe überschreitet. Dies hängt von der Dateigröße sowie vom Ausmaß der Unterscheidung zwischen den Dateiversionen ab.

Die vom Autor beobachteten freeze Fehler sind in beiden Fällen der graphischen Oberfläche zuzuschreiben und nicht der Core Library. In einigen Fällen wird die Anwendung auch fortgesetzt, wenn man lange genug wartet. Dies soll aber nicht darüber hinweg täuschen, dass der Umstand ein Usability Problem darstellt und für ein kommerzielles Produkt selbstverständlich nicht so belassen werden kann. Die einzige Alternative in diesen Fällen wäre eine automatische Begrenzung, was aber anwendungsorientierten Implementierungen überlassen werden soll, da eine solche Begrenzung auf den konkreten Anwendungsfall abgestimmt werden muss. Vor allem in Fällen, wo die Hardware im Vorhinein bekannt ist – wie zum Beispiel wenn zertifizierte Hardware eingesetzt wird – ist ein solcher Schritt erst nach Auswahl und Testung der Hardware sinnvoll.

Als gelungen kann jedenfalls die Erweiterbarkeit insbesondere in Bezug auf kryptographische Kernelemente der Software bezeichnet werden, eine Art von Parametrierbarkeit welche ohne Generics nicht so einfach durchführbar gewesen wäre.

6. Ausblick

Neben der Behebung der in Kapitel 5 erwähnten Mängel gibt es noch eine Reihe weiterer Verbesserungs- und Erweiterungsmöglichkeiten, die hier kurz beleuchtet werden sollen:

VCDiff	Das Implementieren einer VCDiff (Korn 2002) Schnittstelle würde die generierten binär Deltas auch anderen Softwarepaketen zugänglich zu machen.
File System Support	Um als Grundlage eines Delta basierten Dateisystems eingesetzt werden zu können, muss das Framework um eine Reihe von Schnittstellen erweitert werden und für hohe Skalierbarkeit ausgerichtet werden.
Office Support	Insbesondere für Microsoft Word wäre eine gesonderte Implementierung nützlich. Derzeit können Word kodierte Dateien nicht anhand eines Text Delta visualisiert werden.
http, WebDAV	Für Webanwendungen sowie für WebDAV Anwendungen wie Microsofts Webfolder sind noch spezielle Erweiterungen vor allem in Bezug auf Locking nötig.
VCS	Um für Version Control Systeme eingesetzt zu werden sollte das Framework um entsprechende Serialisierer und Deserialisierer erweitert werden. Außerdem ist eine weitere Kompression in Betracht zu ziehen.
Encryption	Für tiefer gehende Sicherheit könnte das Paket um symmetrische und asymmetrische Verschlüsselungsmethoden erweitert werden.
Komprimierung	Zusätzliche Komprimierungsalgorithmen würden das Paket flexibler im Einsatz machen.
Parallelisierung	Durch geschicktes Multithreading könnten Teile der Software parallelisiert werden, was auf heute üblichen Multicore Prozessoren sowie auf anderen Mehrprozessor-Systemen einen Performancegewinn bringen würde.

Tabelle 9 – Übersicht über einige Verbesserungs- und Erweiterungsmöglichkeiten mit denen Performance, Kompressionsrate, Einsatzgebiet und andere wichtige Faktoren der Software optimiert werden können.

In weiterer Folge wird das im Zuge dieser Arbeit entwickelte Delta Paket in einer kommerziellen Dokumentenmanagement Software bei BlueLogic Software Solutions (BlueLogic 2007) zur Anwendung gebracht. Dies wird zweifellos weitere Erkenntnisse und Verbesserungsmöglichkeiten aufzeigen.

7. Danksagung

Mein Dank gilt in erster Line dem Betreuer dieser Arbeit, Hon. Prof. Dr. Roland Blomer, welcher mir mit zahlreichen Referenzen, Hinweisen, Korrekturen und konstruktiver Kritik behilflich war. Gerade bei einem interdisziplinären Thema wie diesem ist es oft nicht leicht, einen Betreuer zu finden, der in beiden Gebieten versiert ist oder bereit ist sich innerhalb von kürzester Zeit in ein möglicherweise neues Gebiet einzulesen.

Weiters geht ein herzliches Dankeschön an meinen Geschäftspartner Bernhard Glück, welcher mir während des gesamten Projekts hilfreich beiseite stand und an der einen oder anderen Stelle auch wertvolle Beiträge zur Software lieferte.

Außerdem möchte ich an dieser Stelle meinen Eltern danken, die dieses Studium durch ihre erzieherische, finanzielle und nicht zuletzt auch moralische Unterstützung überhaupt erst ermöglichten.

Nicht vergessen werden soll hier auch meine Lebensgefährtin Susanne, die während meiner Arbeit viele Stunden auf mich verzichten musste und mich dennoch stets aufmuntern konnte, wenn es gerade nicht so gut lief.

Und zu guter letzt möchte ich auch meinen Wohnungsnachbarn, der Familie Eller danken, die aus mir nicht nachvollziehbaren Gründen nicht nur meine laute Musik sondern sogar mein ohrenbetäubendes Wing Tsun Holzpuppentraining seit Jahren anstandslos über sich ergehen lässt.

8. Verzeichnisse

8.1 Literatur

Beck, K., Cynthia Andres (2004). Extreme Programming Explained. Embrace Change. Amsterdam, Addison-Wesley Longman.

Bellare, M., Tadayoshi Kohno (2002). Hash Function Balance and its Impact on Birthday Attacks. Dept. of Computer Science & Engineering, University of California at San Diego.

Bellare, M., Thomas Ristenpart (2006). "Multi-Property-Preserving Hash Domain Extension - The EMD Transform." NIST Second Cryptographic Hash Workshop August, 2006.

Bellovin, S. M., Eric K. Rescorla (2005). Deploying a New Hash Algorithm. NIST Hash Function Workshop, Department of Computer Science Columbia University

Bentahar, K., D. Page, J. H. Silverman, M.-J. O. Saarinen, N. P. Smart (2006). "LASH." HASH WORKSHOP 2006.

BlueLogic. (2007). "BlueLogic Software Solutions." 2007, from <http://www.bluelogic.at>.

Chrysler, C. T. (1998). "Case Study - Chrysler goes to "Extremes"."

Commons, C. (2007). "Creative Commons (CC) Licences." Retrieved 22. 3., 2007, from <http://creativecommons.org/>.

Contini, S., YiqunLisa Yin (2006). "Forgery and Partial Key Recovery attacks on HMAC and NMAC using Hash Collisions." 2nd NIST Hash Function Workshop.

DeMichiel, L., Michael Keith. (2006). "JSR 220: Enterprise JavaBeansTM, Version 3.0." Retrieved 9. 6. 2007, from <http://jcp.org/aboutJava/communityprocess/final/jsr220/index.html>.

Deutsch, L. P., Jean-Loup Gailly (1996). RFC 1950 - ZLIB Compressed Data Format Specification version 3.3, Network Working Group.

DotGNU Project. (2007). "Portable.NET." from <http://dotgnu.org/pnet.html>.

ECMA (334). C# Language Specification 4th Edition. ISO/IEC 23270:2006(E) 2nd Edition.

ECMA (335). Common Language Infrastructure (CLI) 4th Edition. ISO/IEC 23271:2006(E) 2nd Edition 2006.

