Eidgenössische
Technische Hochschule
Zürich

Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Swiss Federal Institute of Technology Zurich

Department of
Computer Science

Institute of
Theoretical Computer Science

Prof. Dr. P. Widmayer

Strukturelle Eigenschaften von Strings

Diplomarbeit von
Daniel Emmenegger

Departement Informatik
5. November 2001 – 4. März 2002

Daniel Emmenegger
Stutzrain 19
6005 St. Niklausen
daniel.emmenegger@gmx.net

Assistenten:

Zsuzsanna Lipták
Mark Cieliebak

Eidgenössische Technische Hochschule Zürich
Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Swiss Federal Institute of Technology Zurich

Institut für Theoretische Informatik
Prof. Dr. P. Widmayer
Mark Cieliebak
Zsuzsanna Lipták

DIPLOMARBEIT / SEMESTERARBEIT:

Strukturelle Eigenschaften von Strings

Strings und ihre Eigenschaften werden in vielen Gebieten der Informatik verwendet, z.B. in der Kryptographie, beim Entwurf von Programmiersprachen oder in der Bioinformatik: ein Protein kann man vereinfacht als Folge von Buchstaben auffassen. Entsprechend vielfältig und umfangreich sind das Wissen und die Literatur zum Thema Strings. Manchmal interessieren jedoch nur die Eigenschaften eines bestimmten Strings. Dann helfen die allgemeinen Aussagen aus der Literatur meist nicht weiter, und man muss den konkreten String selbst untersuchen.

Ziel dieser Arbeit ist es, ein Java-Tool zu entwickeln, mit dem einzelne Strings oder Klassen von Strings analysiert werden können. Dies beinhaltet im einzelnen

- das Erzeugen von Strings (zufällig oder regelbasiert),
- die Analyse der Eigenschaften dieser Strings,
- sowie die geeignete Visualisierung der Ergebnisse.

„Eigenschaften“ können beispielsweise die Anzahl verschiedener Teilstrings oder die Häufigkeitsverteilung der einzelnen Buchstaben sein. Aber auch komplexere Eigenschaften wie beispielsweise verschiedene Masse für die Vorsortiertheit eines Strings sollen berechnet werden können. Da im voraus nicht klar ist, welche Tests später interessant sein könnten, muss das Tool so konzipiert und implementiert werden, dass es leicht um zusätzliche Tests erweitert werden kann.

Vorkenntnisse Java

Weitere Infos bei:

Mark Cieliebak	Raum: CLW B2	Tel: (01) 632 55 07	email: cielieba@inf.ethz.ch
Zsuzsanna Lipták	Raum CLW BA	Tel: (01) 632 74 05	email: zsuzsa@inf.ethz.ch

Inhaltsverzeichnis

1 Einleitung 1
1.1 Ziel der Diplomarbeit / Aufgabenstellung 1
1.2 Übersicht 2
1.3 Verwendung dieser Diplomarbeit 3
2 Vorangegangene Arbeiten 5
3 Motivation 6
4 Konzepte 8
4.1 Überblick 8
4.2 Eingabe 9
4.2.1 Eingabe per Hand 11
4.2.2 Zufallsgeneratoren 12
4.2.3 Reguläre Ausdrücke 13
4.2.4 Regelbasierte Stringerzeugung 16
4.2.5 Laden aus persistenten Quellen 17
4.3 Test-Box 18
4.4 Ausgabe 21
4.5 Zusammenfassung 22
5 Implementation 24
5.1 Allgemeines 24
5.1.1 „Coding Conventions“ 24
5.1.2 „Packages“ – Ordnung muss sein 24
5.2 Der Kern des Frameworks 25
5.2.1 Strings und Alphabete 25
5.2.2 Datenaustausch 27
5.2.3 Streams 29
5.2.4 Stringerzeugung nach festen Regeln 31
5.3 Die PLUG-IN Schnittstelle 31
5.3.1 Deskriptoren 32
5.3.2 Sicherheit durch Exceptions 38
5.4 Das Schreiben von Tests 39
5.4.1 Das Grundgerüst 39
5.4.2 Der Algorithmus 41
5.4.3 Visualisierung der Ergebnisse 44
5.5 User Interface 48
5.5.1 AWT oder Swing? 49
5.5.2 Benutzerschnittstelle auf Kommandozeilenebene 49

5.6 Sonstige Bestandteile des Frameworks ... 49
5.7 Zusammenfassung ... 51
6 STRATO Tutorial (Version 1.0) ... 52
6.1 Überblick – erste Schritte ... 52
6.2 Aufbau der graphischen Oberfläche ... 53
6.2.1 Eingabebereich ... 54
6.2.2 Tests ... 57
6.2.3 Der Ausgabebereich ... 59
6.2.4 Manipulation von Eingabe- und Ausgabedaten ... 60
6.3 Diverses ... 61
7 Gewichtete Strings ... 63
7.1 Suffix-Trees ... 63
7.2 Teilstrings ... 64
7.3 Multiplizitätsvektoren ... 67
7.4 Teilmassen ... 68
7.5 Implementation ... 68
8 Weiterführende Themen ... 69
8.1 Weitere Testarten ... 69
8.2 Hilfstool für das Schreiben von Tests ... 69
8.3 Mehrere gleichzeitige Inputs ... 69
9 Persönliche Bemerkungen ... 71
9.1 ...zur Programmieraufgabe ... 71
9.2 ...zum Ablauf der Diplomarbeit ... 72
9.3 ...zu meinen Erwartungen an die Benutzer des Tools ... 72
9.4 ...zu diversen Themen (Extreme Programming, Assistenten, ...) ... 73
9.5 Schlussbemerkungen ... 74
10 Anhang A ... 75
10.1 Referenz: Conventions ... 75
10.2 Verzeichnisse ... 76
10.2.1 Abbildungsverzeichnis ... 76
10.2.2 Tabellenverzeichnis ... 77
10.2.3 Beispielverzeichnis ... 77
10.2.4 Verzeichnis der Definitionen ... 77
10.2.5 Verzeichnis der Listings ... 77
10.2.6 Literaturverzeichnis ... 78
10.3 Danksagung ... 78
11 Anhang B ... 79

1 Einleitung

In der vorliegenden Arbeit befassen wir uns eingehend mit Strings und insbesondere deren Eigenschaften. Strings sind Zeichenfolgen, die über einem konkreten (meist endlichen) Alphabet definiert sind. Der Inhalt eines Briefes stellt beispielsweise eine solche Zeichenfolge dar. Aber auch nur eine einzige Zeile aus einem Roman oder ein Namenseintrag aus einer Personendatenbank sind Beispiele für Strings aus dem alltäglichen Leben. Im Gegensatz zu diesen auf dem lateinischen Alphabet definierten Strings, gibt es auch speziellere Varianten wie Bitstrings (0-1 Folgen über binärem Alphabet) oder DNA-Ketten (Sequenzen, die aus den vier Basen A, C, G, T[1] bestehen). Ob in der Kryptographie, der Bioinformatik oder sonst wo; Strings sind an den verschiedensten Orten im Einsatz und daher ein gut erforschtes Gebiet. In der Literatur finden sich unzählige Algorithmen, welche die Eigenschaften von Strings ermitteln. „Kommt ein String M (Muster) in einem andern String T (Text) vor?“ ist dabei eine mögliche Fragestellung aus der Stringtheorie. Die zur Beantwortung dieser Aufgabe bekannten Algorithmen leisten heutzutage in beinahe jedem Textverarbeitungssystem ihre Dienste, beispielsweise für die Volltextsuche.

Völlig unabhängig von der Art der verwendeten Strings werden immer neue Eigenschaften entdeckt. Doch selten steht dem Forscher[2] dabei ein Werkzeug zur Verfügung, mit dem er den selbst entwickelten Algorithmus (zur Berechung einer Eigenschaft) ausgiebig testen kann. Jedes Mal müssen viele Zeilen Programmcode verfasst werden, um anhand unterschiedlichster Arten von Eingaben die Korrektheit des Verfahrens zu überprüfen. Gibt es keine automatisierte Lösung, bleibt sogar nur der mühsame Weg, alles von Hand zu erledigen. Doch mit dieser Variante stösst man oft schon bei kurzen Stringlängen von wenigen Zeichen an die Grenzen der Machbarkeit.

1.1 Ziel der Diplomarbeit / Aufgabenstellung

Ziel dieser Arbeit ist es, ein Java-Tool (Framework) zu entwickeln, mit dem einzelne, konkrete Strings analysiert werden können. Das Tool soll dem Forscher erlauben, seine neuentwickelten Algorithmen in einer einfachen Art und Weise einzubinden und dabei auf eine breite Palette an Funktionen zurückgreifen zu können.

Dazu soll das Tool **erstens** eine Palette von Stringerzeugungsmechanismen bieten. Nebst der manuellen Eingabe von Hand, wird auch die halb- und vollautomatische Generierung von Strings ins Auge gefasst. Wir verstehen darunter unterschiedliche Techniken. Ein String kann beispielsweise zufällig erzeugt werden. Ebenso ist die Auswertung eines (regulären) Ausdrucks der Form *(ab)a^(2+3)(bc)^3* denkbar. Der zugehörige String wäre in diesen Fall (ausgeschrieben) *abaaaaabcbcbc*. Schliesslich kann ein String auch nach gewissen Regeln erzeugt werden. Beispiele für mögliche Anforderungen an den erzeugten String, zeigt die folgende Liste willkürlich ausgewählter Regeln

- Alle Buchstaben sollen etwa gleich oft vorkommen.
- Ein gegebener String τ soll mindestens c-mal vorkommen.
- Ein gegebener String τ soll genau c-mal vorkommen.
- Der erzeuge String soll ein *Palindrom*[3] darstellen.

[1] Die Buchstaben sind Abkürzungen für Adenin, Cytosin, Guanin, Thymin.
[2] Falls die weibliche Form (in diesem Fall wäre es Forscherin) auch nicht immer explizit aufgeführt wird, sind stets beide Geschlechter gemeint.
[3] Dieser und andere kursiv geschriebene Ausdrücke werden später noch definiert!

- Die Entfernung zwischen zwei gleichen Buchstaben soll immer mindestens c betragen.
- An jeder c-ten Stelle soll ein 'a' stehen.
- ...

Das Hauptaugenmerk (nebst der Stringerzeugung) gilt **zweitens** dem Mechanismus für die Analysefunktionen von Stringeigenschaften. Wir bezeichnen solche Analysefunktionen im weiteren als Tests. Wie oben angedeutet, wird die eigentliche Funktionalität des Tools durch den Forscher selbst beigesteuert, indem er seine Algorithmen als Test implementiert und dem Tool beifügt. Dabei wollen wir dem Programmierer keine Steine in den Weg legen, sich an etliche Konventionen halten zu müssen, um den meist schon geschriebenen Algorithmus auch einbinden zu können. Eine klar definierte Schnittstelle, die auch optionale Komponenten enthält, wird das Zusammenspiel von System und Testmodul regeln. Damit sich der Leser wiederum ein besseres Bild machen kann, was wir unter Eigenschaften von Strings verstehen, listen wir erneut ein paar Beispiele auf. Der Eingabestring, für den wir die Eigenschaft extrahieren wollen, heisst ab jetzt σ.

- Kommt ein gegebenes Pattern τ in σ vor? (ja/nein)
- Wie häufig kommt ein gegebenes Pattern τ in σ vor? (Skalar)
- An welchen Positionen kommt ein gegebenes Pattern τ in σ vor? (Liste)
- Wie viele verschiedene *Teilstrings* hat σ? (Skalar)
- Zähle alle Teilstrings von σ auf. (Liste von Strings)
- Hat σ ein Palindrom der Länge c? (ja/nein)
- Wie häufig kommen die einzelnen Buchstaben in σ vor? (Skalar)
- Was ist die grösste Entfernung zwischen zwei gleichen Buchstaben in σ? (Skalar)
- ...

Drittens soll das Tool erhaltene Testergebnisse auch geeignet visualisieren können. Es handelt sich dabei um diverse Arten von Ergebnissen: Boolesche Aussagen, Listen von Strings, Skalarwerten, usw. Im Gegensatz zu reiner Textausgabe, sind auch formatierte Ausgaben oder Grafiken denkbar.

Die Schnittstellen zwischen dem zu untersuchenden String, der Testfunktion und der Ausgabe sollen **viertens** vollkommen unabhängig von der Art der Eingabeerzeugung, der Art des Ergebnisses und der Art des Tests selbst sein.

Das Tool soll **fünftens** ein intuitives Benutzerinterface (UI) besitzen. Es braucht kein aufwendiges UI, muss aber so implementiert sein, dass es einfach ist, neue Stringerzeugungsmechanismen und Tests hinzuzufügen.

Wir definieren **sechstens** das Axiom, dass die Eingabe (für die Analyse) stets ***ein*** String ist (nicht mehrere gleichzeitig), und dieser über einem gegebenen Alphabet definiert ist. Die Integration eines Alphabetes ist in dieser Anforderung implizit enthalten. Auf diesem Alphabet soll zudem eine *Gewichtsfunktion* definiert werden können.

Bei vorhandener Restzeit nach Erfüllung obiger Aufgaben, könnten an diversen Stellen noch Verbesserungen vorgenommen werden. i) Ein grafisches Benutzerinterface, ii) die Möglichkeit, erzeugte Eingabestrings und Ausgaben auf einem persistenten Medium speichern zu können oder iii) alles was *gewichtete* Strings betrifft, sind Beispiele für zusätzliche Komponenten.

1.2 Übersicht

Die vorliegende Diplomarbeit erläutert im wesentlichen den Entstehungsprozess des *Tools*. Sämtliche Phasen des Softwareprojektes zeigen einen Einblick in die Entwicklung – angefangen bei ersten Ideen bis hin zur funktionierenden Software. Letztere stellt zugleich

den grössten Bestandteil dieser Arbeit dar und wird daher auch immer wieder in Form von *Sourcecode*[4] einbezogen. In einem speziellen Kapitel werden ein paar theoretische Fragestellungen im Zusammenhang mit gewichteten Strings untersucht. Dieses Kapitel ist nicht Bestandteil der Aufgabenstellung und wurde Dank schnellem Vorankommen und damit verbundenen Zeitreserven bis zum Abgabetermin ermöglicht. Es wird jedoch ausdrücklich darauf hingewiesen, dass in dieser kurzen Restzeit die wissenschaftlichen Untersuchungen nicht bis ins letzte Detail erörtert werden konnten.

Zur Einführung finden sich in Kapitel 2 Hinweise auf vorangegangene Arbeiten. In Kapitel 3 geben wir eine Auflistung von Gründen, wie und in welchem Kontext ein solches Tool das Leben eines Forschers erleichtern kann. Automation ist nur ein Stichwort in diesem Zusammenhang. Das grobe Konzept sowie eingehende Erläuterungen und Definitionen in Kapitel 4 sollen dem Leser die Thematik der Aufgabe näher bringen. Die konkreten Implementationen einzelner Fragestellungen werden in Kapitel 5 vorgestellt. Es werden aber bewusst nicht alle behandelt; sei dies aus der offensichtlichen Trivialität der Lösung einer Teilaufgabe oder aus fehlender Wichtigkeit für das Verständnis des gesamten Textes. Ein kompaktes Tutorial in Kapitel 6 dient als Crashkurs für die Bedienung des Tools. Dieses Kapitel ist sehr einfach geschrieben und soll einerseits als Dokumentation dienen und andererseits die restliche vorher noch nirgends erwähnte Funktionalität erläutern. Die bereits eingangs erwähnten theoretischen Untersuchungen zu gewichteten Strings, finden sich in Kapitel 7. In Kapitel 8 blicken wir in die Zukunft, und wir geben Hinweise, in welcher Weise der Kern des Tools noch erweitert werden könnte. Persönliche Einschätzungen und ein Fazit (aus verschiedenen Perspektiven) zu den erreichten Resultaten sowie dem Ablauf der Diplomarbeit, runden diese Arbeit mit Kapitel 9 ab. Es folgen noch zwei Anhänge, die Auflistungen, Verzeichnisse und grosse Teile des Sourcecodes enthalten. Die beiliegende CD komplettiert die Arbeit.

1.3 Verwendung dieser Diplomarbeit

Der vorliegende schriftliche Teil der Diplomarbeit hat seit dem Anfang seiner Entstehung im November 2001 einen schweren Stand. Grundsätzlich beinhaltet er die Entstehung eines Tools „from Scratch“ bis hin zur funktionstüchtigen, ausgelieferten Endversion. Die Arbeit umfasst prinzipiell alles von der Architektur (z. B. verwendete *Design Patterns* [2]) bis hin zum Einblick in die konkrete Implementierung einzelner Klassen. Im weiteren besitzt sie aber auch einen dokumentierenden Charakter. Jede Person, die das Tool in irgendeiner Weise später erweitern möchte, wird sich Teile dieses Textes als Referenzhandbuch zu Herzen nehmen. Lediglich teils sehr ausführliche Kommentare in den Sourcecode-Dateien ergänzen diese Informationsquelle. Schliesslich darf aber nicht vergessen werden, dass auch dieses Projekt unter dem Leitgedanken der Forschung entwickelt wird und nebst grossem Programmieraufwand auch vielerlei theoretische Aspekte mit sich bringt, deren Alltagstauglichkeit in diesem Zusammenhang erst noch erprobt werden müssen. Dabei handelt es sich zwar mehrheitlich um Gesichtspunkte aus Informatik (Algorithmen, Datenstrukturen, ...) – teils aber auch um solche aus der *Bioinformatik*. Generell verfolgen wir aber das Prinzip, das Rad nicht neu zu erfinden, und greifen auf jede vorhandene Information zurück, die als Lösung für unsere Probleme im allgemeinen in Frage kommt. Allerdings muss

[4] Der Sourcecode ist teilweise im Text selbst oder komplett auf der beiliegenden CD-ROM vorhanden. Ebenso sind verwendete Softwaretools von Drittherstellern und Dokumentationen in elektronischer Form vorzufinden. Weitere Informationen über den Inhalt der CD-ROM gibt die *readme.txt* Datei im Stammverzeichnis der CD.

jede verwendete Fremdressource all den hohen Ansprüchen, beispielsweise der Erweiterbarkeit im einzelnen, vollends genügen.