Fluhrer, S., Mantin, I., Shamir, A (2001). "Weaknesses in the Key Scheduling Algorithm of RC4 " Preliminary Draft, 25.7.01.

FSF, Free Software Foundation I. (1991). "GNU General Public Licence (Version 2)." Retrieved 13. 3., 2007, from <http://www.gnu.org/licenses/gpl.html>.

Gauravaram, P., William Millan, Ed Dawson, Kapali Viswanathan (2006). "Constructing Secure Hash Functions by Enhancing Merkle-Damgard Construction." ACISP 2006.

Gligoroski, D., Smile Markovski, Ljupco Kocarev (2006). "Edon-R, An Infinite Family of Cryptographic Hash Functions." NIST 2nd CHW.

Haenni, R. (2006). Kryptographie in Theorie und Praxis. Bern, Hochschule für Technik und Informatik Biel.

Halevi, S., Hugo Krawczyk (2007). Strengthening Digital Signatures via Randomized Hashing. Crypto 2006, IBM T.J. Watson Research Center.

Hawkes, P., Michael Paddon, and Gregory G. Rose (2004). "On Corrective Patterns for the SHA-2 Family." Cryptology ePrint Archive Report 2004/207.

Hirai, Y., Takashi Kurokawa, Shin'ichiro Matsuo, Hidema Tanaka, Akihiro Yamamura (2006). "Classification of Hash Functions Suitable for Real-life Systems." 2nd Hash Workshop.

Hunt, J. J., Kiem-Phong Vo, Walter F. Tichy (1998). "Delta algorithms: an empirical analysis" ACM Transactions on Software Engineering and Methodology (TOSEM) archive 7(2): 192 - 214.

IEEE, Institute of Electrical and Electronics Engineers I. (1999). Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications. ANSI/IEEE Std 802.11, 1999 Edition (R2003).

Jean-Loup Gailly, M. A., Greg Roelofs. (2006, 4. Mai 2007). "ZLib online Dokumentation." from http://www.zlib.net/zlib_tech.html.

Kent Beck, C. A. (2004). Extreme Programming Explained. Embrace Change. Amsterdam, Addison-Wesley Longman.

Korn, D., J. MacDonald, J. Mogul, K. Vo (2002). "RFC 3284 - The VCDIFF Generic Differencing and Compression Data Format." Network Working Group.

MacDonald, J. P. (1998). Versioned File Archiving, Compression and Distribution. Department of Electrical Engineering and Computer Sciences. Berkeley CA, University of California at Berkeley.

MacDonald, J. P. (2000). File System Support for Delta Compression. Department of Electrical Engineering and Computer Science. Berkeley, University of California. **Master Thesis**.

Microsoft. (2007). "Sandcastle CTP 2007." from <http://blogs.msdn.com/sandcastle/>.

Myers, E. W. (1986). "An O(ND) Difference Algorithm and Its Variations." Algorithmica 1(2): 251.

NIST, P. B., Joan Hash, Mark Wilson, Ed. (2006). Information Security Handbook: A Guide for Managers - Recommendations of the National Institute of Standards and Technology. NIST Special Publication 800-100. Gaithersburg, MD 20899-8930.

Novell. (2007). "The Mono Project." from <http://www.mono-project.com>.

Novotny, O., Norbert Hranitzky. (2000). "MD4 Message-Digest Algorithm, derived from the RSA Data Security, Inc." from <http://www.rsasecurity.com>.

Potter, S. (2005). Using Binary Delta Compression (BDC) Technology to Update Windows XP and Windows Server 2003. Windows Server System. B. Birger, Microsoft Corporation.

Rivest, R., MIT Laboratory for Computer Science and RSA Data Security, Inc. (1992). "RFC 1320 - The MD4 Message Digest Algorithm."

Stevens, M. (2006). Fast Collision Attack on MD5. Department of Mathematics and Computer Science, Eindhoven University of Technology.

Stone, J., R. Stewart, D. Otis (2002). RFC 3309 - Stream Control Transmission Protocol (SCTP) Checksum Change, Network Working Group.

Sujecki, B. (2005). "Vertrags- und urheberrechtliche Aspekte von Open Source Software im deutschen Recht." JurPC Web-Dok. 145 Retrieved 13. 3., 2007, from <http://www.jurpc.de/aufsatz/20050145.htm>.

Thoughtworks. (2007). "Thoughtworks Website." from <http://www.thoughtworks.com>.

Till Jaeger, O. K., Till Kreutzer, Axel Metzger, Carsten Schulz (2005). Die GPL kommentiert und erklärt. Köln, O'Reilly Verlag.

Trendafilov, D. (2002). "The zdelta Home Page." from <http://cis.poly.edu/zdelta/results.shtml>.

Tridgell, A. (1999). Efficient Algorithms for Sorting and Synchronization, Australian National University. **PhD**.

van Oorschot, P. C., Michael J. Wiener (2004). "Parallel Collision Search with Application to Hash Functions and Discrete Logarithms."

Wang, X., Dengguo Feng, Xuejia Lai, Hongbo Yu (2004). "Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD." Cryptology ePrint Archive Report 2004/199.

Wang, X., Yiqun Lisa Yin, Hongbo Yu (2005). Collision Search Attacks on SHA1. Shandong University, China.

Yuval, G. (1979). "How to Swindle Rabin." Cryptologia 3 187-189, Jul. 1979.

8.2 Abbildungen

Abbildung 1 – Edit Script	Seite 12
Abbildung 2 – Text Diff	Seite 14
Abbildung 3 – Auszug aus der Quellcode Dokumentation	Seite 33
Abbildung 4 – Code Coverage Resultate	Seite 34
Abbildung 5 – Text Delta Unit Test Beispiel	Seite 38
Abbildung 6 – Profiling Ergebnisse	Seite 43
Abbildung 7 – Klassendiagramm: BlueLogic.SDelta.Core	Seite 46
Abbildung 8 – Klassendiagramm: BlueLogic.SDelta.Core.BinaryDelta	Seite 46
Abbildung 9 – Klassendiagramm: BlueLogic.SDelta.Core.TextDelta	Seite 47
Abbildung 10 – Klassendiagramm: BlueLogic.SDelta.Core.Cryptography	Seite 48
Abbildung 11 – Klassendiagramm: BlueLogic.SDelta.Core.Utilities 1	Seite 49
Abbildung 12 – Klassendiagramm: BlueLogic.SDelta.Core.Utilities 2	Seite 49
Abbildung 13 – Klassendiagramm: BlueLogic.SDelta.Core.Utilities 3	Seite 50
Abbildung 14 – Delta Wizard: Dialog 1 - Aufgabe definieren	Seite 51
Abbildung 15 – Delta Wizard: Dialog 2 - Quell und Zieldaten	Seite 51
Abbildung 16 – Delta Wizard: Dialog 3 - Einstellungen	Seite 52
Abbildung 17 – Delta Wizard: Dialog 4 - Bestätigung	Seite 52
Abbildung 18 – Delta Wizard: Dialog 5 - Ergebnisse	Seite 53
Abbildung 19 – Text Delta Wizard - Diff Ansicht	Seite 54
Abbildung 20 – Text Delta Wizard - Edit Script Ansicht	Seite 55
Abbildung 21 – Text Delta Unit Tests der Gruppe „GetDiffTest3“	Seite 75
Abbildung 22 – Code Coverage MyersDiff	Seite 80
Abbildung 23 – Code Coverage TridgellDelta	Seite 81
Abbildung 24 – Performance Report Summary für die binär Delta Erzeugung	Seite 82
Abbildung 25 – Allocation Details für die binär Delta Erzeugung	Seite 82
Abbildung 26 – Performance Report Summary für die Text Delta Erzeugung	Seite 83
Abbildung 27 – Allocation Details für die Text Delta Erzeugung	Seite 83