Um obigen drei Anforderungen Wissenschaftlichkeit, Dokumentation und Forschung gleichermassen gerecht zu werden ist die Diplomarbeit wie folgt aufgebaut. In jedem[5] Kapitel wird zuerst ein Problembewusstsein geschaffen. Im folgenden werden die Vor- und Nachteile möglicher Varianten diskutiert und schliesslich die als beste befundene im Detail erklärt. Viele Beispiele untermalen die oft trockene Materie. Es ist zudem unabdingbar, einen grossen Wert auf Einfachheit zu setzen, so dass auch Informatiker und Mathematiker ohne grosse Programmiererfahrung ihre Wünsche und Idee umsetzen können. Auf das Jonglieren mit technischen Ausdrücken – auch bekannt als Fachchinesisch – wird daher weitgehend verzichtet. Gegebenfalls wird „vor Ort“ in Fusszeilen für Klarheit gesorgt.

Bei obiger Vorgehensweise sollte der Grundgedanke eines sehr *generischen Frameworks* aber stets aufrechterhalten bleiben. Der geneigte Leser wird nicht umhin kommen, sich mit Programmiertechniken, Designgrundlagen und ähnlichem konkret auseinander zu setzen, um die Funktionsweise der entstandenen *STRing Analysis TOols*, kurz *STRATO*, bis ins Detail zu verstehen.

Hinweise auf einschlägige Informatik-Literatur einerseits und bisherige Projekte in diesem recht neuen Gebiet der Bioinformatik andererseits gliedern schliesslich das STRATO-Projekt in den derzeitigen Stand der Forschung ein und untermauern ein weiteres Mal dessen aktuellen Nutzen.

[5] Eine Ausnahme bildet hier das Kapitel 6: STRATO Tutorial (Version 1.0)

2 Vorangegangene Arbeiten

Falls der Leser in diesem Kapitel eine längere Auflistung vorangegangener Arbeiten erwartet, wird er sicherlich enttäuscht. Ein Tool/Framework in dieser Art und Allgemeinheit gibt es bisweilen, soweit ich weiss, nicht. Stattdessen finden sich aber Dutzende von Java-Applets und andere Applikationen, die jeweils ein konkretes String-Problem anvisieren. Ein weitbekanntes ist das *(Exact) Pattern Matching*. Die Aufgabe besteht darin, in einem gegebenen String T einen anderen String M zu finden. Eine Vielzahl an diversen Algorithmen wie jene von *Boyer-Moore* oder *Knuth-Morris-Pratt* [1], spezielle Datenstrukturen wie der *Suffix-Tree* [1] oder ganz andere Techniken und Datentypen lösen diese Aufgabe zumindest in linearer Zeit in Bezug auf die Eingabelänge. Für die Veranschaulichung unzähliger Algorithmen bildet die EXACT STRING MATCHING ALGORITHMS Bibliothek (ESMAJ) eine anschauliche Übersicht [8]. Zu jedem Algorithmus wird eine kleine Beschreibung, ein Stück C-Code, ein konkretes Beispiel und eine lange Referenzliste angeboten. Angaben zur Laufzeit und zum benötigtem Speicherplatz werden auch gegeben. Abrundend wird zu jedem Algorithmus ein interaktives Java-Applet angeboten. Diese und andere Webseiten wie [9] setzen als Ziel aber stets eine Visualisierung des Algorithmus selbst (und nicht dessen Resultates) und sind daher für uns eher uninteressant. Linklisten zu Büchern, Papern und Proceedings wie in [10] könnten hingegen wieder sinnvolle Ressourcen für das Schreiben von neuen Tests darstellen. Als Beispiel aus der Bioinformatik sei die Site [11] genannt. Sie enthält Implementierungen von vier bioinformatikspezifischen Algorithmen. Aber auch ETH-interne Projekte wie das Darwinprojekt [12] sind an dieser Stelle erwähnenswert.

Ähnlich wie bei der Analyse von Strings sind mir keine grösseren Arbeiten zur Stringerzeugung bekannt. Dies mag daher kommen, dass das alleinige Erzeugen eines Strings höchstens im Beziehung mit Compilern und Interpretern eine Rolle spielt, dort aber in einem ganz anderen Zusammenhang.

3 Motivation

Der Umstand, dass wir auf Neuland stossen, hat sicherlich nicht nur Vorteile. Die wirklichen Probleme bei unerforschten Gebieten zeigen sich oft erst im Verlaufe der Zeit und können plötzlich ungeahnte Dimensionen annehmen. Nichtsdestotrotz: Die Aufgabenstellung der vorliegenden Diplomarbeit ist aus vielerlei Perspektiven höchst interessant – nicht nur aus der Sicht eines Informatikers.

Während der „waschechte“ Anwender die enorme Zeitersparnis schätzt, die ihm eine arbeitsunterstützende Software ermöglicht, sieht der Programmierer die Motivation eher im internen Aufbau des Programms, welche obigem Anwender meist ganz and gar verborgen bleibt. Einzig an der Benutzerschnittstelle treffen die beiden Interessen aufeinander. Ein Userinterface sollte intuitiv und schnell bedienbar sein, ohne vorher zentimeterdicke Handbücher wälzen zu müssen. Grafiken sind zwar nicht ein Muss, aber Piktogramme und sonstige kleine Symbole erhöhen die Verständlichkeit der Kommunikation zwischen Software und User.

Beim gegenwärtigen Projekt ist aber nicht nur eine Benutzerschnittstelle zur Bedienung des Programms (graphisch oder auch nicht) vonnöten, sondern auch ein einfaches Interface zur Erweiterung des Tools; und dies ohne jegliche Kenntnisse der Interna. Diese Methode zum Ausbau eines Systems ist anderweitig besser bekannt unter dem Namen *Plug-in*, wie wir es beispielsweise von gängiger Internet-Browsersoftware wie Netscape™ und Internet Explorer™ kennen.

Selbst wenn die Implementierung einer Applikation oft zu grossen Teilen reine Fleissarbeit darstellt, ergeben sich in unserem Fall einige knifflige Fragestellungen, die selbst das durchdachteste Design nicht a priori unter den Tisch wischen kann. Welche Arten von Stringerzeugungsmethoden gibt es denn? Und wie implementiere ich dazu eine einfache, selbst für exotischste Ideen, ausreichende Umgebung? Worin unterscheidet sich der Komfort und die Einfachheit einer Schnittstelle von einer andern und wie muss sie gestaltet sein, dass nicht nur „Informatikgurus“ sie benutzen können? Wie erlaube ich dem Programmierer von Plug-ins Zugriff auf die Ressourcen, ohne dass er mit unsauber oder gar inkorrekt codierten Prozeduren/Methoden gleich das ganze System in die Knie zwingt? Geschieht dies durch strikte Trennung des Kerns und einer diesen Kern schützenden Schicht, welche den kontrollierten Zugriff darauf ermöglicht? Oder erlaube ich sogar sogenannte *Hooks* ins System, die Tür und Tor zu sämtlichen Ressourcen öffnen? Selbst Fragen zur Gestaltung eines Frameworks, so dass die einfachstmögliche Erweiterbarkeit auf der Anforderungsliste an erster Stelle steht, dürfen nicht fehlen. Und nicht zuletzt die Frage: Wie gestaltet man ein Framework, welches total unabhängig von der Benutzeroberfläche ist, aber trotzdem die vielfältigsten Ausprägungen von der Mensch-Maschine-Interaktion unterstützt? Fragen über Fragen, die in den folgenden Kapiteln sowie durch den Sourcecode selbst direkt und/oder indirekt beantwortet werden.

Auch vom theoretischen Standpunkt aus gesehen entstehen im Zusammenhang mit Strings immer wieder neue Ideen. Der Versuch, selbst Stringforschung an der Wurzel zu betreiben, findet sich wie eingangs erwähnt in Kapitel 7 dieser schriftlichen Abhandlung.

Schliesslich und endlich ist vor allem das junge Gebiet der Bioinformatik ausschlaggebend, sich intensiver mit dieser Materie zu beschäftigen. Selbst wenn Proteine, DNA-Ketten und andere sequenzielle Strukturen sich durch die Abstraktion auf Strings vollends von der Bioinformatik abkoppeln lassen, ist gerade der biologische Hintergrund die treibende Kraft,

aktuelle Algorithmen zu verbessern und neue zu erfinden. Und genau an dieser Stelle, wo bisher mühsam alles von Hand an sehr kurzen Strings ausgetestet werden musste, setzt dieses Framework und das darauf bauende Tool an.

4 Konzepte

Trotz der Tatsache, dass in der heutigen Welt schon lange nicht mehr nach dem alten Schema der 5 Hauptphasen *A) Vorabklärungen, B) Konzepte, C) Realisierung(Implementation), D) Tests/Bugfixes & E) Dokumentation* und *Betrieb* Software entwickelt wird, sondern immer mehr die Aspekte von Extreme Programming [3] angewandt werden, finden sich in dieser Diplomarbeit die Kapitelüberschriften „Konzepte“ und „Implementation“ allesamt wieder. Dies hat vor allem den einen Grund, dass der Leser die jeweiligen Kapitel auch als eigene Einheiten studieren kann. Als Beispiel hierzu nehme man einen versierten Anwender des Tools, der in keiner Weise an der Implementation interessiert ist, allerdings durch Kenntnisse der dahinterliegenden Konzepte das Tool noch besser einsetzen kann.

Nebst der Erweiterbarkeit des Tools fordern wir eine strickte Trennung zwischen den einzelnen Hauptkomponenten des Tools. Dieser Aspekt soll beispielsweise die Unabhängigkeit zwischen der STRATO Plattform und den *hotpluggable*[6] Testroutinen garantieren. Die Erweiterbarkeit in viele Dimensionen, sowie weitere Features der Aufgabenstellung werden in den folgenden Kapiteln nun genauer erklärt und untersucht.

4.1 Überblick

Einer der wichtigsten Punkte beim Programmieren von erweiterbarer Software ist eine saubere Architektur; insbesondere im Bezug auf Verwendung von Klassen und Interfaces. Damit der Gesamtüberblick beim Projekt STRATO nicht verloren geht, wird das Tool in erster Linie in seine Basiskomponenten zerlegt. Bei der Wahl dieser Grundpfeiler des Frameworks rangiert an erster Stelle die Einfachheit. Trotzdem soll auch bei komplexen Erweiterungswünschen genügenden Flexibilität gewährleistet sein, sodass die bestehenden Sockel nicht ersetzt werden müssen.

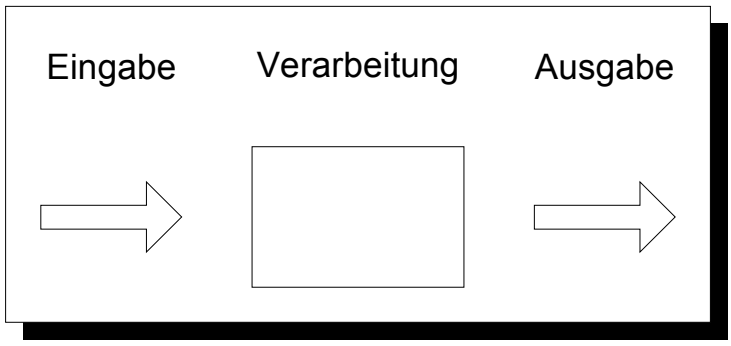

Abbildung 4-1: Das 3-Schichtenmodell: Eingabe, Verarbeitung, Ausgabe

Viele Überlegungen in diese Richtung führen leider immer wieder in ein komplettes *Overdesign*[7]. Dies geschieht einerseits aus Furcht, sich eine Option zu verbauen und andererseits ist es der wilde Eifer, selbst die generischste Klasse um eine weitere Stufe zu abstrahieren. Wir leiten den Aufbau der Architektur nun stufenweise her. Das Prinzip ist wohl

[6] Als hot-pluggable bezeichnet man alles (Hardware, Software), was zur Laufzeit eines System angeschlossen (oder auch abgekoppelt) werden kann, ohne das System neu starten zu müssen.
[7] Unter einem Overdesign versteht man ein Design, das viel zu allgemein gehalten ist für die wenigen Funktionen, die es im Moment unterstützen sollte.

so alt wie die Informatik selbst und basiert auf der 3er-Beziehung: **Eingabe**, **Verarbeitung**, **Ausgabe** (siehe Abbildung 4-1).

Zugegeben, dies allein sagt noch nicht wirklich viel aus. Wir postulieren daher an dieser Stelle ein paar Axiome, die wir im folgenden als gegeben annehmen. Gewisse dieser Axiome sind Teile der Aufgabenstellung.

- Die Eingabe besteht stets aus genau einem String (siehe Definitionen im Kapitel 4.2).
- Dieser Eingabestring wird in irgendeiner Weise verarbeitet (wir betrachten diesen Prozess im Moment als Blackbox) und produziert ein eindeutiges[8] Resultat als Ausgabe.
- Zwischen diesen unabhängigen drei Schichten kann Zugriff untereinander nur mittels klar definierter Schnittstellen erfolgen.
- Diese Schnittstellen sind meist die (eventuelle leicht ans Problem angepasste) Ausprägung eines oder mehrerer sehr bekannter und in der Praxis erprobten *Design Patterns* [2]. Sei dies, um die Erweiterbarkeit zu erhalten oder eine klar definierte (meist hierarchische) Datenstruktur zu erreichen.

4.2 Eingabe

Die in Kapitel 4.1 gezeigte Grafik der groben Struktur von STRATO wird nun etwas genauer unter die Lupe genommen. Wir zoomen dazu auf die erste Komponente, die Eingabe (siehe Abbildung 4-2).

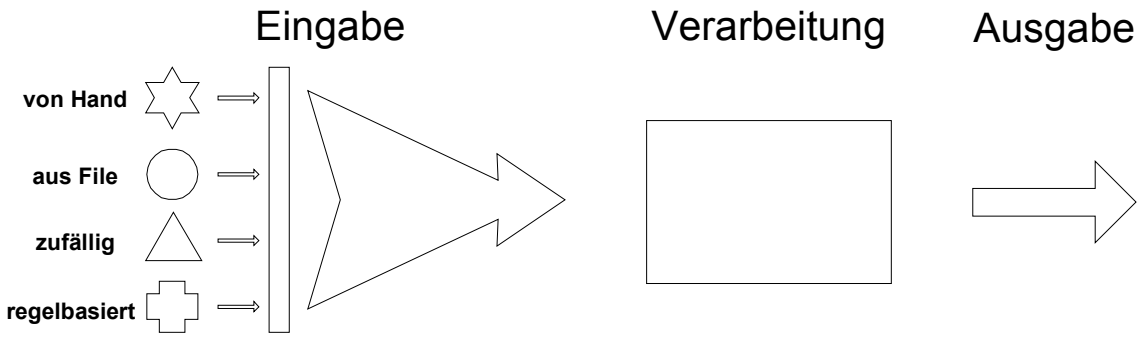

Abbildung 4-2: Verschiede mögliche Quellen für die Eingabe

Wie der Aufgabenstellung zu entnehmen ist, sollen nicht nur von Hand eingegebene Strings verwendet werden können. Die Quelle der Eingabe kann auch eine Datei, ein Eintrag aus einer Datenbank oder einfach ein Stringgenerator sein (eine weitere Blackbox, die nach gewissen Regeln Strings als Ausgabe produziert). Letztere lässt sich in mehrere Unterklassen aufteilen und wird in den folgenden Unterkapiteln ebenfalls genauer erläutert.

Doch bevor wir uns diesen Einzelheiten widmen, müssen wir uns überlegen, wie sich ein solcher Eingabestrings überhaupt zusammensetzt, sprich was einzutippen oder zu erzeugen ist. Folgende Definitionen werden fortan verwendetet.

[8] Dieser Einschub kommt daher, dass gewisse Tests eine Zufallskomponente enthalten können und das Resultat daher variieren kann. In andern Worten: Selbst wenn das Resultat aus einer nicht-deterministischen Funktion stammt ist es trotzdem eindeutig.

Definition A: Alphabet, Gewichtsfunktion, Gesamtgewicht, Teilmasse

Ein *Alphabet* $\Sigma = \{a_1,...,a_c\}$ ist eine endliche Menge von Zeichen der Grösse c. Auf ihnen besteht eine Ordnung. Wir definieren im weiteren eine *Gewichtsfunktion* $\mu : \Sigma \rightarrow IR^+$. Jedem Zeichen $z \in \Sigma$ kann ein Gewicht (oder eine Masse) $\omega_z \in IR^+$ zugeordnet werden. Standardmässig wird das Gewicht 1.0 zugeordnet, was dazu führt, dass das Gesamtgewicht μ eines *Strings* identisch mit der Länge dieses Strings ist. Das *Gesamtgewicht* eines Strings τ (formal bezeichnet mit $\mu(\tau)$), ist definiert als die Summe der Massen der einzelnen Zeichen in τ. Wir nennen im weiteren *M* eine *Teilmasse* von τ, falls τ einen *Teilstring* σ enthält, welcher die Masse $M := \mu(\sigma)$ besitzt.

Beispiel I: Ein gewichtetes Alphabet

Alphabet $\Sigma_1 = \{a, b\}$, $\omega_a=1.5$, $\omega_b=0.8$ ➔ binäres Alphabet mit den Gewichten 1.5, 0.8

Definition B: String

Ein *String* $\tau = z_1z_2z_3...z_n = \tau[1]\tau[2]\tau[3]...\tau[n]$ ist eine geordnete Folge von Zeichen z_i der Länge n. Die Zeichen z_i entstammen einem indexierten Alphabet Σ.

Wir bezeichnen die Multiplizität eines Zeichens a in τ mit $|\tau|_a := |\{i \mid \tau[i] = a\}|$. Der Multiplizität-Vektor eines Strings τ über einem Alphabet Σ ist $mult(\tau) := (|\tau|_{a1,...,} |\tau|_{ac})$.

Beispiel II: Ein einfacher String

Alphabet $\Sigma_2 = \{A, B, C\}$, $\tau_2 = ABBA$ ➔ ein String ($|\tau_2| = 4$) über einem Alphabet ($|\Sigma_2| = 3$)

Definition C: Teilstring, Prefix, Suffix

Für jeden String τ ist $\tau[i..j]$ der kontinuierliche, nichtleere *Teilstring* aus τ, welcher an Position i beginnt und an Position j in τ endet ($1 \le i \le j \le |\tau|$). $\tau[1..i]$ ist der *Prefix* von τ, welcher an Position i endet, $\tau[i..|\tau|]$ der *Suffix* des Strings τ, welcher an Position i beginnt. $|\tau|$ bezeichnet dabei jeweils die Länge des Strings (Anzahl der Zeichen).

Beispiel III: Spezielle Arten von Teilstrings

Alphabet $\Sigma_3 = \{a, b, c, d, e, f, g\}$, String $\tau_3 = abcaabc$ ➔ $\tau_3[1..3] = abc$, $\tau_3[2..5] = bcaab$, $\tau_3[1..|\tau_3|] = abcaabc = \tau_3$

Definition D: Palindrom

Ein *Palindrom* ist ein String, der sich vorwärts (von links nach rechts) wie auch rückwärts gleich liest. Beispiele: OTTO, ROTOR, LAGERREGAL, ...

Definition E: Komplementiertes Palindrom

Ein *komplementiertes Palindrom* ist ein DNA oder RNA String, welcher jeweils dann ein Palindrom wird, wenn für die Zeichen jeweils einer Seite des Strings die entsprechenden Komplementär-Zeichen eingesetzt werden. Bei der DNA sind A – T und C – G komplementär; bei der RNA sind dies A – U und C – G.
AGCTCGCGAGCT ist beispielsweise ein komplementiertes Palindrom für einen DNA String.

Damit die Definitionen noch aus einer anderen Sichtweise dargestellt werden können, greifen wir dem Kapitel 5 etwas vor. Wir erläutern an dieser Stelle unsere gewählte Implementation einzelner Definitionen.

Wir verwenden für die Elemente eines Alphabetes ASCII/UNICODE[9] Zeichen. Sie sind in einem Array gespeichert und ihre Ordnung lässt sich durch einen Indexvergleich bewerkstelligen. Index 0 ist für das Jokerzeichen[10] (z.B. §) reserviert, welches als Platzhalter für ein (noch) nicht definiertes Zeichen in einem String steht. Wird beispielsweise ein String nur durch die Vorgabe einer Länge erzeugt, wird dieser an allen Positionen mit dem Jokerzeichen initialisiert. Ein String ist aus Sicht der Implementation somit eine Folge von Indizes, die über einem konkreten Alphabet definiert ist.

Es folgt in Tabelle 4-1 eine Liste von Alphabeten, welche in STRATO implementiert sind.

NAME	BESCHREIBUNG	INHALT
UserDefined	Dynamisch erzeugtes Alphabet	beliebige UNICODE Zeichen
NotQuiteASCII	Fasst alle Zeichen des US-ASCII	{\u0020, …, \u0070}
UpperCase	Alle Grossbuchstaben des lateinischen A.	{A, B, C, …, X, Y, Z}
LowerCase	Alle Kleinbuchstaben des lateinischen A.	{a,b,c, …, x, y, z}
Digits	Alle arabischen Ziffern	{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
AlphaNum	Vereinigung der drei vorhergehenden A.	{A..Z, a..z, 0..9}
Hexadecimal	Die Zeichen des Hexadezimalsystems	{0, 1, 2, ..., E, F}
Binary	Ein binäres Alphabet	{a, b}
Dann	Die vier DNA Basen	{A, C, G, T}
Proteins	Die 20 häufigsten Aminosäuren	{A, R, N, D, C, …, W, Y, V}

Tabelle 4-1: Liste der vordefinierten Alphabete

4.2.1 Eingabe per Hand

Beinahe selbsterklärend ist die erste *String-Erzeugungs-Methode* (*SEM*), im folgenden als *Generator* bezeichnet. Die Eingabe per Hand ist die einzige Methode, bei welcher der Benutzer sämtliche „Bestandteile“ eines Strings (die Länge, das zugeordnete Alphabet, die enthaltenen Zeichen selbst, ...) frei bestimmen kann. Gewisse dieser Grössen sind jedoch voneinander abhängig. Betrachten wir dazu folgendes Szenario.

[9] ASCII: American Standard Characters of Information Interchange, 8-Bit Code für die am häufigsten verwendeten Zeichen; UNICODE ist eine neuer Standard, der im Gegensatz zu ASCII 16-bit besteht und Zeichen vieler verschiedener Sprachen (Chinesisch, Arabisch) aber auch Sonderzeichen aus der Mathematik umfasst. Genaueres findet man auf der UNICODE Homepage http://unicode.org.
[10] Das JOKER-Zeichen dient als Platzhalter für (noch) undefinierte Zeichen eines Strings.

Gegeben sei ein String *s* = roma. Die Länge von *s* ist |*s*| = 4. Wird nun der Strings *t* = ntisch an *s* angehängt (engl. append), wird *s* automatisch um dessen Länge |*t*| = 5 inkrementiert was zu *s* = romantisch und einer Länge von 9 Zeichen führt. Im Gegenzug kann der Benutzer aber auch die Länge von s verändern, was abhängig von der neuen Länge l' entweder dazuführt, dass ein Stück am Ende abgeschnitten oder der String verlängert wird[11]. Falls s auf Länge 3 gekürzt wird, ergäbe dies rom und auf Länge 6 ausgeweitet roma§§. Dieses Vorgehen ist ziemlich offensichtlich und daher nicht sonderlich spannend. Interessant wird's eigentlich erst, wenn wir beispielsweise das Alphabet ändern wollen.

Die einfachste Methode an dieser Stelle ist, eine solche Aktion – Transformation eines Strings τ_x (über dem Alphabet Σ_x) in einen String τ_y (über dem Alphabet Σ_y) – schlicht und einfach zu verbieten. Damit spart sich der Programmierer viel Konvertierärger und der Theoretiker die Mühe, die besagte Transformation formal zu notieren. Wie transformiert man beispielsweise eine Proteinsequenz (Alphabet der Grösse 20) in eine DNA-Sequenz (Alphabet der Grösse 4)? Es gäbe sicher diverse Lösungen für dieses Problem. Diese sind allerdings nicht Teil der vorliegenden Arbeit, und wir wenden uns daher wieder den SEMs zu.

4.2.2 Zufallsgeneratoren

Wird dem Computer die Aufgabe gestellt, eine zufällige Zahl zu erzeugen, kann dieser nur Pseudo-Zufallszahlen generieren; also Zahlen, die möglichst viele (oder manchmal auch nur einige gezielte) Eigenschaften von „echten" Zufallszahlen besitzen. Die Erzeugnisse eines solchen Generators stellen trotz alledem eine gute Approximation dar, und die Annäherungen sind für die meisten Programme ausreichend[12].

Damit man nicht Gefahr läuft, eine gebräuchliche Faustregel: *Never choose a random generator at random*[13] zu verletzen, stellen wir mehrere Zufallsgeneratoren zur Verfügung. Diese beruhen auf verschiedenen Modellen, welche allesamt mathematisch genauestens untersucht worden sind. Zwei bekannte Verfahren zur Erzeugung von (Pseudo-)Zufallszahlen sind unter den gleich folgenden Bezeichnungen in der Literatur zu finden. Die Informationen wurden dazu aus [4] entnommen und werden hier in gekürzter Form wiedergegeben.

Methode der linearen Kongruenz

Dies ist das bekannteste Verfahren, welches seit seiner Vorstellung durch D. Lehmer im Jahre 1951 fast ausschliesslich verwendet wird (Restmethode von Lehmer). Falls *seed* eine beliebige Zahl enthält, so erstellt die folgende Anweisung unter Benutzung dieser Methode ein Feld (Array) *a* mit *N* Pseudo-Zufallszahlen:

```
a[0] := seed;
for i:=1 to N do
  a[i] := (a[i-1]*b+1) mod m;
```

Die Bezeichner *b* und *m* stellen dabei Konstanten dar. Das Ergebnis ist stets eine Zahl zwischen 0 und *m*-1 (inklusive der Grenzen). So einfach die Methode doch scheint ist sie Gegenstand einer Bücher füllenden, ausführlichen und (in R. Sedgewick's Augen) schwierigen mathematischen Analyse. Aus deren Ergebnissen können wir schliesslich gewisse Anhaltspunkte für eine geeignete Wahl der Konstanten *seed*, *b* und *m* erhalten. Die Konstante *m* sollte gross sein (beispielsweise die Grösse eines Computerwortes), *b* hingegen weder zu gross noch zu klein. Am besten besitzt *b* eine Ziffer weniger als *m* und endet auf

[11] Die neu hinzugekommenen Zeichen werden in unserer Implementation mit Jokerzeichen '§' aufgefüllt!
[12] Ausnahmen bilden z.B. Anwendungen aus der Kryptographie!
[13] Quelle unbekannt

...x21, wobei x eine gerade Zahl darstellt. Der Wert des s*eed* schliesslich wird meist beim Benutzer des Generators erfragt und erfüllt daher keine besonderen Kriterien.

Methode der additiven Kongruenz

Diese zweite Methode beruht auf *Schieberegistern* mit linearer *Rückführung*, die für ältere Verschlüsselungsmaschinen verwendet wurden. Die Idee besteht darin, mit einem Register zu beginnen, das mit einem beliebigen Bitmuster gefüllt ist (nicht alles Nullen), und es dann (zum Beispiel) um einen Schritt nach rechts zu verschieben. Die dabei ganz links freigewordene Position (Bit) wird auf Grundlange des Registerinhalts gesetzt (zum Beispiel aus einer *XOR*-Verknüpfung[14] zweier genau festgelegter Bits des Registers).

Die aufeinanderfolgenden Registerinhalte sind als zufällige Folge allerdings nicht brauchbar, da in jedem aufeinanderfolgenden Paar alle Bits ausser einem übereinstimmen[15]. Vielmehr dient diese als eine Vorrichtung zur Erzeugung einer Folge zufälliger Bits. Wir können die in diesem Verfahren inhärente Rekursionsformel nun auch wortweise statt bitweise gebrauchen. Die Verwendung eines Schieberegisters mit Rückführung und mit *Abgriff* der Bits *b* und *c* entspricht der Anwendung der Rekursion

```
a[k] = (a[k-b] ⊕ a[k-c]) mod m.
```

Um den Zusammenhang mit den Schieberegistern zu gewährleisten, sollte das ⊕ in dieser Formel ein bitweises XOR sein. Die Implementation eines solchen Generators verlangt eine Tabelle der Grösse *c*, welche stets die letzten *c* erzeugten Zufallszahlen enthält. Die Berechnung erfolgt nun, indem eine der Zahlen in der Tabelle durch die Summe zweier anderer Zahlen in der Tabelle ersetzt wird.

Beim STRATO Projekt werden die Zufallsgeneratoren nicht ausschliesslich für die Erzeugung von zufälligen Buchstaben-Sequenzen (über einem vom Benutzer gewählten Alphabet) verwendet, sondern können auch bei den anderen Generatoren als Rauschen eingebunden werden, um nicht immer dieselben, identischen Strings zu erzeugen[16].

Um dieses Ziel zu erreichen, wurde ein einfaches Interface entworfen, welches von allen Zufallszahlen-Generatoren implementiert wird. Schon bestehende Javaklassen aus der Standardbibliothek werden dazu gekapselt, wobei intern aber alle Befehle lediglich mittels *Delegation* and die gekapselte Klasse weitergegeben werden. Oder mit andern Worten: Gegen aussen besitzen alle Varianten dasselbe Interface, intern aber wird die Zufallszahl entweder nach obigen Regeln selbst generiert oder durch den Aufruf einer Methode aus der Standardbibliothek gewonnen.

4.2.3 Reguläre Ausdrücke

Nach den zwei am häufigsten verwendeten Generatoren wollen wir uns nun einer ausgeklügelteren Methode der Stringerzeugung widmen. Einen Eingabestring von Hand einzutippen mag sicherlich eine der effektivsten Arten sein, einen String zu erzeugen, der ganz genau spezifizierten Anforderungen genügt. Bis zu einer Länge von 20-30 Zeichen ist dies auch kein Problem. Sollte sich die Stringlänge aber im Bereich von mehreren tausend Zeichen bewegen, wird das Tippen dieses Bandwurmes beinahe zur Sisyphusarbeit. Auch die Tatsache, dass man teils grosse Redundanz (Wiederholungen bestimmter Muster, …) im

[14] a XOR b: Exklusives Oder. Ist immer dann und nur dann wahr, wenn entweder nur a wahr oder nur b wahr ist.
[15] Die Bits sind allerdings um eine Position verschoben!
[16] In der regalbasierten Stringerzeugung (siehe Kapitel 4.2.4) werden beispielsweise nicht definierte Zeichen – unter Berücksichtigung des Alphabets – zufällig gewählt.

Eingabestring wünscht, führte uns auf die Idee, diesen doch am besten als eine Art regulären Ausdruck (engl. Regular Expression, kurz RegEx) dem System zu übergeben. Das System wertet eine solche „Rechnung“ schliesslich aus und erzeugt daraus einen äquivalenten, in voller Länge ausgeschriebenen String. Um diesen Sachverhalt zu veranschaulichen betrachten wir folgendes Beispiel:

Aufgabe: Erzeuge einen String, der mit 500 aufeinanderfolgenden Teilstrings *ab* beginnt, dann eine Kombination aus den Zeichen *c* und *d* enthält und mit weiteren 300 aufeinanderfolgenden Sequenzen von *ef* endet. Das ganze soll zudem durch ein Präfix *xyz* und ein Suffix *zxy* begrenzt sein. Um diese Zeichenfolge zu erzeugen, müssen wir versuchen, obige Aufgabe in einen Ausdruck umzuwandeln. Der gefundene Ausdruck kann anschliessend von einem Parser *gescannt*, auf Korrektheit geprüft und zum Schluss ausgewertet werden.

Wir begeben uns kurzzeitig in die Welt der Compilerbauer, um diesen Anforderungen gerecht zu werden. Eine ausführliche Beschreibung über den Bau eines Compilers wäre an dieser Stelle ein absoluter Overkill, da ein simpler Interpreter unseren Ansprüchen mehr als genügt. Stets nach dem Motto, das Rad nicht neu zu erfinden, verwenden wir für unsere Zwecke ein bereits existierendes Tool, das aus einer Reihe von Produktionsregeln einer kontextfreien Sprache den zugehörigen Parser generiert. Im Falle von regulären Sprachen würde dazu allerdings schon ein endlicher Automat genügen. Um aber die Erweiterbarkeit des Tools aufrecht zu erhalten, gerechtfertigt sich der Gebrauch eines solch mächtigen Compiler-Compiler-Tools.

Im Internet tummeln sich Dutzende von Tools, die sich mit der automatischen Parsererzeugung auseinandersetzen. Alleine für die Programmiersprache Java finden sich eine grosse Zahl von populären Hilfs-Tools: ANTLR, BYACC/JAVA, COCO/JAVA, CUP, JACCIE, JAX, JAY, JELL und so weiter. Die Wahl bei unserer Implementation fiel schliesslich auf JAVACC[17], weil das Tool als Eingabe eine Liste von Produktionsregeln verlangt, die man ohne Probleme von der gebräuchlichen *Erweiterten Bakus-Naur Form* (*EBNF*) in die Syntax einer Eingabedatei des JAVACC Compiler-Compilers umwandeln kann. STRATO unterstützt eine selbstentwickelte und recht umfangreiche Sprache REGULA. Zur Funktionsweise des Compiler-Compilers konsultiere man die zugehörige Dokumentation.

```
REGULA: Version 1.0 © by Daniel Emmenegger

result      ::= concat

concat      ::= string { string }
string      ::= base [ <POT> logical ]
base        ::= <CHAR> | <CHAR2> | <SEQ>) | '(' concat ')'

logical     ::= sum { ( <AND> | <OR> | <XOR> ) sum }
sum         ::= term { (<PLUS> | <MINUS> ) term }
term        ::= exp { ( <MULT> | <DIV> ) exp }
exp         ::= unary { <EXP> exp }
unary       ::= [ <MINUS> ] element
element     ::= <CONSTANT> | function | '(' logical ')'
function    ::= <ID> '(' [ logical { ',' logical } ] ')'

<AND>       ::= '&'     <OR>        ::= '|'     <XOR>       ::= '°'
<PLUS>      ::= '+'     <MINUS>     ::= '-'
<MULT>      ::= '*'     <DIV>       ::= '/'     <POT>       ::= '^'
```

[17] Siehe dazu URL http://www.webgain.com/products/java_cc/ oder die beiliegende CD-ROM

```
<CONSTANT>  ::= <FLOAT> | <FLOAT> [ ['e','E'] (['-','+'] ] <INTEGER>
<FLOAT>     ::= <INTEGER> | <INTEGER> [ '.' <INTEGER> ] | "." <INTEGER>
<INTEGER>   ::= <DIGIT> { <DIGIT> )

<DIGIT>     ::= '0'..'9'

<CHAR>      ::= <LETTER> | <SPECIAL>
<CHAR2>     ::= <ESC> ( <ASCII> | <STRQT> | <ESC> )
<SEQ>       ::= <STRQT> ( <ASCII> | <ESC> ) ( <ASCII> | <ESC> )* <STRQT>
<ID>        ::= <FUNQT> <LETTER> ( <LETTER> | <DIGIT> )

<ASCII>     ::= ' ' | '! ' | '#'-'[' | '] '-'~'
<LETTER>    ::= 'a'..'z' | 'A'..'Z'
<SPECIAL>   ::= '!'-''' | '.' | ':' | ';' | '<' | '=' | '>' | '?' |
                '['-'`' | '{'-'~'
<ESC>       ::= '\'
<STRQT>     ::= '"'
<FUNQT>     ::= '$'
```

Nun aber zurück zu unserem Wunsch, diesen mehr als 800 Zeichen langen Spezial-String zu erzeugen. Gemäss obiger Produktionsregeln handelt es sich nur noch um eine kurzen Ausdruck der Form: *xyz(ab)^500cd(ef)^300zxy*, der nun innert weniger Sekunden eingetippt und in noch viel kürzerer Zeit ausgewertet werden kann.