8.3 Tabellen

Tabelle 1 – Verschiedene Delta Lösungen	Seite 3
Tabelle 2 – Verschiedene bekannte Hash Funktionen	Seite 9
Tabelle 3 – Der Adler-32 Algorithmus	Seite 11
Tabelle 4 – Traditionelle XP Praktiken	Seite 28
Tabelle 5 – .NET Namespaces und deren Verwendung	Seite 32
Tabelle 6 – Testdaten für die manuellen Tests des binären Delta Algorithmus	Seite 39
Tabelle 7 – Testläufe mit dem Binär Delta Wizard.	Seite 42
Tabelle 8 – Performance Ergebnisse mit dem Binärdelta Wizard	Seite 56
Tabelle 9 – Verbesserungs- und Erweiterungsmöglichkeiten	Seite 60
Tabelle 10 – Beispiele für D-Pfade	Seite 70
Tabelle 11 – Invertierung eines Edit Script	Seite 71

8.4 Glossar

Anonyme Methode – Eine Methode (Funktion) die ohne Methodennamen direkt an einen Delegaten übergeben werden kann.
Assembly – Zusammenstellung von Code-, Konfigurations-, Ressourcendateien und weiteren Dateien, die zu einer Anwendung oder einer Softwarebibliothek kompiliert werden können.
Attribut – Ein gesondert markierter Textabschnitt vor einer Klasse oder Methode, welcher Metadaten zu der nachfolgenden Codestruktur beinhalten und diesen beschreiben kann.
Balanciertheit – Die Ergebnisse einer gut balancierten Hash Funktion sind auch bei ähnlichen Eingangsdaten auf die Ausgangsmenge statistisch gleichmäßig verteilt.
CIL – Common Intermediate Language , eine Zwischensprache für alle .NET Sprachen, die von der Common Language Runtime interpretiert wird. Die CIL hieß früher Microsoft Intermediate Language (MSIL), wurde aber im Rahmen der Standardisierung durch die ECMA umbenannt.
CRC – Cyclic Redundancy Checksum . Eine Methode zur Berechnung von Checksummen.
Delegat – Ein Funktionspointer im objektorientierten Sinn. Der Delegat ist streng typisiert und kann auch Parameter übernehmen.
ECMA – European Computer Manufacturers Association , eine private Normungsorganisation zur Standardisierung von Informations- und Kommunikationssystemen und Unterhaltungselektronik mit Sitz in Genf.
Emit – Ein Mechanismus zur dynamischen Erzeugung von CIL Code. Gehört zum so genannten Reflection System von .NET.
Event – Ein Event (Deutsch: Ereignis) ist ein Mechanismus, der verwendet wird um Klassen über ausgewählte Zustandsänderungen von Objekten zu informieren.
Generics – Eine Code Entwicklungstechnik, die Typenabstraktion und Parametrierbarkeit ermöglicht.
IEC – Internationale elektrotechnische Kommission , Normierungsgremium für Elektronik und Elektrotechnik mit Sitz in Genf.
IEEE – Institute of Electrical and Electronics Engineers , ein weltweiter Berufsverband von Ingenieuren aus den Bereichen Elektrotechnik und Informatik.
ISO – Internationale Organisation für Normung , eine internationale Vereinigung von Normungsorganisationen.
JIT - Just in Time Compiler . Ein Compiler welcher den auszuführenden Code bei Bedarf dynamisch kompiliert. Ein JIT-Compiler ist in der Lage, den Befehlssatz des Prozessors, auf dem das Programm ausgeführt wird, optimal auszunutzen und erreicht durch weitere Techniken wie Caching und zusätzliche Optimierung von oft benutzten Code Abschnitten in vielen Fällen mit native code vergleichbare Ausführgeschwindigkeiten. In Einzelfällen werden Programme sogar schneller ausgeführt als eine native Version.
Konkatenation – Das Aneinanderhängen von zwei Zeichenketten.
Managed – Als Managed wird Code bezeichnet, welcher von einer virtuellen Maschine oder einem Just in Time Compiler ausgeführt und gemanagt wird.
Metadaten – Daten über Daten. In C# Code können Code Metadaten in Form von

Attributen erfasst werden, die dann über die Reflection ausgewertet werden können.
MSIL – Microsoft Intermediate Language. Umbenannt, siehe Common Intermediate Language
Native Code - wird Quellcode genannt, welcher in kompilierter Form ausgeführt wird, also nicht interpretiert oder mithilfe eines JIT Compilers übersetzt wird. Ein Nachteil von native code ist, dass die kompilierte Version in der Regel nur für einen einzigen Maschinentyp ausführbar ist.
O(n) – Rechen oder Speicherkomplexität wird in der IT mit dem O Symbol und einem Ausdruck der die Komplexität in Bezug auf den oder die zentralsten Kennzahlen des betreffenden Algorithmus angibt. O(n) bedeutet, dass der damit beschriebene Algorithmus für einen vollen Durchlauf n Einheiten Zeit oder Speicher benötigt.
Property – Eine methodenähnliche Codestruktur welche darauf spezialisiert ist für einen bestimmten Wert (Konstante oder Klassenvariable) oder ein Objekt Lese- und optional auch Schreibzugriff zur Verfügung zu stellen. In Java wird dies über Methoden mit vorangestelltem „Get“ oder „Set“ bewerkstelligt.
Reflection – Eine Datenstruktur in .NET Softwareprojekten, welche den aktuellen Code als Objektmodell widerspiegelt, so dass die ausführende Assembly beispielsweise ihren eigenen Code auf ungewollte Veränderungen überprüfen kann.
Refactoring – Eine Technik aus der Softwareentwicklung: Bestehender Code wird schrittweise modernisiert oder erneuert. Mit Hilfe von Unit Tests wird sichergestellt, dass dadurch keine ungewollten Verhaltensänderungen in der Software auftreten.
VCS – Version Control Systems sind Softwaresysteme zur Verwaltung von mehreren Revisionen von verschiedenen Informationseinheiten, wie zum Beispiel im SCM (Sourcecode Management) oder bei Dokumentenmanagement Systemen.
WEP – Wired Equivalent Privacy , ein veralteter Standard-Verschlüsselungsalgorithmus für Wireless LAN Verbindungen.

9. Lebenslauf

30. Oktober 1978: geboren in 6800 Feldkirch

1985-1997: Volksschule, Bundesgymnasium Gallusstraße in Bregenz

Juni 1994: privater Sprachaufenthalt USA

Juni / August 1996: Sprachaufenthalt Frankreich (Alpha B Institut Linguistique)

Mai 1997: Hauptpreis beim europaweiten Wettbewerb „Europa 1997 in der Schule“ mit dem Thema Computergrafik.