Die Vielfalt der möglichen Strings ist lediglich durch den Ideenreichtum des Anwenders beschränkt und ihr ist absolut keine Grenze gesetzt. Die bestehende Sprache REGULA kann zudem durch Hinzufügen zusätzlicher Produktionen erweitert werden. Der Liste der Produktionsregeln ist zu entnehmen, dass wir in einem logischen Ausdruck (`logical`) auch Funktionen aufrufen können. Diese werden mit dem Sonderzeichen '`$`' eingeleitet. REGULA 1.0 kennt dabei `$max, $min, $sin, $cos, $tan`. Sollen auch Ziffern und Sonderzeichen (Punkt, Komma, ...) im erzeugten String enthalten sein, muss diesen ein Escape-Zeichen '\' vorangestellt werden. Alternativ kann eine Folge von Spezialzeichen (alle Zeichen ausser den Buchstaben) auch als Sequenz in Anführungszeichen "..." stehen.

Folgende Beispiele sollen aufzeigen, was mit Hilfe der derzeitigen Version 1.0 von REGULA unter anderem erzeugt werden kann.

Beispiel IV: Ein verschachtelter Ausdruck

Ausdruck: b(ba(ab((ac)bc(bba)^2)^3)b)^2b
Lös: *bbaabacbcbbabbaacbcbbabbaacbcbbabbabbaabacbcbbabbaacbcbbabbaacbcbbabbabb*

Beispiel V: Komplexe Exponenten → kein Problem

Ausdruck: ab^(3+max($sin(35), $cos(55))*7)a
Lös: *abbba*

Beispiel VI: Spezialzeichen mit führendem Escape '\' oder als Sequenz "..."

Ausdruck: (IE^3\ "1394"\ Firewire\.)^2
Lös: *IEEE 1394 Firewire.IEEE 1394 Firewire.*

4.2.4 Regelbasierte Stringerzeugung

Der Prozess des Erzeugens von Strings ist eigentlich die Umkehrfunktion der Analyse von Stringeigenschaften. Wir synthetisieren einen String nach vorgegebenen Regeln, Statistiken, etc. Dabei können einerseits Vorgaben an den zu generierenden String gemacht werden, die sehr konkret sind (zum Beispiel „der fünfte Buchstabe ist ein 'c'") oder andererseits bloss unscharfe Wünsche formuliert werden. Ein Beispiel für Letzteres wäre die Forderung, dass die Vokale {a, e, i, o, u} alle etwa gleich häufig vorkommen oder dass ein beliebiges Palindrom der Länge 7 mindestens zwei Mal auftaucht. Werden nicht alle Zeichen eines Wortes vorgegeben, können die restlichen Zeichen frei (beliebig) gewählt werden können.

Auch an dieser Stelle könnte eine Art Beschreibungssprache (siehe Kapitel 4.2.3) eine Lösung darstellen, welche obige und andere Anforderungen an den synthetisch hergestellten String formalisiert. Allerdings wollen wir uns nicht ein weiteres Mal in der Welt der regulären, kontextfreien oder andern Sprachen wiederfinden. Es gibt eine andere, einfachere Möglichkeit, die gewünschten Eigenschaften (oder zumindest Teile davon) zu spezifizieren. Dem Ganzen muss auch noch eine Zufallskomponente beigemischt werden, damit bei jedem Versuch ein neuer String entsteht[18].

Es folgen zwei alternative Lösungsansätze, die uns erlauben, einen String nach Regeln zu erzeugen.

Variante A:

Bei einer ersten Variante teilt man die Anforderungen (gewünschten Eigenschaften) in Gruppen ein, welche mit Prioritäten versehen werden. Oder mit anderen Worten gesagt: es wird versucht, den neuen String als Verknüpfung verschiedenster Funktionen zu erzeugen, welche sich gegenseitig beeinflussen können und daher einer Ordnung unterstehen müssen (vergleiche *Punkt-vor-Strich-Regel* in der Arithmetik). Eine mögliche Kategorisierung dieser Typen von Funktionen könnte wie folgt aussehen: geordnet nach Priorität der Auswertungsreihenfolge, beginnend mit der höchsten. Bemerkung: Wir erzeugen einen String τ der Länge N basierend auf Alphabet Σ.

1.Priorität
Position ($\in$ [1..N]) **und Wert des Zeichens** ($\in \Sigma$) **vorgegeben**
zum Beispiel $\tau[4]$ = 'x'
➔ An der vierten Stelle des Strings steht ein 'x'

2. Priorität
Position gegeben, Wert des Zeichens ist abhängig von Funktion *f*
zum Beispiel $\tau[9]$ = *randomCharOfList*('u','i','t')
➔ Der String hat an Position 9 ein 'u ', ein 'i ' oder ein 't ' (alle mit derselben Wahrscheinlichkeit). Bem: Die Funktion *f* kann aber „beliebig" komplex sein (wie auch ihre Parameter)

[18] Ausgenommen, man wählt immer dasselbe Seed für den Zufallsgenerator.

3. Priorität
Wert des Zeichens gegeben, Position abhängig von Funktion *f*
zum Beispiel str[*atRandomOddPosition()*] = 'h'
➔ An einer beliebigen ungeraden Index-Position (also 1,3,5,...) kommt ein 'h' vor:
Beispiel: „cabbaghggbaac"

... usw ...

(n-1). Priorität
Statistische Angaben über Häufigkeiten der Zeichen im String τ

n. Priorität
Restliche, noch nicht belegte Plätze gemäss zugeordnetem Alphabet zufällig füllen

Die hier aufgeführte Liste betrachtet jeweils nur ein einzelnes Zeichen, kann aber auch analog auf Teilstrings ausgedehnt werden.

So gut diese Idee im ersten Moment scheint, so schwierig ist dabei zu beweisen, dass so auch wirklich alle erdenkbaren Strings mittels Kombination obiger Varianten erzeugt werden können. Hinzu kommt die Tatsache, dass selbst einfachste Eigenschaften zuerst einmal in dieses doch sehr restriktive Schema transformiert werden müssen. Jede Anforderung müsste als Kombination von Position *P*, Wert des Zeichens *W* und eventuellen Zusatzinformationen spezifiziert werden. Im weiteren müssten sowohl *P* wie auch *W* eine Funktion *f* darstellen, die als Parameter einen Funktionsvektor *g* hat, welcher selbst wiederum einen Argumentenvektor *x* als Parameter besässe (Beispiel: $P = f(g(x))$).

Variante B:

Ein viel einfacherer Weg besteht darin, einfach nichts vorzugeben! Der Programmierer der SEM muss lediglich die Möglichkeit besitzen, Parametereingaben durch den Benutzer auswerten zu können.

Wir haben uns für dieses Modell entschieden, da es keinerlei Einschränkungen vorgibt, und das Parameterhandling lässt sich sehr einfach handhaben. Falls mit der Zeit immer mehr unterschiedliche Regeln implementiert werden, kann man sich zu gegebener Zeit erneut mit dieser Fragestellung auseinandersetzen. Im Moment besteht aber keine Anlass dazu. Da dieses Unterfangen gut realisierbar ist, verweisen wir ausschliesslich auf die entsprechenden Sourcecode-Dateien und die Erklärungen im noch folgenden Kapitel 5.2.4.

4.2.5 Laden aus persistenten Quellen

Es gibt eine grosse Vielfalt von permanenten Speicherungssorten: Dateien (ASCII, binär) Datenbanken (relational, objektorientiert), auf welche mit den unterschiedlichsten Techniken zugegriffen werden kann. Für relationale Datenbank Management-Systeme (DBMS) eignet sich besonders die *Structured Query Language*, kurz *SQL*, welche mit den bekannten *SELECT – FROM – WHERE* Termini die benötigten Daten beschreibt. Damit wir uns auch hier nicht in den Tiefen der Informationssysteme verlieren, wurde vorerst nur die Speicherung in eine ACSII Textdatei ins Visier gefasst[19]. Dazu kommt der Vorteil, das Daten aus Datenbanken meist in ein Textfile extrahiert, danach mit einem einfachen Algorithmus ins richtige Format konvertiert und eingelesen werden können.

[19] Textdateien sind stets human-readable und können im Gegensatz zu Binärdateien mit einem beliebigen Texteditor eingesehen oder manipuliert werden.

Unser Format sieht vor, zumindest immer den Stringinhalt selbst zu speichern. Optional sind Angaben über das verwendete Alphabet und dessen Eigenschaften (zum Beispiel Gewichte) möglich. Fehlen diese weitergehenden Informationen , werden standardmässig vordefinierte Werte angenommen (siehe Abbildung 4-3).

```
String als UNICODE/ASCII Zeichenfolge (eventuell
verteilt über mehrere Zeilen)
```

```
                    Leerzeile
```

```
Das Alphabettag: ¦alphabet¦
Das Alphabet in einer der folgenden Formen:
  a) als Zeichenfolge: z1z2z3...zM
    -> zi ist das i-te Zeichen des Alphabets
  b) als Klassenname (inklusive Packages)
    -> packageA.packageB.....packageK.Klasse
```

```
Das Gewichtstag: ¦weights¦
Die Gewichte in vektorieller Form:
  "[" w1, w2, w3, ..., wM "]"
    -> wi ist das Gewicht des i-ten Zeichens
```

```
Das IDtag: ¦id¦
Der Bezeichner als einzeilige Zeichenfolge
```

Abbildung 4-3: Dateiformat eines Eingabestrings

```
ACGCCCAGGACTTTATATATCCTCATCA
TTCTAGAGTCTAGACGTTACCCCCCCCC
CCCTATGAGGGGCA

¦alphabet¦
com.strato.lib.alphabet.CDna
¦weights¦
[0.0, 1.0, 0.7, 2.5, 0]
¦id¦
2002-01-02 $Random$ QXTB HuGero9b
```

Abbildung 4-4: Eine Beispieldatei für unser Dateiformat

4.3 Test-Box

Dieses Unterkapitel bildet meines Erachtens den spannendsten Teil der 3er Gruppe: Eingabe – Verarbeitung – Ausgabe. Ein paar Beispiele sollen den Zweck dieser Test-Box veranschaulichen.

Beispiel VII: Ein Test zu Exact Pattern Matchings

Geg: τ = bcababcabcaccabcaccb, Σ = {a, b, c, d}
Ges: Wie oft und an welchen Positionen tritt der Teilstring abca auf?
Lös: 3 mal, an den Startpositionen {5, 8, 14}

Beispiel VIII: Ein Test mit gewichteten Strings

Geg: τ = xxyxyxxy, Σ = {x, y}, ω_x=1.5, ω_y=0.8
Ges: Wie gross ist das Gesamtgewicht von τ?
Lös: 5 * 1.5 + 3 * 0.8 = 9.9

Beispiel IX: Ein Test zur Erkennung komplementierter Palindrome

Geg: τ = AGCGCT, A = 'DNA-Alphabet'
Gesucht: Ist τ ein komplementiertes Palindrom?
Lös: τ' = TCGCGA (das „Komplement" von τ) → c = τ[1..|τ|]τ'[1..|τ'|] = AGCGCTTCGCGA
→ Neue Fragestellung: Ist c ein Palindrom? Ja → τ ist komplementiertes Palindrom!

Wie soll eine Blackbox gestaltet werden, so dass ihr Interface gegen aussen so minimal und einfach ist, dass trotz allem die Realisierung einer Unmenge verschiedenartiger Tests möglich wird. Betrachten wir den einfachst möglichen Fall.

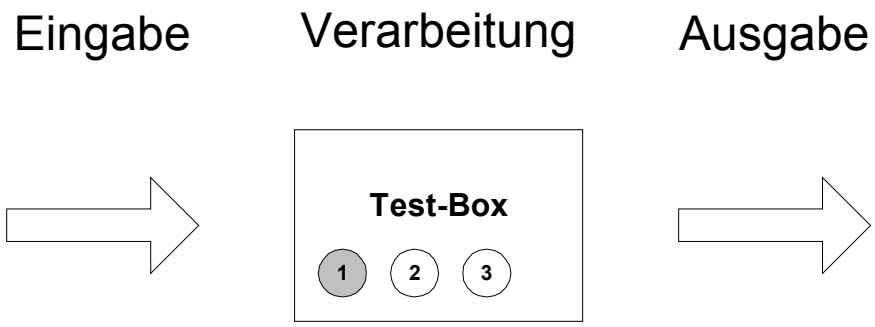

Abbildung 4-5: Basismodell einer Test-Box

Die in Abbildung 4-5 dargestellte Blackbox (Test-Box) erlaubt durch Auswahl einer der drei Schalter (Knöpfe) eine Algorithmusvariante[20] zu wählen, die auf die Eingabe angewandt wird. Doch selbst wenn die Anzahl der selektierbaren Schalter enorm gross wäre, würde es stets nur eine endliche Anzahl verschiedener Varianten *eines* Tests geben. Zudem sind die wenigsten Tests unabhängig von weiteren Parametern, und so müsste für jeden möglichen Parameter aus einer unendlichen Menge (Universum) ein eigener Test geschrieben werden. Diese Art von Testbox erlaubt also bloss Tests, die unabhängig von weiteren Eingabedaten sind. Dazu zählt jede Art von statistischen Auswertungen. Wir definieren daher einen weiteren Typ eines „Blackbox-Eingangs" – die Parameter eines Tests (siehe Abbildung 4-6).

Die Parameter besitzen immer einen bestimmten Datentyp. Die zugewiesenen Werte liegen daher immer im Definitionsbereich dieses Datentyps. Parameter sind somit vergleichbar mit Variablen einer höheren Programmiersprache. Nun sind bereits Anfragen wie „Kommt ein Pattern *P* in Eingabestring *s* vor?" möglich. Der Parameter (das Pattern *P*) ist vom Typ String. Wenn wir diesen Sachverhalt formal für mehrere Parameter ausdrücken wollen, liesse sich diese Stufe der Blackbox als Funktion f darstellen

[20] Unter Variante verstehen wir hier die Auswahl des Resultattypes: Beispielsweise kann der Algorithmus von Boyer-Moore nicht nur die binäre Frage, ob ein Substring in einem andern enthalten ist, beantworten, sondern auch noch deren Positionen im String angeben. Die beiden Varianten sind grundlegend verschiedene Fragestellungen, die allerdings durch fast den selben Algorithmus beantwortet werden können.

Funktion f: **x** → y = f(*s*, x_1, ..., x_n). Die x_i stellen dabei die Parameter dar, *s* ist der Eingabestring und y die berechnete Ausgabe.

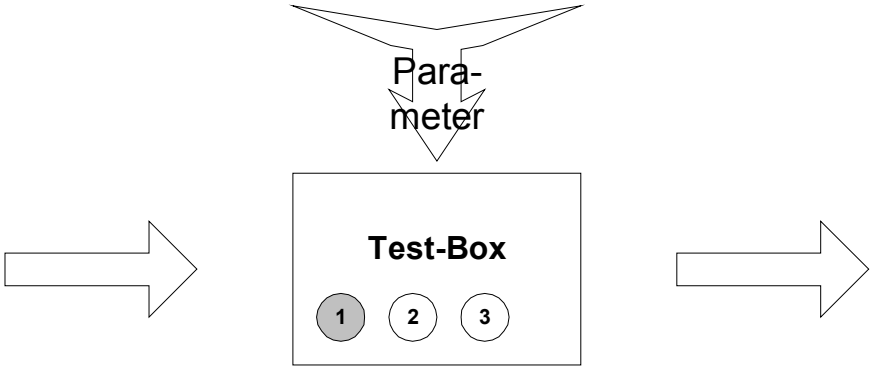

Abbildung 4-6: Erste Ausbaustufe des Basismodells durch Einführung von Parametern

Dieses Modell ist für unsere Zwecke ebenfalls noch nicht ausreichend. Wir haben einen entscheidenden Punkt noch nicht berücksichtigt. Diese Testbox ist bloss durch die Wahl der Parameter erweiterbar. Grundlegend neuartige Tests oder deren Varianten können nicht einfach eingebunden werden, ohne die Blackbox selbst zu erweitern. Wir wollen nun versuchen, den Test selbst nicht als festen Bestandteil der Blackbox zu sehen. Aber auch dessen Varianten (hier Knöpfe) sollen je nach Test verschieden gewählt werden können. Dies führt dann zur letzten Ausbaustufe des Modells, welche es erlauben soll, die Tests zur Laufzeit auszuwählen, indem ins System ein Modul eingesteckt wird (siehe Abbildung 4-7). Dieses Modul enthält schliesslich alles, was es zur Berechnung der Ausgabe benötigt. Es teilt dem System unter anderem mit, welche Parameter es verlangt und welche Varianten angeboten werden. Diese Technik ist weitverbreitet, um die Erweiterbarkeit von Software sicherzustellen und am besten unter dem Namen *Plug-in*[21] bekannt.