Juni 1997: Matura (Deutsch, Englisch, Französisch, Mathematik, Biologie und Chemie)

1997 / 1998: Präsenzdienst in der Dauer von 8 Monaten

WS 1998: Beginn eines Studiums der Medizin an der Universität Innsbruck

Januar 2002: Einstellung bei der Firma „holzweg e-commerce solutions“

Ab März 2002: Projektleitung bei mehreren großen Kundenprojekten

WS 2002: Wechsel zum Studium der medizinischen Informatik an der UMIT

30. November 2002: Beendigung des Arbeitsverhältnisses bei der Firma „holzweg“.

März 2003: Anmeldung des Gewerbes „Dienstleistungen in der automatischen Datenverarbeitung und Informationstechnik“

Sommer 2005: Bakkalaureat und anschließend Beginn des Master Studiengangs

Sommer 2006: Zusammenlegung des Unternehmens mit Bernhard Glück und Umbenennung zu BlueLogic Software Solutions

Hiermit erkläre ich an Eides statt, die Arbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet zu haben.

Matthias Gruber

Innsbruck, am 11.06.2007

10. Anhänge

10.1 Anhang 1 – Text Delta

Zusätzliche Informationen zum vorgestellten Algorithmus nach Myers.

10.1.1 Der Algorithmus

Der Algorithmus von Myers basiert auf dem Greedy Algorithmus, welcher ebenfalls als Pseudo Code in (Myers 1986) angegeben wird. Dazu wird das Edit Script als Baum betrachtet, welcher dann durchsucht wird. Ein D-Pfad wird definiert als ein Weg durch das Edit Script, welcher aus D Einfüge- oder Löschoperationen besteht und beliebig viele diagonale Kanten enthalten kann. Bei zwei völlig disjunkten Texten der Längen M und N gilt $D = M + N$. Dies kann als Terminierungskriterium für den Algorithmus benutzt werden.

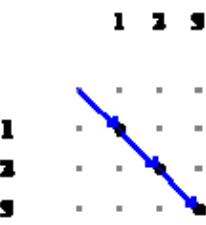
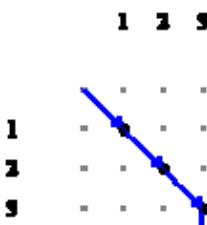
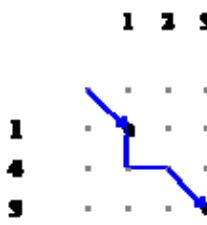
0-Pfad	1-Pfad	2-Pfad
		

Tabelle 10 – Beispiele für verschiedene D-Pfade. D kann maximal so groß wie $M + N$ werden, wenn die beiden Texte keinerlei Übereinstimmung aufweisen.

Die Diagonalen im Edit Script werden so durchnummeriert, dass der Index k der Diagonalen gleich $x - y$ ist. Bei geraden D muss ein D-Pfad dann stets auf einer geraden Diagonale enden, die Umkehrung gilt ebenfalls. Außerdem kann ein D-Pfad nur auf Diagonalen mit Nummern von $-D$ bis D enden.

Der Algorithmus nach Myers sucht nun von $(0, 0)$ und von (M, N) zugleich ausgehend mit $D = 0$ beginnend nach Pfaden auf den Diagonalen $k = -D$ bis $k = D$ und hält die letzte diagonale Wegstrecke sowie die Endpunkte der längsten Pfade pro Diagonale im Speicher. Sobald zwei überlappende Pfade gefunden wurden, wird die Länge des

kürzesten Edit Script und die zuletzt gefundene diagonale Strecke – die so genannte „middle snake“ – zurückgegeben. Damit kann dann sehr einfach durch eine „teile und herrsche“ Strategie rekursiv das gesamte kürzeste Edit Script gefunden werden.

10.1.2 Invertierung

Die Invertierung eines Edit Script wird erreicht, indem die Bedeutung von Einfüge- und Löschoperationen umgekehrt wird. Dies wird im Folgenden anhand eines einfachen Beispiels demonstriert:

$A, B, C, D, E, F, G \rightarrow A, X, Y, B, C, D, Z, F$	$A, X, Y, B, C, D, Z, F \rightarrow A, B, C, D, E, F, G$
$\{X=0, Y=0\} \{X=1, Y=3\} \{X=2, Y=4\}$ $\{X=3, Y=5\} \{X=5, Y=7\}$	$\{X=0, Y=0\} \{X=3, Y=1\} \{X=4, Y=2\}$ $\{X=5, Y=3\} \{X=7, Y=5\}$

Tabelle 11 – Invertierung eines Edit Script. Einfügeoperationen werden zu Löschoperationen und umgekehrt. Auch die Koordinaten der überquerten Match Points (Trace) vertauschen sich durch die Invertierung.

10.2 Anhang 2 – Binär Delta

Zusätzliche Details und Beispiele zum binären Delta Algorithmus nach Tridgell.

10.2.1 Der Algorithmus

Wie bereits einleitend erklärt wird, benötigt der rsync Algorithmus nach (Tridgell 1999) von der Quelldatei eine Sammlung von Hash Werten, die im SDelta Library als BlockedHashset bezeichnet wird. Dabei handelt es sich um Hash Paare bestehend aus je einem MD4 und einem Adler32 Wert, welche pro Block angelegt werden. Der eigentliche Delta wird dann anhand der Zielfile generiert, indem für jedes Byte Offset in der Datei überprüft wird, ob der an dieser Stelle beginnende Block in der Quelldatei vorhanden ist. Dazu werden im BlockedHashset zu jeder Adler32 Checksumme eine oder mehrere MD4 Werte gespeichert, über welche die Adresse des übereinstimmenden Blocks relativ zur Quelldatei bezogen werden kann. Wenn kein übereinstimmender Block gefunden wurde wird das erste Byte als eingefügt interpretiert und an der nächsten Adresse weiter gesucht.

Um nicht für jedes eingefügte Byte und für jeden kopierten Block einen eigenen Befehl erstellen zu müssen, werden die Kommandos jeweils so lange akkumuliert, bis das jeweils andere Kommando auftritt oder der Vorgang beendet ist. Dieser Teil der Verarbeitung ist nicht in (Tridgell 1999) beschrieben.

10.2.2 Einfluss der Blockgröße

Da die Quelldatei aus Gründen der Performance und Praktikabilität nur in Blöcken verglichen werden kann, hat der entsprechende Parameter für die Blockgröße einen besonderen Einfluss auf die Arbeitsweise des Algorithmus. Dies kann mit einem einfachen Beispiel verdeutlicht werden: Bei einer Blockgröße von fünf Byte soll folgendes Stringpaar verarbeitet werden:

Quelldaten:

ABCDE FGHIJ KLMNO

Zieldaten:

ABCD1 EFGH2 IJKL3

Es ist leicht zu sehen, dass dies nur zu einer Reihe von Einfügebefehlen führt, da kein einziger übereinstimmender Block gefunden werden kann. Bei einer Blockgröße von 4 Byte würden dagegen 3 Kopier- und 3 Einfügeoperationen von je einem Block beziehungsweise einem Byte entstehen, was natürlich einer höheren Kompressionsrate entspricht. Somit steigt die Kompressionsrate oft bei Verkleinerung der Blockgröße, allerdings steigt dann auch das Risiko für ein Fehlschlagen der Operation durch MD4 Kollisionen.

10.2.3 Beispiel

Ein weiteres Beispiel für die binäre Delta Berechnung, welches aus den Unit Tests für den Algorithmus kommt. Als Blockgröße für den Test wurde hier fünf Bytes gewählt, so dass der Vorgang für Menschen überschaubar bleibt. Der Test stellt unter anderem sicher, dass der Zähler für Wiederholungen bei Kopierbefehlen korrekt auf Null gesetzt wird, wenn im Anschluss an eine Kopie mit mehreren Wiederholungen ein Einfügebefehl folgt.

Quelldatenstring in Fünfergruppen:

```
01234 56789 abcde fghij kAAAAA AAAAAA AAAAAA BBBBBB BBBBBB BBBBBB BBBCC CCCCCC CCCCCC  
CCCCC 01234 56789 abcde fghij k
```

Zieldatenstring in Fünfergruppen:

```
5678x cv9zz zzzzz zzAAA AAAAAA AAöüä öüööü äöüäö üäüäö AAAAAA AAAAAA AAAAAA AAAAAA  
AABCC CCCCCC CCCCCC 01234 tt567 89abc defgh ijk
```

Der zugehörige Delta sieht in XML Form folgendermaßen aus:

```
<?xml version="1.0" encoding="utf-8"?>  
<Delta digestname="SHA256" hashname="MD4" checksumname="Adler32" blocksize="5">  
  <commands>  
    <insert>  
      <base64Binary>NTY30Hhjdj16enp6enp6eno=</base64Binary>  
    </insert>  
    <copy>  
      <from>5</from>  
      <to>5</to>  
      <repeat>1</repeat>  
    </copy>  
    <insert>  
      <base64Binary>w7bDvM0kw7bDvM0kw7bDvM0kw7bDvM0kw7bDvM0kw7zDpMO2  
    </base64Binary>
```

```

</insert>
<copy>
<from>5</from>
<to>5</to>
<repeat>3</repeat>
</copy>
<insert>
<base64Binary>QUFC</base64Binary>
</insert>
<copy>
<from>11</from>
<to>11</to>
<repeat>2</repeat>
</copy>
<insert>
<base64Binary>Q0M=</base64Binary>
</insert>
<copy>
<from>0</from>
<to>0</to>
<repeat>0</repeat>
</copy>
<insert>
<base64Binary>dHQ=</base64Binary>
</insert>
<copy>
<from>1</from>
<to>3</to>
<repeat>0</repeat>
</copy>
<insert>
<base64Binary>aw==</base64Binary>
</insert>
</commands>
<digest>AZj3s52diP+tf9IZyxb3NUM2mGx4XLchc6hYAxjdN7lCbHVMb2dpYw==</digest>
</Delta>

```

10.3 Anhang 3 – Weitere Testergebnisse

10.3.1 Weitere Unit Tests

Die folgenden Abbildungen 21.1 bis 21.10 geben die Text Delta Unit Tests der Gruppe „GetDiffTest3“ wieder. In der linken Spalte finden sich jeweils die beiden Strings, die verglichen werden. Auf der rechten Seite sind die Spalten nach den Indices des ersten und die Zeilen nach den Indices des zweiten Strings beschriftet.