Diese flexible Lösung hat allerdings einen kleinen Nachteil. Die sogenannte Plug-in-Schnittstelle muss allen möglichen Anforderungen gerecht werden – d.h. sehr generisch sein. Ob ein Test komplex ist oder sehr einfach, stets müssen die selben Informationen ans System geliefert werden. Damit die Implementation einfacher Tests nicht zur reinen Schreibarbeit und das Schreiben komplexer Tests nicht unnötig kompliziert wird, wurde eine Schnittstelle entwickelt, die nur so viel Information verlangt wie nötig. Genauere Informationen dazu finden sich im Kapitel 5.3.

[21] Nicht zu verwechseln mit Skins, die im Zusammenhang mit einer zur Laufzeit wählbaren graphischen Oberfläche der Benutzerschnittstelle verwendet werden.

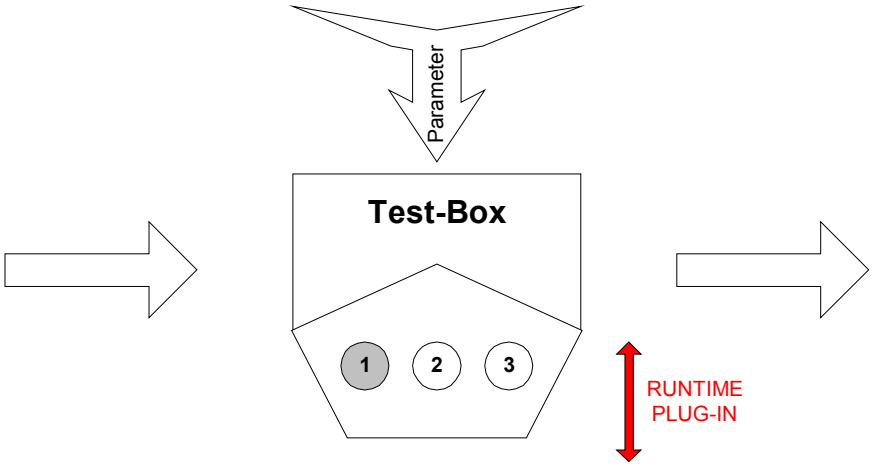

Abbildung 4-7: Finale Version der Test-Box mit integrierter Plug-in Funktionalität

4.4 Ausgabe

Zurück zu unserem anfangs Kapitel 4 vorgestellten Modell. In der selben Manier wie beim Eingabemodell, könnte die Methodik der verschiedenen Quellen analog für die Ausgabe verwendet werden (natürlich in umgekehrter Richtung, d.h. für diverse Zielorte). Doch welche Informationen werden überhaupt ausgegeben, gespeichert oder sonst in irgendeiner Art weiterverarbeitet?

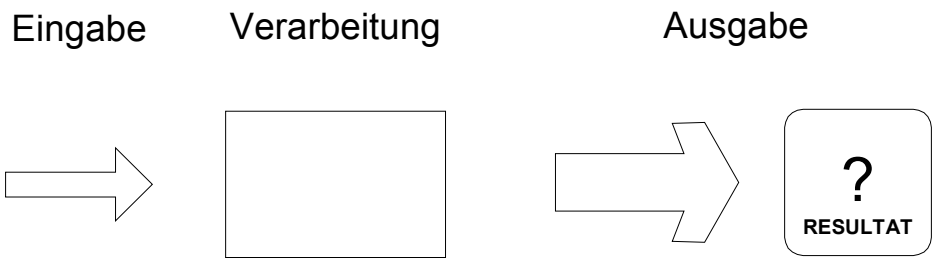

Abbildung 4-8 Das berechnete Resultat

Im Gegensatz zur eingangs erwähnten Eingabe, produzieren wir eine Art Ausgabeobjekt, das sich aus mehreren Werten zusammensetzt. Verschafft man sich einen Überblick, wo sich die interessanten Informationen befinden, die mit der Ausgabe in einer nicht zu vernachlässigenden Art in Verbindung stehen, findet man gleich mehrere Datenquellen.

In den seltensten Fällen wird bloss die Ausgabe eines Tests gespeichert werden; vielmehr werden der dazugehörige Eingabestring, der verwendete Test oder die eingegebenen Testparameter das Ausgabeobjekt abrunden. Denn blosse Angaben wie „das Resultat ist fünf“, bringt bei späteren Recherchen, z.B. in einer eigens angelegten Datenbank mit Testresultaten, eine unbrauchbare Information. Abbildung 4-9 zeigt das ganze Projekt und die grauen, nummerierten Pfeile deuten auf mögliche Informationen hin, die im Ausgabeobjekt enthalten sein könnten. Das Auge bezeichnet dabei die optische Anzeige der Eingabe. Die Registerzungen im unteren Teil des Bildes zeigen die verschiedenen Kategorien von Tests, aus denen der verwendete Test gewählt wird.

Wir haben uns entschieden, sämtliche Informationen, die beim Ausführen eines Tests involviert sind, in das Resultatobjekt zu speichern. Dieses Resultatobjekt ist logischer Natur, d.h. es setzt sich aus mehren physischen Komponenten zusammen. Dies sind im einzelnen:

(1) Der Eingabestring (inklusive Angaben über das Alphabet, etc.),

(2) der verwendete Test (Name, Beschreibung und gewählte Variante),

(3) die Werte der Parameter und

(4) das Resultat selbst (ein Zahl, ein String, ein Array, ...).

Aus der Sicht der Implementation stellt das Resultatobjekt eines Ansammlung von Variablen dar. Diese werden entweder implizit vom Test (2, 3, 4) zur Verfügung gestellt oder sind Teil des Systems (1). Die Werte dieser Variablen können (bei unserer Implementation) in Form einer HTML formatierten Textdatei persistent gespeichert werden.

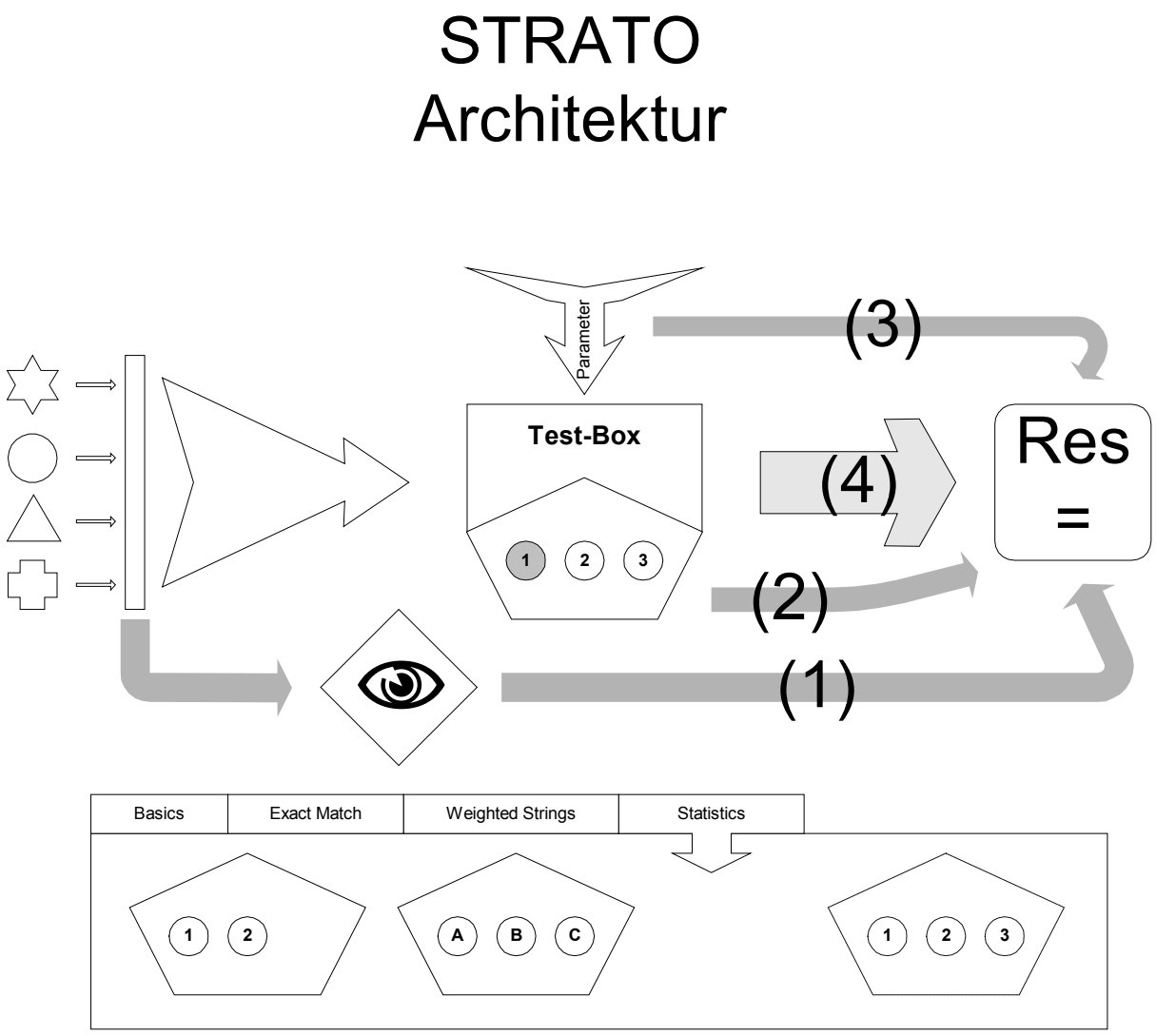

Abbildung 4-9: Gesamtübersicht zur Architektur von STRATO

4.5 Zusammenfassung

In diesem Kapitel wurden die Grundpfeiler des Frameworks präsentiert. Wir kennen nun verschiedene Quellen und Typen von Generatoren, welche Eingabedaten für unsere Tests liefern. Die Palette reichte von der Auswertung regulärer Ausdrücke über das Laden aus einer Datei bis hin zum manuellen Eintippen des Eingabestrings. Im weiteren hatten wir die Anforderungen an die Plug-in Schnittstelle erörtert und diese schrittweise hergeleitet. So

garantieren wir eine fast uneingeschränkte Erweiterbarkeit in Bezug auf die selbst programmierten Analysefunktionen. Dass ein Resultat nicht nur durch eine einzelne Zahl repräsentiert wird und die Verwaltung der Ergebnisse ein besonderes Augenmerk verdient, zeigte der letzte Teil des Kapitels. Abgerundet wurde das ganze durch einige Definitionen und viele Beispiele.

5 Implementation

Die Theorie ist nur eine Seite der Medaille. Das schönste Design ist nur wenig wert (oder gar nichts), wenn es sich in der Praxis nicht bewährt – dann nützen auch die farbenfrohesten Zeichnungen und Diagramme nichts. *Falls es Quellcode gibt, der sich am besten in Bildern ausdrücken lässt, dann sollte man ihn in Bildern ausdrücken, bearbeiten und pflegen*[22]. Sagt eine kurzes Programmfragment aber genauso viel aus, kann man sich die Zeit sparen, in der man mit den CASE-Tools[23] kämpft, um ein verständliches Fluss-Diagramm oder ähnliches zu produzieren. Die folgenden Unterkapitel geben nur einen kleinen Einblick in die Implementation der in Kapitel 4 diskutierten Konzepte. Es wird auch bei weitem nicht alles erklärt werden. Die ausgewählten Themenkreise sollten aber im Grossen und Ganzen einen Gesamteindruck des Programms STRATO vermitteln können. STRATO wurde mit JAVA™ (Version 1.3) programmiert .

5.1 Allgemeines

5.1.1 „Coding Conventions“

Die Analyse fremder Sourcecode-Dateien (ohne Dokumentation) mit vielen kryptischen Variablen und Methodennamen und deren noch mysteriösere Verwendung kann oft zur Geduldsprobe werden. Aus diesem und anderen Gründen[24] wurden die gängigen Javarichtlinien[25] angewandt, allerdings um ein paar Ideen erweitert. Im Anhang findet sich zusammenfassend eine Liste mit den wichtigsten Konventionen.

5.1.2 „Packages“ – Ordnung muss sein

Java besitzt seit Version 1.0 die Möglichkeit seine mittlerweile mehrere 1000 Klassen (inklusive inneren Klassen) umfassende Standard-Bibliothek kategorisch zu organisieren. Dies geschieht mittels sogenannter *Packages (*Klassenpakete*)*, deren hierarchische Struktur an ein Dateisystem erinnert.

Dasselbe Prinzip der Organisation kann auch innerhalb einer Applikation angewandt werden. STRATO besitzt zur Zeit (Version 1.0) folgende Toplevel Packages.

```
com.strato
       "      .framework                  // Alle globalen Interfaces & Klassen
       "      .lib                        // Vordefinierte Implementationen
       "      .Xui                        // Alles zur Benutzerinteraktion (X ∈ {c,g})
       "      .tests                      // Eine hoffentlich bald "riesige" Test-Bibliothek
```

Ihre Struktur ist dahingehend optimiert, dass je nach Art der Erweiterung am Tool nur auf ein- zwei Klassenpakete (und eventuell deren Unterpakete) zugegriffen werden muss. So hat beispielsweise die gesamte GUI (Graphisches User Interface) einen eigenen Package-„Ast“. Wir wollen nun die Implementation Package für Package inspizieren, dabei aber, wie

[22] Zitat von Kent Beck, dem Erfinder von XP
[23] Software zur Erstellung von Fallbeispielen, Abläufen und ähnlichem in Form von Diagrammen, welche beim Benutzen der Software auftreten.
[24] Beispielsweise die Verhinderung von Namenskonflikten mit Klassen aus der Standardbibliothek.
[25] Siehe Coding Convention Richtlinien unter http://java.sun.com/docs/codeconv/.

eingangs erwähnt, nur die interessantesten Aspekte näher erläutern, um die Dimensionen dieses Dokumentes in Grenzen zu halten.

5.2 Der Kern des Frameworks

5.2.1 Strings und Alphabete

Bereits in Kapitel 4 definiert, bilden Strings die grundlegende Einheit dieser Diplomarbeit. Es gilt daher, eine geeignete Repräsentation für Strings zu finden, die sämtliche Definitionen aus Kapitel 4.2 erfüllt.

Beinahe jede höhere Programmiersprache bietet heutzutage eine Möglichkeit zur Verwaltung von unveränderlichen (engl. immutable) Strings. Die Strings können dabei auf eine beliebiges Gerät (Bildschirm, Drucker) ausgegeben werden. Obendrein werden sie meist noch mit numerischen Werten oder anderen Strings kombiniert. Je nach Programmiersprache dienen dazu spezielle Stringmanipulationsfunktionen (strcpy, strcat) oder gar ein Operator für das Zusammenhängen (engl. concat) zweier Strings wird unterstützt. Der Typ String gehört trotzdem nicht zum Basisset der gängigen Typen (Integer, Boolean, Float, ...) einer jeder Programmiersprache, wie wir am Beispiel von SML sehen. Die Repräsentation im Speicher variiert von Fall zu Fall: C/C++ kennt ein char Array, welches den String durch ein spezielles Nullzeichen terminiert. Java dagegen verwaltet einen String gar als eigenes Objekt der Klasse *java.lang.String*. Jede Methode hat dabei ihre Vor- und Nachteile, je nach den Operationen, die der Benutzer auf den Strings ausführen will.

So läge es in unserem Fall auch nahe, die vordefinierte Klasse *java.lang.String* für unsere Zwecke zu verwenden. Die gängigsten Methoden wie das Verketten zweier Strings, das Extrahieren eines Teilstrings, eines Prä- oder Suffixes, sowie der Test auf das Vorhandensein eines Teilstrings sind standardmässig vorhanden. Ergänzt wird dies durch eine Reihe von Funktionen, welche die restlichen Basistypen in Strings umwandeln oder umgekehrt.

Doch all dies nützt nichts, wenn wir den String beispielsweise Schritt für Schritt, Zeichen für Zeichen erzeugen wollen. Da sind solche unveränderlichen Strings unbrauchbar. Denn beispielsweise jedes Aneinanderketten zweier Strings erzeugt wieder ein neues Stringobjekt – alloziert neuen Speicher und ist daher nicht gerade performant. Daher käme sicher eher die etwas dynamischere Variante des Strings: *java.lang.StringBuffer* in Frage. Bei dieser Klasse wird intern ein Array von Zeichen/Buchstaben angelegt, welches bei Bedarf in der Länge wächst. *Append*, *insert* und *replace* gehören zum Angebot der exportierten Methoden. Doch auch diese Funktionalität reicht noch nicht aus, um all unsere Anforderungen zu erfüllen. Wie repräsentieren wir beispielsweise einen String über einem speziellen Alphabet?

Vererbung (engl. Subclassing) scheint hier eine mögliche Lösung darzustellen. Eine abgeleitete Klasse *CExtendedStringBuffer* (oder so ähnlich), die einen Pointer auf ein Alphabet besitzt und für jeden Buchstaben beispielsweise noch das Gewicht vermerkt, reicht dabei nicht aus. Sämtliche Methoden, die den String in irgendwelcher Weise manipulieren, müssten überschrieben werden. Jede Manipulation muss durch eine prüfende Instanz auf ihre Korrektheit hin untersucht werden (Ist neu eingefügtes Zeichen $c \in \Sigma$?). Nur so würde sichergestellt, dass sich der String ausschliesslich aus erlaubten Zeichen zusammensetzt. Dies ist keine undurchführbare Geschichte, stellt aber eine naive Datenstruktur dar. Es wird nicht nur viel zu viel abgespeichert (das Gewicht eines Zeichens einmal zu speichern würde genügen), sondern es müssen bei komplizierteren Manipulationen wie dem Umkehren eines Strings auch noch ein allfälliges Gewichtsarray und all die andern Daten mit umkopiert werden.

Aus all diesen Gründen muss das Alphabet ausgelagert werden und darf nicht Bestandteil des Strings sein. Selbst Alternativen zur Vererbung, wie die *Komposition*[26] zweier Objekte, helfen hier nicht weiter.