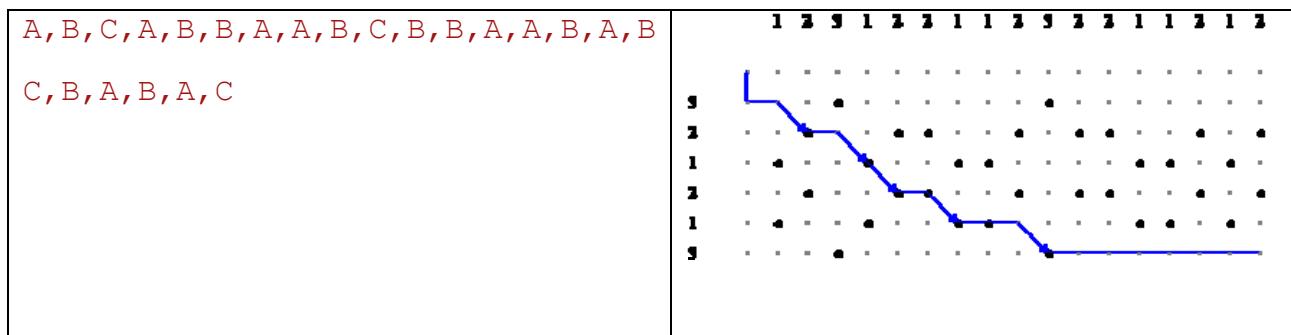


Abbildung 21.1

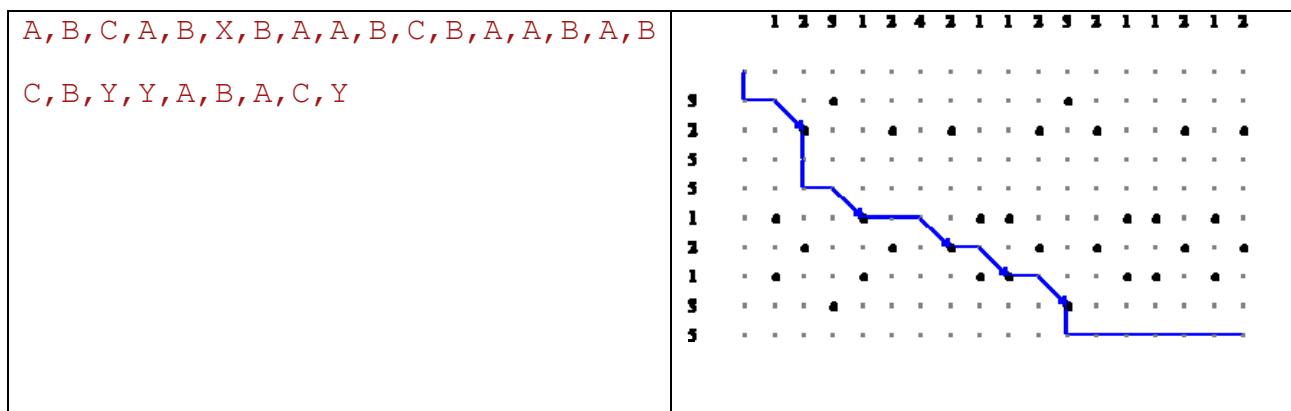


Abbildung 21.2

A, B, C, A, B, X, B, A, A, B, C, B, A, A, B, A, B
A, B, C, A, B, B, A, A, B, C, B, B, A, A, B, A, B

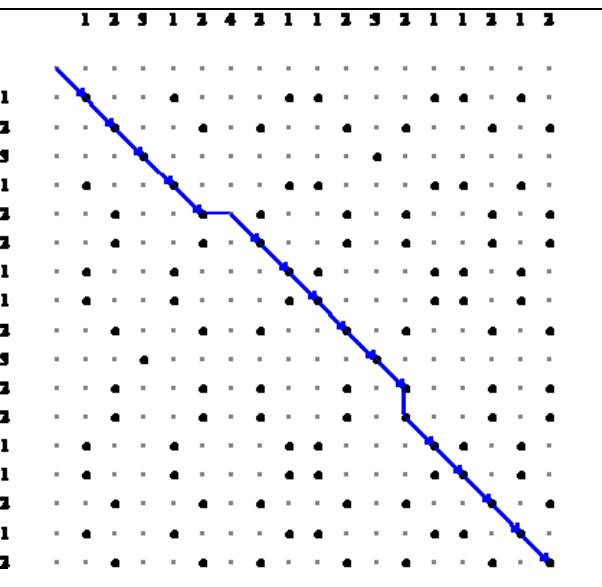


Abbildung 21.3

A, B, C, D, E, F, G, H, I, J, K
1, 2, 3, 4, 5, 6, 7, 8, 9, 0

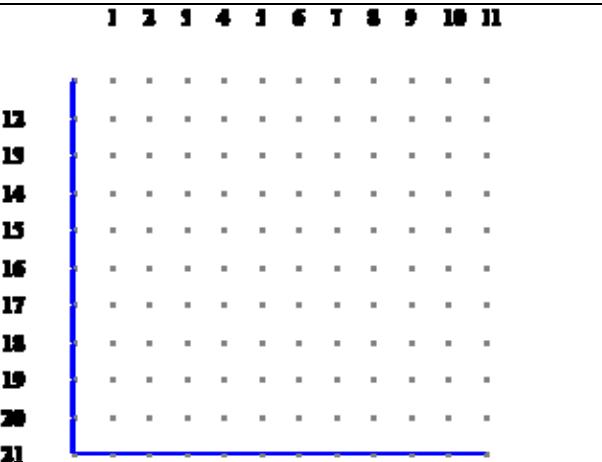


Abbildung 21.4

A, B, C, D, E, F, G, H, I, J, K

A, B, C, D, E, F, G, H, I, J, K

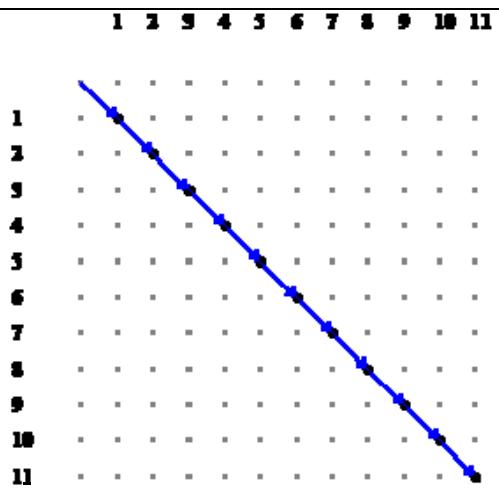


Abbildung 21.5

A, B

, A, , , B,

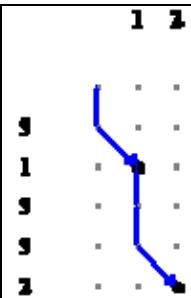


Abbildung 21.6

F

0, F, 1, 2, 3, 4, 5, 6, 7

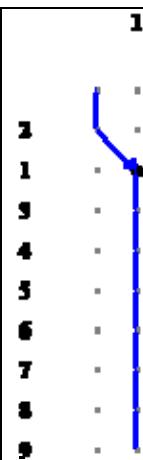


Abbildung 21.7

A, B, Y, C, D, E, F, F

A, B, X, C, E, F

1 2 3 4 5 6 7 7

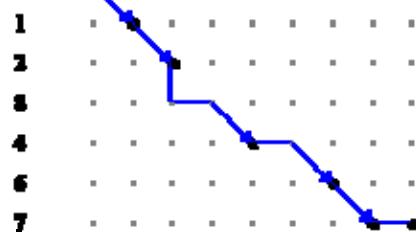


Abbildung 21.8

A, B, C, D, E, F, G, H, I, J, K

B, C, D, E, F, G, H, I, J, K, L

1 2 3 4 5 6 7 8 9 10 11

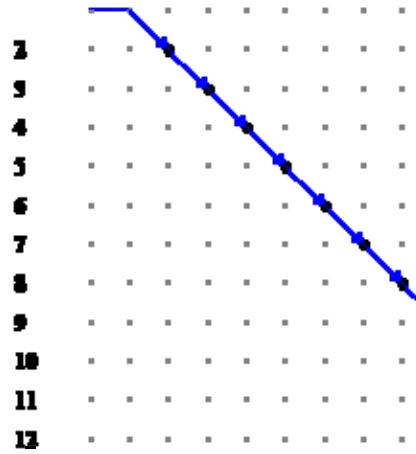


Abbildung 21.9

1, A, 2, B, C, D, E, G, H, I, J, 3, K, L
4, A, 5, B, C, D, E, 6, E, G, H, I, J, 7, K, 8, L

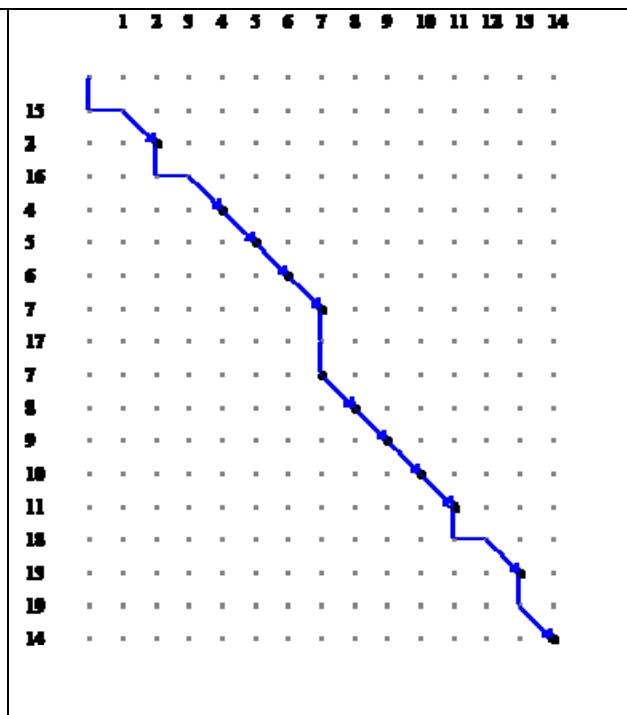


Abbildung 21.10

10.3.2 Code Coverage und Profiling

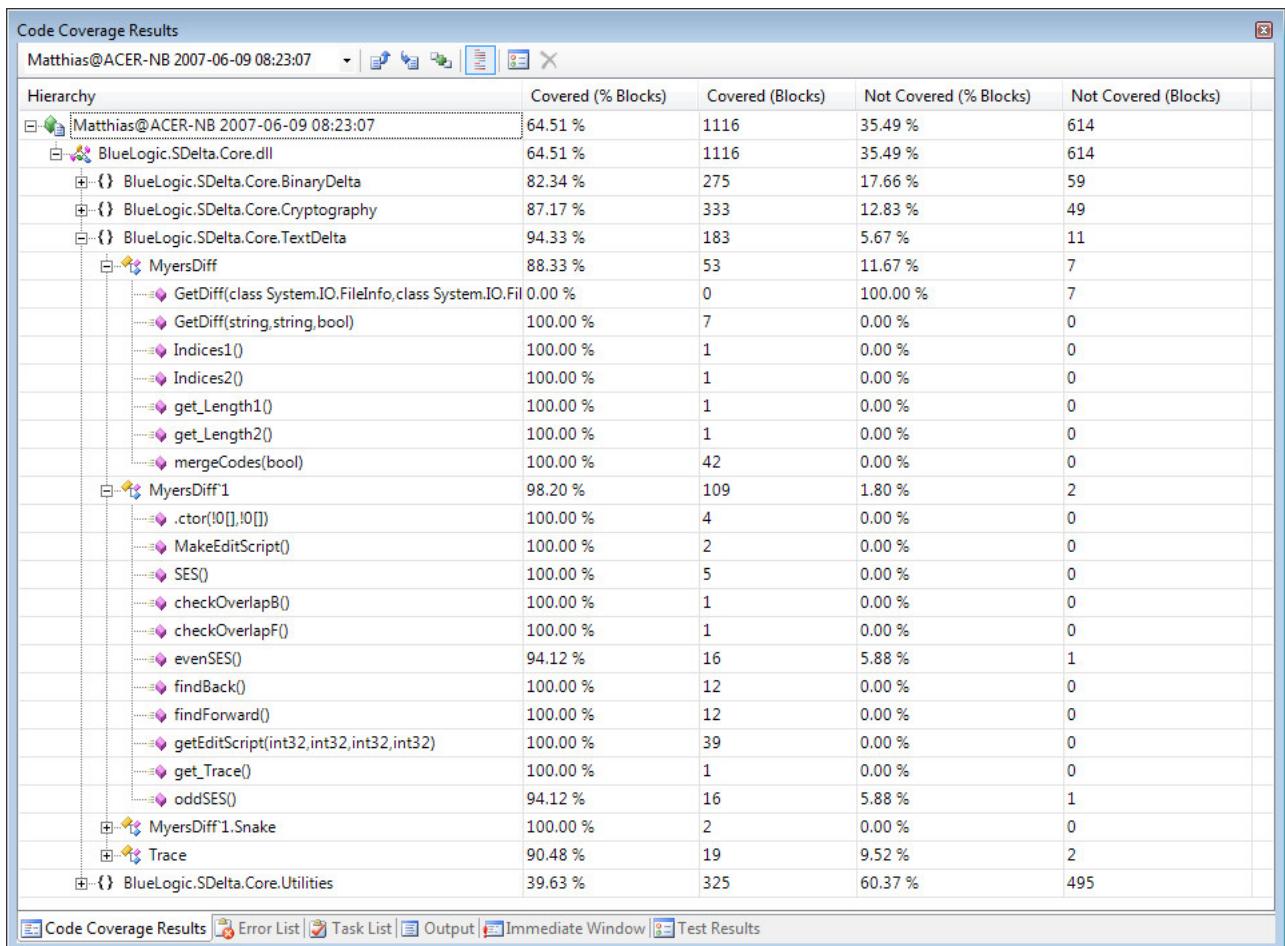


Abbildung 22 – Code Abdeckung durch Unit Tests in der MyersDiff Klasse

Hierarchy	Covered (% Blocks)	Covered (Blocks)	Not Covered (% Blocks)	Not Covered (Blocks)
Matthias@ACER-NB 2007-06-09 08:23:07	64.51 %	1116	35.49 %	614
BlueLogic.SDelta.Core.dll	64.51 %	1116	35.49 %	614
{} BlueLogic.SDelta.Core.BinaryDelta	82.34 %	275	17.66 %	59
{} BinaryCommand	76.74 %	33	23.26 %	10
{} BlockedHashSet2	96.08 %	49	3.92 %	2
{} Delta1	100.00 %	16	0.00 %	0
{} TridgellDelta	100.00 %	17	0.00 %	0
{} .ctor()	100.00 %	3	0.00 %	0
{} .ctor(int32)	100.00 %	3	0.00 %	0
{} .ctor(int32, class BlueLogic.SDelta.Core.IDigestPro	100.00 %	5	0.00 %	0
{} ApplyDelta(class System.IO.Stream, class System.I	100.00 %	2	0.00 %	0
{} GetBlockSignatures(class System.IO.Stream, class S	100.00 %	2	0.00 %	0
{} GetDelta(class System.IO.Stream, class BlueLogic.	100.00 %	2	0.00 %	0