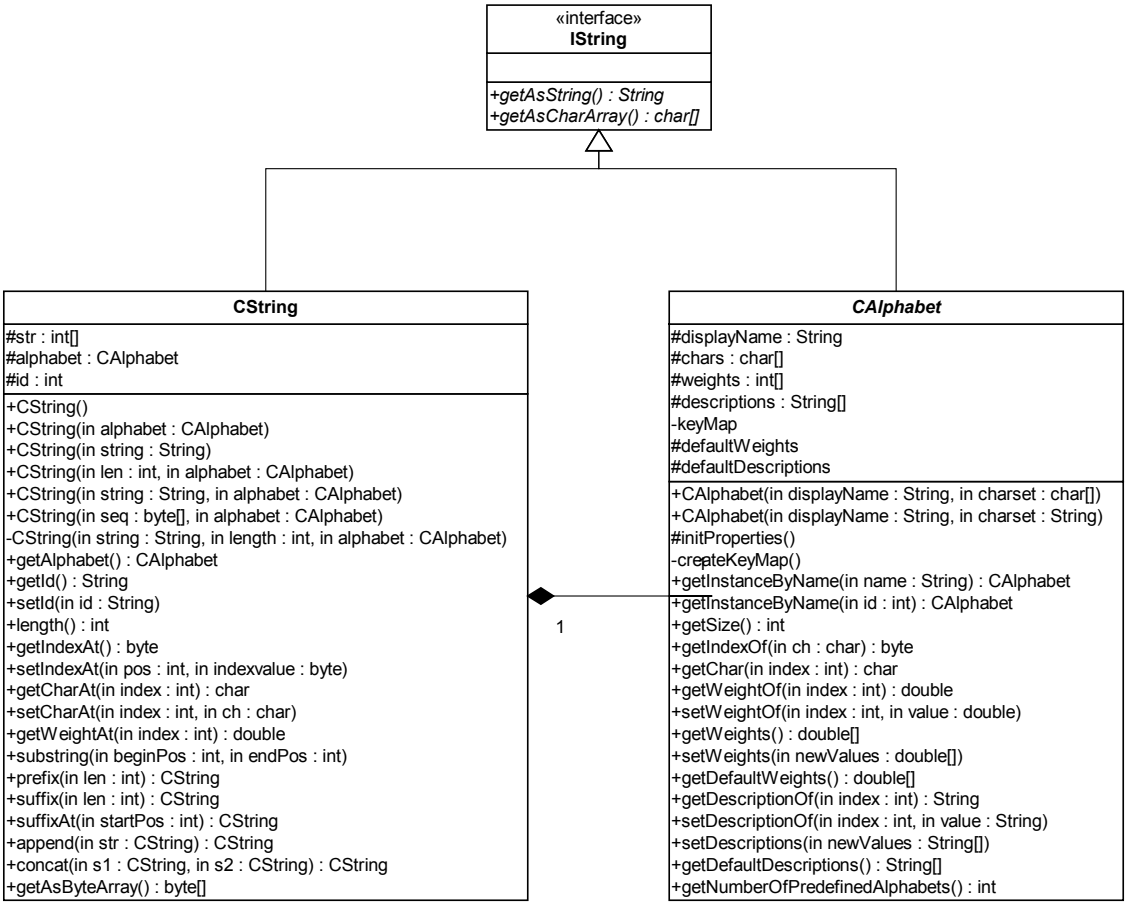

Abbildung 5-1: UML Diagramm der Kernkomponenten CString, CAlphabet, IString

Ausgehend von der Definition in Kapitel 4.2 kreieren wir eine abstrakte Klasse *CAlphabet* (siehe Abbildung 5-1). Sie stellt das gesamte Interface im Umgang mit Strings zur Verfügung. Um schliesslich eine konkrete Instanz zu erzeugen, wird diese von einer Subklasse von *CAlphabet* kreiert. Zusätzlich benötigte Daten wie der verwendete Zeichenumfang werden bei der Erzeugung dem Konstruktor als Parameter mitgegeben oder durch spezielle Methoden gesetzt. Der Tatsache zufolge, dass beispielsweise das Alphabet der arabischen Ziffern immer aus den Zeichen ’0’..’9’ besteht, verwenden wir das Singleton Design Pattern [2]. Dies stellt sicher, dass es im System genau eine Instanz eines konkreten Alphabetes gibt, welche von allen darauf basierenden Strings referenziert wird. Dazu deklarieren wir den Konstruktor von *CAlphabet* als nicht-*public* und bieten eine Methode an, mit der die gewünschte Instanz eines Alphabetes per Namen angefordert wird. Die Klasse *CAlphabet* agiert daher zusätzlich wie eine Fabrik (Factory Design Pattern [2]), die gemäss dem Singleton Pattern aber pro konkrete Alphabet-Klasse nur eine Instanz produziert und immer die Referenz auf diese eine Instanz zurückliefert. Die Instanzen werden in einer *Hashtable* verwaltet deren Key durch den

[26] Bei der Komposition wird im Gegensatz zur Vererbung die neue Klasse B nicht von einer bestehenden Klasse A abgeleitet, sondern die Klasse B erhält eine Referenz auf eine Instanz von A. Wird nun eine Anfrage an die Instanz der Klasse B gestellt, die beim Subclassing durch eine der geerbten Methoden (aus Klasse A) ausgeführt würde, wird die Aufgabe in diesem Fall an die referenzierte Instanz der Klasse A delegiert.

Klassennamen vom Typ *java.lang.String* repräsentiert wird. Wird nun eine Instanz angefordert, wird zuerst überprüft, ob das Objekt nicht bereits vorliegt. Durch dieses Vorgehen sind immer nur die zur Zeit oder früher einmal benötigten Alphabete im Speicher geladen, was sehr speichersparend ist.

Wir erklären die Implementation vom *CAlphabet* in ein-zwei Sätzen und verweisen auf den Sourcecode im Anhang (respektive auf der CD-ROM). Die das Alphabet deklarierenden Zeichen werden in einem Array von Characters gespeichert. Index 0 dieses Arrays wird durch das JOKER-Zeichen repräsentiert. Indizies 1..N durch das Characterset des jeweiligen Alphabetes. Im Falle des binären Alphabetes sind dies die UNICODE-Zeichen 'a' und 'b'. Der Umstand, dass es eine ein-eindeutige Zuordnung zwischen Zeichen und Indizes gibt, erlaubt es in Preprocessing Schritten von diversen Algorithmen, die auf Zeichen/Buchstaben basierenden Daten ebenfalls in Arrays abzulegen und per einfachen Indexzugriff zu verwalten.

Nachdem wir nun eine Implementation eines Alphabetes besitzen, ist die Umsetzung einer neuen Stringklasse *CString* schnell geschehen. Die Klasse besitzt ein Array von Bytes[27] und eine Referenz auf ein Alphabet. Die Länge des Arrays bestimmt die Stringlänge und eingetragene Elemente sind Indexverweise auf das Zeichen-Array des zugeteilten Alphabetes. Die Implementierung ist ein 1:1 Umsetzung der im vorangegangen Kapiteln aufgeführten Definitionen und Implementationshinweisen. Die vorhin erwähnte Konkatenation zweier Strings können wir mit Hilfe der Methode *CString.concat* vornehmen. Sind die Alphabete der beiden Strings allerdings unterschiedlich, wird von *concat* der Wert *null* (anstelle der neuen *CString* Instanz) zurückgeliefert! Will der Programmierer trotzdem zwei „Alphabets-fremde" Strings zusammenhängen, bleibt immer noch die Variante, auf die Grundfunktionalität von Java Strings zurückzugreifen. Wir konvertieren dazu unsere *CStrings* in *java.lang.String* Objekte, führen die Operationen aus und generieren aus dem Resultat einen neuen *CString*. Dies ist ein Umweg mit Performanceeinbussen, wobei der Programmierer auch noch das Problem der unterschiedlichen Alphabete selbst lösen muss. Die Konvertierung zwischen diesen beiden Klassen *java.lang.String* und *CString* ist jedoch immer gewährleistet. Denn wer sich nämlich nicht damit anfreunden kann, in den vom ihm programmierten Tests den Eingabestring und das Eingabealphabet als *CString* respektive *CAlphabet* Objekte zu verwenden, kann den aktuellen Inhalt immer noch in Form des Java Standard Strings *java.lang.String* oder eines Character Arrays *char[]* beziehen. Dies geschieht dadurch, dass beide Klassen zusätzlich noch das *IString* Interface implementieren (siehe Listing 1). Die Konstruktoren von *CString* und *CAlphabet* akzeptieren ihrerseits einen *java.lang.String* als Parameter, was wiederum die Konvertierung in die andere Richtung sicherstellt.

```
public interface IString {
  public String getAsString();
  public char[] getAsCharArray();
}
```

Listing 1: Interface IString

5.2.2 Datenaustausch

Im selben Package wie *CString* und *CAlphabet* befinden sich noch die Interfaces für den Datenaustausch zwischen den verschiedenen Schichten. Aus der Sicht des Datenflusses besteht das Tool aus drei Grundkomponenten (siehe Architektur in Abbildung 4-9) und einer Kontrollinstanz. Führen wir uns den Datenfluss zwischen den einzelnen Komponenten unseres Projektes einmal an Abbildung 5-2 vor Augen. Die „Lollipop-ähnlichen" Kreise

[27] Die Grösse eines Alphabete ist somit auf maximal 256 verschiedene Zeichen beschränkt.

stellen das Interface der jeweiligen Komponente dar, während die dünne, schwarzen Pfeile den Zugriff darauf signalisieren. Die grauen Pfeile hingegen zeigen den Datenfluss selbst.

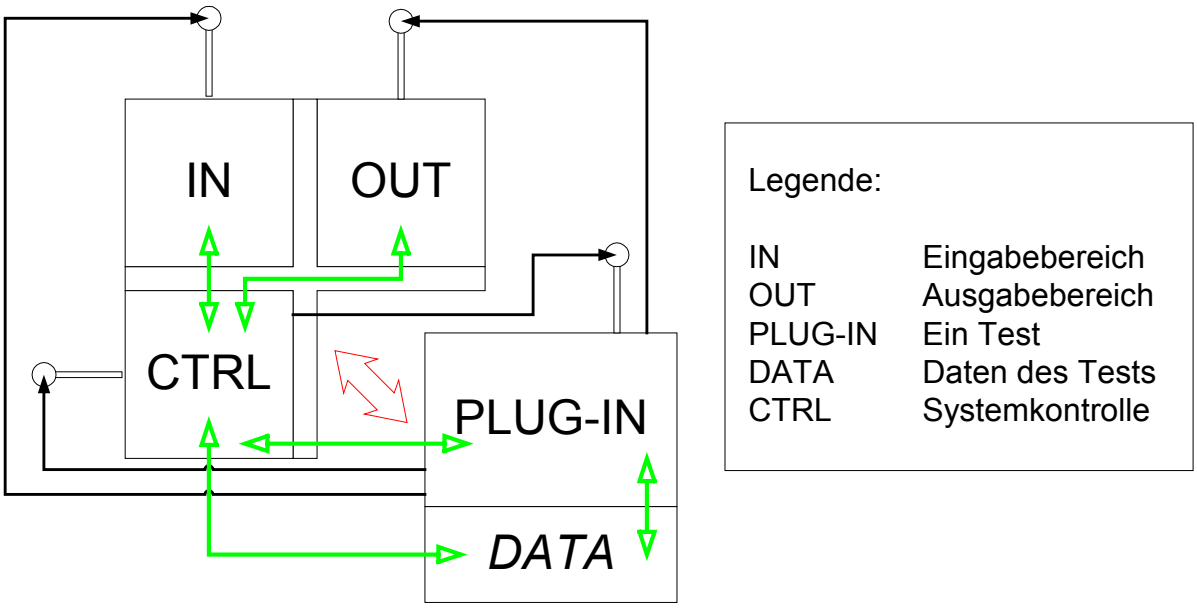

Abbildung 5-2: Der Datenfluss zwischen den einzelnen Programmkomponenten

Am einfachsten betrachtet man die Situation aus der Sicht des Plug-ins. Dieses benötigt einen bestimmten Input und generiert einen spezifischen Output. Zudem wird der ganze Ablauf durch den Benutzer gesteuert, der seine Befehle der Kontrolleinheit CTRL übergibt, welche, nebst anderen Aufgaben, einen Test startet.

Auf Grund der Abbildung 5-2 müsste das Plug-in mindestens drei Referenzen auf die einzelnen Interfaces von IN, OUT und CTRL haben. Da dies zu einer riesigen Parameterflut (beim Konstruktorenaufruf) führt, haben wir den Zugriff auf die einzelnen Interfaces via einem weiteren zusätzlichen Interface, *IAccess* genannt, realisiert (siehe Listing 2).

```
public interface IAccess {
  public IInput getInputs();
  public IOutput getOutputs();
}

public interface IInput {
  public abstract CString getInputString();
}

public interface IOutput {
  public java.io.PrintWriter Console();
  public void setStatus(String text);
  public void setProgress(int value);
}
```

Listing 2: Interfaces IAccess, IInput und IOutput

Das Interface *IInput* bietet zur Zeit bloss den Zugriff auf den in Kapitel 4.2 erklärten Eingabestring. Dabei ist stets zu beachten, das diese Funktion pro Testdurchlauf nur einmal aufgerufen werden sollte, da wir als Testprogrammierer nicht wissen können, aus welcher Quelle die Eingabe stammt. Ist diese ein File oder Editierfenster, so wird diese Funktion stets den selben String zurückgeben – durch erneutes Auslesen des Files respektive Auslesen der Editierfensterbuffers. Wird der Eingabestring aber beispielsweise erst beim Aufruf der

Funktion *getInputString()* vollständig generiert, könnte es sein, das bei einer beigemischten Zufallskomponente problemlos jedes Mal ein anderer Inputstring geliefert wird. Dieses Verhalten könnte beispielsweise durch ein Flag[28] unterbunden werden, wurde aber aus Gründen der Erweiterbarkeit nicht eingebaut. Eine Testbox – sprich ein Test – kann die Eingabe zur weiteren Verwendung selbst zwischenspeichern. Wenn wir das Verhalten der Testbox mit einem IC-Chip vergleichen, können wir auch nicht davon ausgehen, dass bei einem IC am Input-Pin noch dieselbe Spannung anliegt wie beim vorherigen Auslesen. Dass das Interface ansonsten keine Methoden besitzt, ist daraus zu schliessen, dass sämtliche wichtigen Informationen, die den Eingabestring betreffen, Bestandteil des durch die Methode *getInputString()* zurückgelieferten Objektes (der Klasse *CString*) sind.

Das Interface *IOutput* hat im Vergleich zu *IInput* die Aufgabe, dem Test alle nötige Funktionalität zur Verfügung zu stellen, sodass der Test während der Ausführung *aktiv*[29] Informationen an den Benutzer liefern kann. Dazu zählt nebst Schreibzugriffen auf eine Statuszeile oder den Fortschrittbalken die Ausgabemöglichkeit auf eine Konsole. Die Konsole wurde weniger für Debugging Zwecke, als vielmehr für die Ausgabe des Resultates in Textform konzipiert. Aus diesem Grund wird der an die Konsole geschickte Output soweit als möglich (bis zu internem Bufferoverflow) *gecacht* und erst nach Beendigung des Tests ins entsprechende Textfeld geschrieben. So können auch Laufzeitanalysen von Tests gemacht werden, ohne dass sie durch „langsame“ graphische Controls der Benutzeroberfläche gebremst werden.

Bei den aktiven Inhalten angelangt, verweisen wir auch kurz auf die passiven. Unter *passiv* verstehen wir in diesem Kontext Daten, die der Test zwar berechnet aber nicht ohne die explizite Aufforderung an das System liefert. Nehmen wir als Beispiel einen Statistik-Test, der als Antwort einen Vektor von Auftretenshäufigkeiten der einzelnen Buchstaben liefert. Vielleicht lassen sich die Daten in irgendeiner graphischen Form wiedergeben. Falls der Test diese Art Darstellung oder tatsächliche Zusatzinformationen wie den berechneten Mittelwert oder die Standardabweichung bereitstellt, werden diese durch einen Callback[30] durch das System abgefragt. Mehr Information zu diesem Callbackmechanismus finden sich in Kapitel 5.4 „Das Schreiben von Tests“.

5.2.3 Streams

In Kapitel 4.2.5 wurde das Dateiformat für einen einzelnen String bereits eingeführt (siehe Abbildung 4-3). An dieser Stelle wollen wir uns ganz allgemein damit befassen, wie die Daten abgespeichert werden können. Das Tutorial in Kapitel 6 erläutert abschliessend, welche Daten des Tools abgespeichert oder anderweitig persistent gemacht werden können – beispielsweise durch einen Ausdruck (engl. printout).

Die in Abbildung 4-9 hergeleitete Architektur sieht die verschiedenen Quellen vor, für welche es eine Möglichkeit geben soll, diese zusätzlich zum berechneten Resultat abzuspeichern. Dies verlangt ein Speicherungsmodell, welches über das gesamte Framework auf der gleichen Basis funktioniert.

Wir haben uns entschieden, dass die verschiedenen Datenquellen und das Resultat sowohl einzeln, wie auch als Komplettpaket (siehe Kapitel 4.4) abgespeichert werden können. Dies

[28] Das Flag (ein Marker) würde den Systemzustand festhalten. So könnte festgestellt werden, ob *getInputString()* zweimal vom selben Test aufgerufen wurde.
[29] Aktiv meint in diesem Zusammenhang, dass ein Test die Ausgabendaten selbstständig (und nicht erst auf Anfrage = passiv) an das System liefert.
[30] Callback meint den Aufruf einer vorgegebenen Methode, deren Funktionalität durch den Benutzer vorgegeben wird.

bringt den Vorteil, die Daten je nach Bedarf abspeichern zu können, ohne dass immer alles gesichert werden muss. Insbesondere bei der Visualisierung der Resultate ist es wichtig, dass auch diese Art der Darstellungen eines Testergebnisses (beispielsweise in Form einer Grafikdatei) separat abgespeichert werden kann. Nachteile dieser Vorgehensweise sind uns bis anhin nicht bekannt.

Zuerst definieren wir durch das Interface IPersistence, welche Funktionalität wir in diesem Kontext anbieten wollen. Die Schnittstelle kennt vier Methoden (siehe Listing 3):

```
public interface IPersistence {
  public void clear();
  public void open();
  public void save();
  public void print();
}
```

Listing 3: Interface IPersistence

Fangen wir bei der einfachsten Methode *clear* an. Da unsere Daten sich in der Regel in einem graphischen Element (zum Beispiel einem Textfeld) befinden, löschen wir hierbei deren Inhalt. Sollten sich die Daten stattdessen in einem File befinden, würde dieses gelöscht.

Etwas schwieriger sieht es mit dem Laden und Speichern von Daten aus. Wir betrachten im folgenden nur das Speichern und sehen das Laden als Umkehrfunktion dieser, die auf dem selben Prinzip basiert – bloss in der anderen Richtung. Die Untersuchung des *java.io.packages* bietet dem Benutzer eine Vielzahl an Klassen für den Zugriff auf Files. Die Dateien können als Randomfile[31] angesprochen werden, als sequentielle Binär- oder Textdateien. Obwohl ein eigenes Binärformat die flexibelste Variante darstellt, haben wir uns bereits in der Konzeptphase für ein einfaches Textformat[32] bei der Abspeicherung von erzeugten Strings festgelegt und wollen dies nun auf das ganze Projekt ausweiten. Durch diesen Beschluss treffen wir implizit auch eine weitere Entscheidung, die Daten sequentiell, d.h. Zeile für Zeile, abzuspeichern. Doch sobald man von sequentiellem Abspeichern oder eben Auslesen redet, findet man sich schnell im Reich der *Streams* wieder. Ein Stream erlaubt es, Daten in einer vom Programmierer festgelegten Reihenfolge einzuspeisen und diese später in der selben Reihenfolge wieder auszulesen. Dabei spielt es keine Rolle, ob der Stream die Daten in eine Datei leitet (und sie später wieder von dort holt), via Netzwerk an einen andern Computer überträgt (zum Beispiel via TCP/IP-Protokoll im Internet) oder bloss in einen dafür vorgesehenen Speicherbereich des eigenen Hauptspeichers (RAM) leitet. Diese Flexibilität von Streams wurde in der Standardbibliothek von Java konsequent implementiert und stellt meines Erachtens den einfachsten Umgang mit sequentiell verfügbaren Daten dar. Das Prinzip eines Streams kann etwa mit dem Förderband in einer Fabrik verglichen werden. Am einen Ende werden Produkte nacheinander auf das Förderband gelegt und am andern wieder in der selben Reihenfolge entnommen. Dabei kann auf dem (langen) Weg, welcher das Produkt zurücklegt, jeder am Förderband tätige Arbeiter etwas mit dem Produkt anstellen (Waschen, Einpacken, ...) und es sogleich wieder zurücklegen. Im Bereich der Streams wären diese Tätigkeiten allerdings Aufgaben wie Verschlüsseln, Komprimieren oder den Inhalt in eine andere Form bringen (zum Beispiel binäre Daten in Textform in einer ASCII-Datei speichern).

Nachdem wir uns für Streams entschieden haben, brauchen wir allerdings noch ein Interface, das durch die jeweiligen Klienten, welche etwas auf den Stream schreiben oder etwas davon

[31] In einem solchen File kann, mittels Zeiger, beliebige Byteschritte weit vor und zurück navigiert werden.
[32] Heutzutage wäre beispielsweise XML ein mögliche Alternative, wenn es darum ginge, einen offiziellen Standard für die Repräsentation der Daten zu verwenden.

lesen wollen, implementiert werden muss. Wohin die Daten (der Stream) dann geleitet werden (wird), spielt für den Klienten keine Rolle. Diese Konvention findet sich im Interface *IStreamable* (siehe Listing 4).

```
public interface IStreamable {
  public void loadFromStream(InputStream ins) throws Exception;
  public void saveToStream(OutputStream outs) throws Exception;
}
```

Listing 4: Interface IStreamable

Die beiden Methoden besitzen jeweils eine Referenz auf einen Stream, welche zum „Schreiben auf“ bzw. „Lesen von“ verwendet wird. Dabei handelt es sich um die sehr generischen Klassen *java.io.OutputStream* und *java.io.InputStream*, deren dynamischer Typ, wie oben erwähnt, vielfältige Ausprägungen besitzen kann.

Ohne jetzt alle Klassen aufzulisten, welche dieses Interface implementieren, betrachte man die konkrete Implementation *CStringStreamable* im Anhang dieser Arbeit. Es handelt sich um die Verwaltung der Eingabestrings, die wir schon in Kapitel 4.2.5 theoretisch untersucht haben.

Die letzte der vier Methoden des Interfaces *IPersistence* befasst sich mit dem Abspeichern auf Papier – d.h. mit dem Drucken. Diese Art der Speicherung ist in Java relativ kompliziert. Da die Implementierung nicht zwingend war, wird an dieser Stelle nur auf entsprechenden Web-Seiten[33] verwiesen, die in Online-Tutorials zur „Java Printing API“ alle dazu nötigen Informationen beherbergen. Darin erwähnte Vorgehensweisen wurden weitgehend umgesetzt und können dort Schritt für Schritt nachgelesen werden.

5.2.4 Stringerzeugung nach festen Regeln

Abschliessend für dieses Unterkapitel 5.2 gilt es noch die Klasse *CRule* zu erwähnen. Sie bildet die Superklasse aller Arten von regelbasierter Stringerzeugung (siehe Kapitel 4.2.4). Diese abstrakte Klasse besitzt eine einfache Konvertierungsmethode *extractParametersFromString(String argstring)* für Eingabeparameter in „Stringform“. Je nach Notation (siehe auch Tabelle 6-2) werden die einzelnen Teilstrings als Elemente eines speziellen Typs erkannt und deren Objektinstanz im Resultatvektor zurückgeliefert. Der String *5,“hallo“,1.354d,true* wird in einen vierdimensionalen Vektor umgewandelt, deren Komponenten die Typen *int, String, double und boolean* aufweisen.

Jede Subklasse überschreibt im weiteren zumindest die Methode *createString(String params, int length, CAlphabet alphabet)* welche ein nach Regeln erzeugtes *CString* Objekt zurückliefert. Die Implementation, wie eine spezielle Regel programmiert wird, ist vollständig dem Programmierer der SEM überlassen und ist daher nicht Bestandteil dieses Textes. Ein paar vorhandene Beispiel-Klassen zeigen die korrekte Verwendung und sollen als Anleitung zur Implementierung weiterer Arten dienen.

5.3 Die PLUG-IN Schnittstelle

Eine wichtige Ressource, die zum erweiterten Kern des Frameworks zählt stellt das *com.strato.framework.test Package* dar. Bereits in Kapitel 4 wurde die verwendete

[33] Beispiele sind: http://java.sun.com/products/java-media/2D/forDevelopers/sdk12print.html
http://developer.java.sun.com/developer/onlineTraining/Programming/JDCBook/render.html
http://developer.java.sun.com/developer/onlineTraining/Programming/JDCBook/advprint.html
http://java.sun.com/j2se/1.4/docs/guide/jps/spec/jpsTitle.fm.html

Architektur für den Verarbeitungsschritt (Blackbox/Testbox) beschrieben. Doch wie vorgängig bereits erwähnt, ist die Implementation einer Plug-in API (*Application Programming Interface*) eine Gratwanderung zwischen Über- und Unterdimensionierung. Während die Überdimensionierung „bloss" sehr viel Aufwand für einfache Aufgaben mit sich bringt, kann eine Unterdimensionierung schon bald den Flaschenhals des ganzen Systems darstellen – insbesondere was deren Erweiterbarkeit betrifft. Nun wollen wir versuchen, unsere allgemeinen Ideen und Ansätze einer Plug-in Schnittstelle, wie wir sie in Kapitel 4 hergeleitet haben, bestmöglich umzusetzen.

5.3.1 Deskriptoren

Ein erster Ansatz besteht darin, dass wir das Interface (d.h. die Signaturen) einer (sehr) grossen Anzahl von Methoden/Prozeduren vorgeben, die jedes Plug-in implementieren muss. Dies ist heutzutage eine weit verbreitete Art, die Erweiterbarkeit eines Tools zu ermöglichen. Die Vorteile dieser Vorgehensweise liegen auf der Hand. Es ist genau definiert, was in jeder Methode geschehen soll (das Neuzeichnen des Objekt, eine Berechnung starten, den Zustand zurücksetzen, usw.). Das neu hinzugefügte Modul muss schliesslich noch in ein spezielles Unterverzeichnis des zugehörigen Programms kopiert und/oder irgendwo registriert werden. Fertig!

Doch was in erster Linie als die sauberste Lösung aussieht, zwängt den Programmierer leider in ein sehr enges Schema. Ist irgendeine Funktionalität nicht vorgesehen, kann diese nur durch das Hinzufügen eines zweiten Interfaces korrigiert werden (siehe Versionierungsprinzip bei der Microsofts™ Komponententechnologie COM™). Die alte, erste Schnittstelle muss aber weiterhin unterstützt werden, um abwärtskompatibel zu bleiben. Im weiteren wird dem Programmierer auch jegliche Freiheit bezüglich Programmierstil (d.h. Nomenklatur seiner Methoden, Variablen, ...) genommen. Wird bloss eine einfache Funktionalität benötigt, müssen trotzdem alle Interfaces vollständig ausprogrammiert werden, um die korrekte Arbeitsweise zu garantieren.

Ein zweiter Ansatz geht in eine ganz andere Richtung. Gehen wir einmal davon aus, dass wir sämtlichen Code unseres Plug-ins in einer einzigen Klasse zur Verfügung stellen wollen. Wir codieren demzufolge unsere Methoden und halten uns dabei an keinerlei Richtlinien. Ohne auch nur die geringste Information zu publizieren, was unsere Klasse zu leisten vermag, veranlassen wir das System, sich diese Auskunft automatisch zu extrahieren – nämlich aus dem Code selbst. Diese Technik ist bekannt unter dem Namen Reflexion (engl. Reflection). Moderne Programmiersprachen wie Java oder Microsoft™ C# bieten standardmässig eine Bibliothek von Klassen an (das sog. Reflection-API), welche es mittels eines Introspektions-Mechanismus erlaubt, den internen Aufbau einer Klasse zu durchleuchten. Dabei ist es möglich, Information über Methoden, Felder und Konstruktoren einer Klasse abzufragen. Gefundene Methoden können schliesslich zur Laufzeit ausgeführt werden, als hätten wir diese „hardcoded"[34] im Sourcefile. Dies geschieht dadurch, dass man den im Sourcecode verwendeten Namen einer Prozedur, Funktion, Variablen oder Konstanten einer speziellen Methode als Parameter des Typs *java.lang.String* übergibt. Dieser String kann könnte ebenso gut vom Benutzer eingetippt worden sein. Zur Veranschaulichung dieser mächtigen Technik siehe folgendes „Hello ETH" Programm.

[34] Hardcoded meint das Ausprogrammieren einer Funktionalität direkt im Sourcefile. Beispielsweise alle Konstanten einer Applikation werden direkt in den Quellcode (mit den zugehörigen Werten) als *static final "Klasse" "Name" = Wert;* verankert.

```
import java.lang.reflect.* ;         // Reflection API

public class Test {
  public static final String MY_STR = "ETH";

  public String foo(String str) {
    return "Hello " + str;
  }

  public static void main(args[] String) {
    Test t = new Test();
    String res = null;
}
    // Konventioneller Aufruf der Methode foo(…)und Ausgabe auf Standard-Out
    res = t.foo(MY_STR);
    System.out.println(res);

    // Aufruf der Methode mittels Reflexion
    try {
      Method m = t.getClass().getMethod("foo", new Class[]{String.class});
      int res = m.invoke(t, new Object[]{ MY_STR}) ;
      System.out.println(res);
    }
    catch (Exception e) {
      System.err.println("Error occurred!!!");
    }
  }
} // eof class Test
```

Listing 5: Beispielprogramm zur Reflection API

Diese Art von Introspektion wird beispielsweise bei Java-Beans™ (ebenfalls eine Komponententechnologie) stark ausgereizt. Sämtliche Eigenschaften (engl. Properties) einer Komponente sind dabei als Membervariablen zu deklarieren (vorzugsweise mit eingeschränktem Zugriff durch einen *private/protected Modifier*). Um von aussen kontrolliert (und trotz etwaigen Modifiern) auf sie zugreifen zu können, müssen zusätzlich eine Getter- und eine Setter-Methode zur Verfügung gestellt werden. Java geht hier den direkten Weg, dass die Methodennamen aus dem Namen der Variablen konstruiert werden können, indem einfach ein *get* resp. ein *set* vor den Namen gestellt wird und der erste Buchstabe des Elementnamens in einen Grossbuchstaben umgewandelt wird. Für ein Element des Typs *int* mit Namen *count* müssen demzufolge zusätzlich zwei Methoden *void int getCount()* und *void setCount(int value)* existieren, welche es erlauben, einen Wert auszulesen bzw. ihn zu setzen. Auch diese Vorgabe zwängt den Programmierer in einen unnötigen Formalismus, die mit einer dritten Methode elegant umgangen werden kann: Die Idee eines Deskriptors.

Die Deskriptorenvariante stellt eine Mischung aus den beiden vorhergehenden dar. Anstelle der Wühlmaustechnik, wobei das System in der Klasse blind nach vorhandenen Methoden (eventuell mit speziellen Namen) sucht, wird die angebotene Funktionalität über sog. Deskriptoren abgefragt. Ein Deskriptor beschreibt im Wesentlichen, wie die angebotenen Dienste (engl. *Services*) heissen, was sie tun und welche Parameter sie verarbeiten können. Abbildung 5-3 zeigt das UML-Diagramm der Deskriptorenschnittstelle.

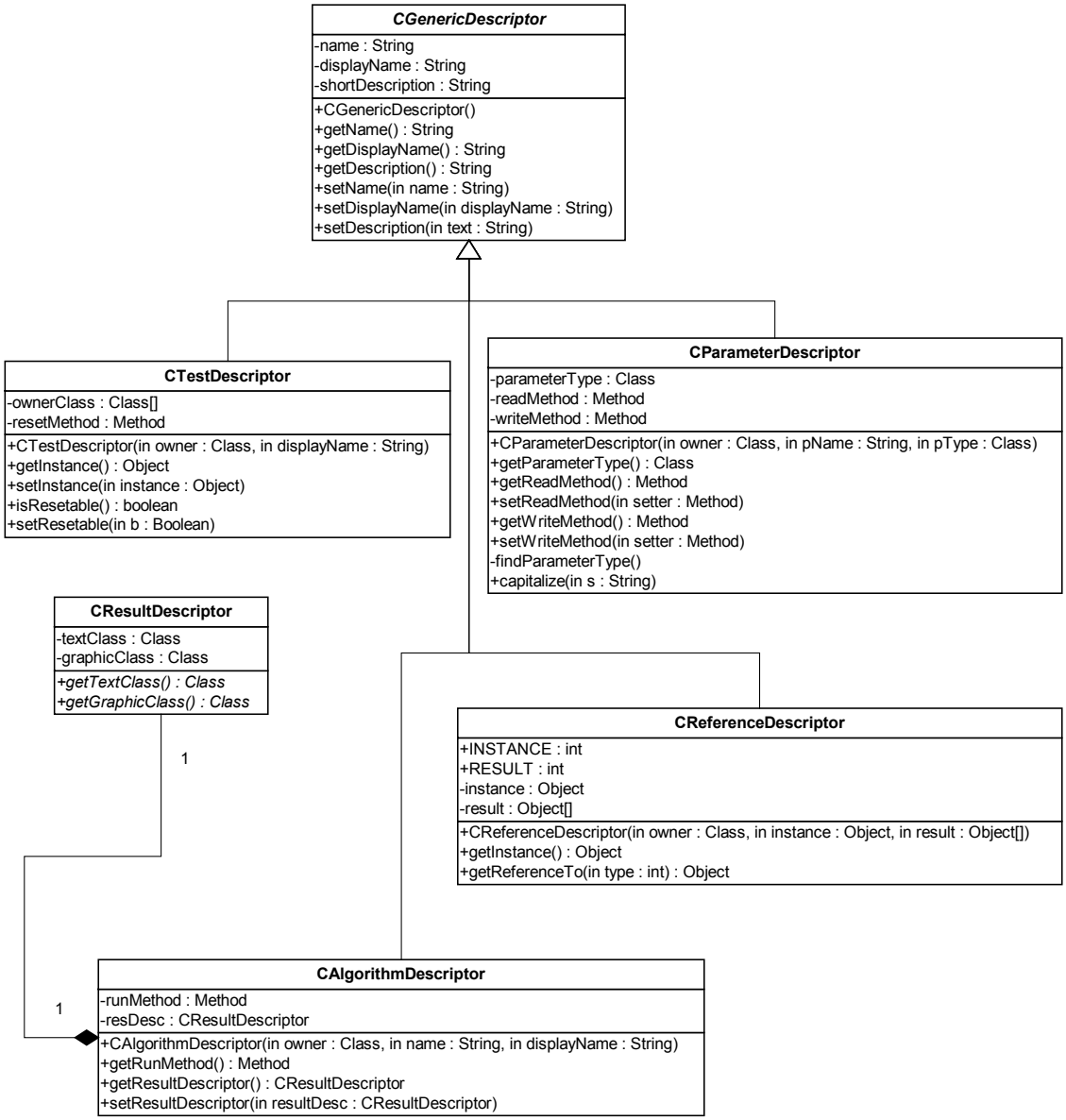

Abbildung 5-3: UML Klassendiagramm der Deskriptoren

Damit wir auf den gewählten Test wirklich nur in einem kontrollierten Masse zugreifen können, implementiert jeder Test (das heisst, die den Test repräsentierende Klasse) das *ITestInfo* Interface. Dem System stehen damit ausschliesslich *CxxxDescriptor* Instanzen zur Verfügung, um auf den Test zuzugreifen (siehe Abbildung 5-4).