{} TridgellDelta2	77.29 %	160	22.71 %	47
{} .ctor()	0.00 %	0	100.00 %	13
{} .ctor(int32)	0.00 %	0	100.00 %	13
{} .ctor(int32, class BlueLogic.SDelta.Core.IDigestPro	100.00 %	4	0.00 %	0
{} .ctor(int32, class BlueLogic.SDelta.Core.IHashProv	100.00 %	5	0.00 %	0
{} ApplyDelta(class BlueLogic.SDelta.Core.BinaryDel	100.00 %	5	0.00 %	0
{} ApplyDelta(class System.Collections.Generic.List`	0.00 %	0	100.00 %	2
{} ApplyDelta(class System.Collections.Generic.List`	100.00 %	34	0.00 %	0
{} ApplyDelta(class System.IO.Stream, class System.I	100.00 %	6	0.00 %	0
{} GetBlockSignatures(class System.IO.Stream)	66.67 %	14	33.33 %	7
{} GetBlockSignatures(class System.IO.Stream, class S	83.33 %	5	16.67 %	1
{} GetDelta(class System.IO.Stream, class BlueLogic.	100.00 %	29	0.00 %	0
{} GetDelta(class System.IO.Stream, class BlueLogic.	100.00 %	10	0.00 %	0
{} emitCopy(int64)	100.00 %	25	0.00 %	0
{} emitInsert(uint8)	62.50 %	5	37.50 %	3
{} emitInsert(uint8[])	50.00 %	6	50.00 %	6
{} flushLastCommand()	75.00 %	6	25.00 %	2
{} writeCopy(int64,int64,int32)	100.00 %	3	0.00 %	0
{} writeInsert(uint8[])	100.00 %	3	0.00 %	0
{} BlueLogic.SDelta.Core.Cryptography	87.17 %	333	12.83 %	49
{} BlueLogic.SDelta.Core.TextDelta	94.33 %	183	5.67 %	11
{} BlueLogic.SDelta.Core.Utilities	39.63 %	325	60.37 %	495

Code Coverage Results | Error List | Task List | Output | Immediate Window | Test Results

Abbildung 23 – Code Abdeckung durch Unit Tests in der TridgellDelta Klasse

Performance Report Summary

Functions Allocating Most Memory

Name	Bytes	%
BlueLogic.SDelta.Core.BinaryDelta.BlockedHashset`2.Check	149257104	68.275
System.Xml.Serialization.XmlSerializationReader.ReadByteArray	23455216	10.729
System.String.GetStringForStringBuilder	13042464	5.966

Types With Most Memory Allocated

Name	Bytes	%
System.UInt64	149647344	68.453
System.Byte[]	40283935	18.427
System.String	17874730	8.176

Types With Most Instances

Name	Instances
System.UInt64	9352959
System.String	145647
System.Byte[]	67765

Abbildung 24 – Performance Report Summary bei einem Testlauf für binäre Delta Erzeugung.

Function Name	Exclusive Allocati...	Inclusive Allocati...	Exclusive Bytes A...	Inclusive Bytes A...
BlueLogic.SDelta.Core.dll	9543275	9706540	168779009	206995245
... BlueLogic.SDelta.Core.BinaryDelta.BlockedHashset`2.Check(uint8[],uint8[],class BlueLogic.SDelta.C...	9328569	9427629	149257104	155611494
... BlueLogic.SDelta.Core.Cryptography.MD4..ctor()	59685	59685	3581100	3581100
... BlueLogic.SDelta.Core.Cryptography.MD4.engineDigest()	39630	39630	2058944	2058944
... BlueLogic.SDelta.Core.Utilities.XmlSerializationProvider`2.WriteBlockedHashSet(class BlueLogic.SD...	36618	49463	1043912	1629248
... BlueLogic.SDelta.Core.BinaryDelta.BlockedHashset`2.AddPair(I0,I1)	24390	49305	487744	2097692
... BlueLogic.SDelta.Core.Cryptography.MD4.GetHash(uint8[],uint8[])	19815	257595	554820	16523894
... BlueLogic.SDelta.Core.Cryptography.MD4.BytesToHex(uint8[],int32)	19815	138705	396300	10343430
... BlueLogic.SDelta.Core.Utilities.Hashtable`2.Add(I0,I1)	12188	12310	195008	637952
... BlueLogic.SDelta.Core.BinaryDelta.TridgellDelta`2.emitInsert(uint8)	904	943	9267808	9341076
... BlueLogic.SDelta.Core.Utilities.XmlSerializationProvider`2..ctor()	369	1939	19630	88444
... BlueLogic.SDelta.Core.Cryptography.NetHashes.ComputeHash(class System.IO.Stream,string,uint8	240	967	10880	395870
... BlueLogic.SDelta.Core.Utilities SlidingStreamWindow..ctor(class System.IO.Stream,int32,int32)	160	160	739200	739200
... BlueLogic.SDelta.Core.Utilities.XmlSerializationProvider`2.WriteDelta(class BlueLogic.SDelta.Core.Bi...	124	3491	2864	699842
... BlueLogic.SDelta.Core.BinaryDelta.TridgellDelta`2.GetBlockSignatures(class System.IO.Stream)	121	208126	112562	12553750
... BlueLogic.SDelta.Core.BinaryDelta.TridgellDelta`2.GetDelta(class System.IO.Stream,class BlueLogic...	120	9428946	3840	165833247
... BlueLogic.SDelta.Core.BinaryDelta.TridgellDelta`2.GetDelta(class System.IO.Stream,class BlueLogic...	90	9433445	2694	166750763
... BlueLogic.SDelta.Core.BinaryDelta.TridgellDelta`2..ctor(int32, class BlueLogic.SDelta.Core.IDigestPr...	80	240	820160	824000
... BlueLogic.SDelta.Core.BinaryDelta.BlockedHashset`2..ctor(int32,string,string)	80	124	3840	9660
... BlueLogic.SDelta.Core.BinaryDelta.TridgellDelta`2.ApplyDelta(class System.IO.Stream,class System...	80	13395	2400	24967732
... BlueLogic.SDelta.Core.BinaryDelta.TridgellDelta`2.GetBlockSignatures(class System.IO.Stream,class ...	49	258739	1434	14240580
... BlueLogic.SDelta.Core.BinaryDelta.TridgellDelta`2.ApplyDelta(class System.Collections.Generic.List`...	40	95	82400	308340
... BlueLogic.SDelta.Core.BinaryDelta.Delta`1..ctor()	40	40	960	960
... BlueLogic.SDelta.Core.Utilities SlidingStreamWindow.GetNextWindow()	40	40	46057	46057
... BlueLogic.SDelta.Core.BinaryDelta.TridgellDelta`2.flushLastCommand()	20	46	60480	64152
... BlueLogic.SDelta.Core.Cryptography.SHA256.GetHash(class System.IO.Stream,uint8[])	4	971	124	395994
... BlueLogic.SDelta.Core.BinaryDelta.TridgellDelta`2.emitInsert(uint8[])	3	4	26696	26932
... BlueLogic.SDelta.Core.Utilities.XmlSerializationProvider`2.ReadDelta(class BlueLogic.SDelta.Core.Bi...	1	12768	48	24640336
... BlueLogic.SDelta.Core.BinaryDelta.TridgellDelta`2.emitCopy(int64)	0	4	0	496
... BlueLogic.SDelta.Core.Utilities SlidingStreamWindow.<GetEnumerator>d__0.MoveNext()	0	40	0	46057
... BlueLogic.SDelta.Core.BinaryDelta.TridgellDelta`2.writeInsert(uint8[])	0	50	0	75192
... BlueLogic.SDelta.Core.Utilities.Hashtable`2..ctor()	0	1	0	144
... BlueLogic.SDelta.Core.BinaryDelta.TridgellDelta`2.writeCopy(int64,int64,int32)	0	20	0	2480

Abbildung 25 – Allocation Details für die binär Delta Erzeugung.

Performance Report Summary

Top Inclusive Sampled Functions

Name	Samples	%
[mscorwks.dll]	127	99.219
[mscoree.dll]	125	97.656
BlueLogic.SDelta.DiffWizard.Program.Main	122	95.313

Top Exclusive Sampled Functions

Name	Samples	%
[gdiplus.dll]	63	49.219
[mscorwks.dll]	27	21.094
[RichEd20.DLL]	11	8.594

Abbildung 26 – Performance Report Summary bei einem Testlauf für Text Delta Erzeugung.

Function Name	Inclusive Percent	Exclusive Percent	Inclusive Samples	Exclusive Samples
+ LPK.DLL	1.563	0.000	2	0
+ comctl32.dll	2.344	0.000	3	0
+ BlueLogic.SDelta.DiffWizard.exe	95.313	0.000	122	0
+ mscoree.dll	97.656	0.000	125	0
+ ole32.dll	3.125	0.000	4	0
BlueLogic.SDelta.Core.dll	50.000	0.000	64	0
... BlueLogic.SDelta.Core.TextDelta.Trace.ToString()	0.781	0.000	1	0
... BlueLogic.SDelta.Core.TextDelta.MyersDiff1.SES()	0.781	0.000	1	0
... BlueLogic.SDelta.Core.TextDelta.MyersDiff1.evenSES()	0.781	0.000	1	0
... BlueLogic.SDelta.Core.TextDelta.MyersDiff1.getEditScript(int32,int32,int32)	1.563	0.000	2	0
... BlueLogic.SDelta.Core.TextDelta.MyersDiff.GetDiff(class System.IO.FileInfo,class System.IO.FileInfo,	1.563	0.000	2	0
... BlueLogic.SDelta.Core.Utilities.ScriptDrawer.DrawHeader()	3.125	0.000	4	0
... BlueLogic.SDelta.Core.Utilities.FormsDraw.Write(string,string,int32,valueuetype System.Drawing.Colo	3.125	0.000	4	0
... BlueLogic.SDelta.Core.Utilities.ScriptDrawer.DrawGrid()	43.750	0.000	56	0
... BlueLogic.SDelta.Core.Utilities.FormsDraw.Pset(valueuetype System.Drawing.Color,float32,float32,flo	43.750	0.000	56	0
... BlueLogic.SDelta.Core.Utilities.ScriptDrawer..ctor(int32[],int32[]])	47.656	0.000	61	0

Abbildung 27 – Allocation Details für die Text Delta Erzeugung.