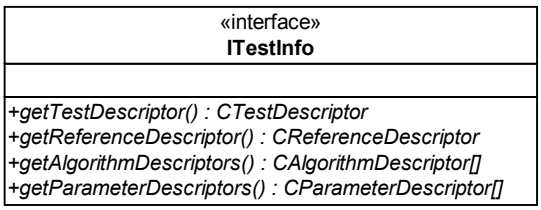

Abbildung 5-4: Interface zur Veröffentlichung der Funktionalität eines Tests

Bleibt noch die Frage zu klären, was denn die einzelnen Deskriptoren beinhalten sollen, d.h. welche Art von Information sie bereitstellen? Jeden einzelnen Deskriptor-Typ im Detail zu

beschreiben ist eigentlich gar nicht nötig, wenn man das Prinzip einmal begriffen hat. Wir wollen uns die zu Grunde liegende Idee am Beispiel des *CParameterDeskriptor*s veranschaulichen. Die Dokumentation der restlichen Deskriptoren folgt im Anhang der Arbeit. Im folgenden Kapitel 5.4 wird Schritt für Schritt ein Test programmiert und es ergibt sich die Möglichkeit, die ganze Sache aus einer praktischen Sichtweise zu verstehen. In diesem Abschnitt wird also vorerst nur der Aufbau eines Deskriptors und nicht dessen Verwendungsweise dargestellt.

```
package com.strato.framework.test;

/*
 * @(#)CParameterDescriptor.java   1.0   2002/03/03
 *
 * Title:        STRing-Analysis-TOols (STRATO)
 * Description:  Diploma Thesis: Structural Properties of Strings
 * Copyright:    Copyright (c) 2001, 2002 by Daniel Emmenegger
 * Company:      ETH Zurich, Switzerland / Department of Computer Science
 * Homepages:    http://www.ethz.ch , http://www.ti.inf.ethz.ch/pw
 *
 * For further information see LICENCE.TXT
 */

import java.lang.reflect.*;
import java.beans.*;

/**
 * A CParameterDescriptor describes one parameter that a STRATO Test
 * exports via a pair of accessor methods.
 *
 * @author Daniel Emmenegger
 * @version 1.0 2002/03/03
 * @see     com.strato.framework.test.CGenericDescriptor
 * @see     com.strato.framework.test.ITestInfo
 * @since   STRATO 1.0
 */
public class CParameterDescriptor extends CGenericDescriptor {

  private Class parameterType;
  private Method readMethod;
  private Method writeMethod;

  /**
   * Constructs a CParameterDescriptor for a parameter that follows
   * the standard Java convention by having getFoo and setFoo
   * accessor methods.  Thus if the argument name is "fred", it will
   * assume that the writer method is "setFred" and the reader method
   * is "getFred" (or "isFred" for a boolean parameter).  Note that the
   * parameter name should start with a lower case character, which will
   * be capitalized in the method names.
   *
   * @param parameterName The programmatic name of the parameter.
   * @param testClass The Class object for the target test.  For
   *           example com.strato.test.MyTest.class.
   * @exception EConventionException if an exception occurs during
   *              introspection.
   */
  public CParameterDescriptor(Class ownerClass, String parameterName, Class parameterType)
              throws EConventionException {
    if (parameterName == null || parameterName.length() == 0) {
          throw new EConventionException("bad parameter name");
    }
    setName(parameterName);
    String base = capitalize(parameterName);

    // Since there can be multiple setter methods but only one getter
    // method, find the getter method first so that you know what the
    // property type is.  For booleans, there can be "is" and "get"
    // methods.  If an "is" method exists, this is the official
    // reader method so look for this one first.
    try {
      readMethod = CConvention.findMethod(ownerClass, "is" + base, 0);
    }
    catch (Exception getterExc) {
```

```
    // no "is" method, so look for a "get" method.
    readMethod = CConvention.findMethod(ownerClass, "get" + base, 0);
  }
  Class params[] = { readMethod.getReturnType() };
  writeMethod = CConvention.findMethod(ownerClass, "set" + base, 1, params);

  findParameterType();
}

/**
 * Gets the Class object for the parameter.
 *
 * @return The Java type info for the property.  Note that
 * the "Class" object may describe a built-in Java type such as "int".
 * The result may be "null" if this is an indexed property that
 * does not support non-indexed access.
 * <p>
 * This is the type that will be returned by the ReadMethod.
 */
public Class getParameterType() {
  return parameterType;
}

/**
 * Gets the method that should be used to read the parameter value.
 *
 * @return The method that should be used to read the property value.
 * May return null if the property can't be read.
   */
public Method getReadMethod() {
  return readMethod;
}

/**
 * Sets the method that should be used to read the parameter value.
 *
 * @param getter The new getter method.
 */
public void setReadMethod(Method setter) throws EConventionException {
  readMethod = setter;
  findParameterType();
}

/**
 * Gets the method that should be used to write the property value.
 *
 * @return The method that should be used to write the property value.
 * May return null if the property can't be written.
 */
public Method getWriteMethod() {
  return writeMethod;
}

/**
 * Sets the method that should be used to write the property value.
 *
 * @param setter The new setter method.
 */
public void setWriteMethod(Method setter) throws EConventionException {
  writeMethod = setter;
  findParameterType();
}


private void findParameterType() throws EConventionException {
  try {
    parameterType = null;
    if (readMethod != null) {
     if (readMethod.getParameterTypes().length != 0) {
        throw new EConventionException("bad read method arg count");
     }
     parameterType = readMethod.getReturnType();
     if (parameterType == Void.TYPE) {
        throw new EConventionException("read method " +
                                  readMethod.getName() + " returns void");
     }
    }
```

```
      if (writeMethod != null) {
        Class params[] = writeMethod.getParameterTypes();
        if (params.length != 1) {
          throw new EConventionException("bad write method arg count");
        }
        if (parameterType != null && parameterType != params[0]) {
           throw new EConventionException("type mismatch between read and write methods");
        }
        parameterType = params[0];
      }
    }
    catch (EConventionException ex) {
      throw ex;
    }
  }

  static String capitalize(String s) {
    if (s.length() == 0) {
      return s;
    }
    char chars[] = s.toCharArray();
    chars[0] = Character.toUpperCase(chars[0]);
    return new String(chars);
  }

} // eof CParameterDescriptor
```

Listing 6: Die Klasse CParameterDescriptor

Die Methode *getParameterDescriptors()* aus dem *ITestInfo* Interface gibt ein Array von *CParameterDeskriptor*en zurück, dessen Elemente den einzelnen Parametern eines Tests entsprechen. Dem Konstruktor eines Objektes wird dabei die zugehörige Klasse (beispielsweise *MyTestClass.class)*, der verwendete Variablename im Sourcefile (beispielsweise *pattern*) und der Typ des Parameters (beispielsweise *java.lang.String*) übergeben. Werden keine weiteren Angaben gemacht, nimmt das System automatisch an, dass dazu eine Getter- und eine Setter-Methode existieren, die nach den obigen Regeln benannt sind. Sollte dies nicht der Wunsch des Programmierers sein, kann er zusätzlich nach dem Erzeugen die Methoden *setReadMethod(...)* oder *setWriteMethod(...)* aufrufen und eigene Namen vorgegeben. Der alles umschliessende *try-catch* Block sorgt dafür, dass nur „Wahrheiten“ zurückgeliefert werden und gewisse wenige Konventionen eingehalten wurden. Genaueres zur Einhaltung von Konventionen folgt gleich im Anschluss an dieses Kapitel.

Diese so erhaltenen Informationen werden vom System ausgewertet, um diesen *pattern* Parameterwert neu zu setzen, nachdem der neue Wert in der GUI eingegeben wurde. Den Code für die Interaktion zwischen System und Test findet man in der Klasse *CCtrlCenter*. Dabei wird ganz genau nach obigem Beispielsource des „Hello ETH Programms“ vorgegangen.

Abschliessend sei noch einmal auf die Hauptvorteile von Deskriptoren hingewiesen. Einerseits kann nun jeder Programmierer seinen bewährten und persönlichen Programmierstil pflegen und andererseits wird verhindert, dass das System sich sämtliche Informationen selbst beschaffen muss. Im weiteren sind keine Daten verfügbar, die nicht explizit für den Export gedacht sind. Der Programmierer hat die volle Kontrolle, was er bei seinem Code als öffentlich (d.h. durch das System zugänglich) deklarieren will und was nicht. Ein paar Klassen, welche vorgegebene Implementationen des Interfaces *ITestInfo* vornehmen, erleichtern das Leben des Testprogrammierers um einen weiteren Faktor, indem er seinen Test von einer dieser Klassen ableite. Siehe stellvertretend dazu die Implementation von *CSimpleTestInfo*.

5.3.2 Sicherheit durch Exceptions

Nebst den Deskriptoren Klassen finden sich im selben Package noch ein paar weitere, die ausschliesslich im Zusammenhang mit dem Schreiben von Tests eine Rolle spielen. Wie im vorhergehenden Unterkapitel erwähnt, werden die Tests mit Hilfe von Deskriptoren beschrieben. Die Vorgabe an den Programmierer, seinen Test in Form eines solchen deskriptiven Verfahrens dem System zu übergeben, beinhaltet einen weiteren Sicherheitsmechanismus. Die konkreten Aussagen über vorhandene Parameter, Algorithmenvarianten und ähnliches kann durch die vorher erwähnte Introspektion erst zur Laufzeit überprüft werden. Sollte ein Tests nun aber Angaben über nicht vorhandene Features bekannt geben, wird eine sogenannte *EConventionException* ausgelöst.

Der zweite Typ von Exceptions heisst *EUpdateException*. Diese kann dazu verwendet werden, dem Benutzer zu signalisieren, dass beispielsweise die angezeigten Daten und die soeben modifizierten Parameterwerte nicht mehr dem neuesten Stand entsprechen. Der Test muss in einem solchen Fall erneut gestartet werden, um korrekte Ergebnisse (graphisch oder textbasiert) für die derzeitig gewählten Eingaben zu erhalten.

Ein letzte spezielle Exceptionklasse *EInvalidUserInputException* erlaubt es dem Testprogrammierer Rückmeldungen an den Benutzer zu liefern, falls der Test mit den derzeitigen Eingabewerten nicht ausführbar ist. Soweit als möglich versucht allerdings auch das Tool selbst (bereits vor Ausführung des Tests) inkorrekt Eingaben zurückzuweisen. Dazu gehören falsche Zeichen im Eingabestring (im Bezug auf das zugeteilte Alphabet) oder eine negative Eingabelänge.

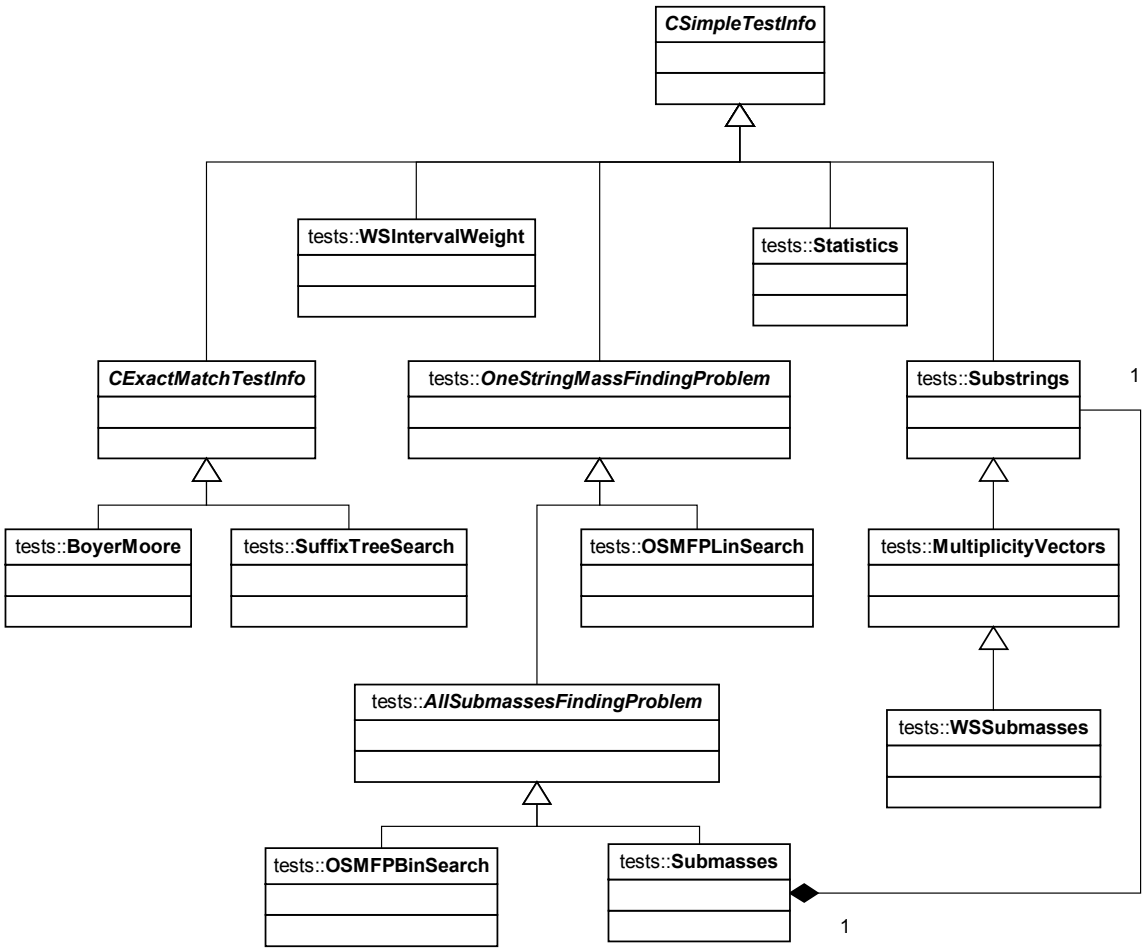

Abbildung 5-5: UML Klassendiagramm aller in STRATO 1.0 verfügbaren Tests

5.4 Das Schreiben von Tests

Wir widmen uns nun dem Schreiben eines eigenen Tests. In diesem Kapitel wird ein „Muster-Test“ entwickelt, der die meisten vorhandenen Features enthält. Es wird dabei alles aus Sicht der Implementation, d.h. des Programmierers, erläutert. Man betrachte aus diesen Gründen die nun folgenden Abschnitte als eine Art *Survival-Guide* zum Schreiben neuer Tests. Eine Übersicht zu allen in Version 1.0 verfügbaren Tests zeigt Abbildung 5-5.

Wir beginnen das Unterkapitel mit den Anforderungen an unseren Muster-Test. Im Anschluss entwickeln wir den dazugehörigen Sourcecode.

Aufgabe:

Schreibe einen Test, der den Algorithmus BINSEARCH für das ONE-STRING MASS FINDING PROBLEM[35] implementiert. Dabei soll der Benutzer als Parameter die gesuchte Teilmasse *m* eingeben können. Der Algorithmus macht ein Preprocessing. Nebst dem eigentlichen Problem

- (1) Existiert die Teilmasse *m* im Eingabestring *s*?

wären weitere Varianten (Fragestellungen) an den Eingabestring *s* denkbar:

- (2) Wie viele verschiedene Teilmassen hat *s*?
- (3) Gib eine Liste aller auftretenden Teilmassen in *s* als Liste/Vektor zurück?

Wir sehen zwar, dass nur in der ersten Variante den Parameter *m* gebraucht wird. Dies ist allerdings kein Problem, denn Varianten (2) und (3) ignorieren *m* einfach.

5.4.1 Das Grundgerüst

Grundsätzlich besteht ein Test aus mindestens einer Javaklasse. Diese bietet das Interface gegenüber dem STRATO System an, welches durch Deskriptoren bekannt gegeben wird.

Wir leiten unsere Klasse *OSMFPBinSearch*[36] von *CSimpleTestInfo* ab, deren Funktionalität sich darauf beschränkt, ein paar Vereinfachungen für den Zugriff auf die I/O-Interfaces zu definieren und das Interface *ITestInfo* mit Standardangaben (Rückgabewert ist meist *null*) zu implementieren. Gleichzeit importieren wir die nötigen Packages und erstellen den Konstruktor.

```
package com.strato.tests;

import com.strato.framework.*;
import com.strato.framework.test.*;
import java.util.*;

/** A test class that implements the BINSEARCH algorithm */
public class OSMFPBinsearch extends CSimpleTestInfo {

  /** Constructor */
  public OSMFPBinsearch(IAccess access) {
    super(access);
  }
  ...
}
```

Listing 7: Beispieltest Teil I: Importklauseln und Konstruktor

[35] Weitere Erklärungen dazu siehe [5]

[36] Wenn wir die Abbildung 5-5 genau betrachten, so ist *OSMPFBinsearch* eine Subklasse von *AlSubmasseFindingProblem*, welche wiederum eine Subklasse von *OneStringMassFindingProblem* ist. Um uns nicht mit der ganzen Vererbungshierarchie beschäftigen zu müssen, präsentieren wir hier eine Variante von *OSMPFBinsearch*, welche die gesamte Funktionalität der drei Klassen in einer vereint.

Nun entwickeln wir die einzelnen Deskriptoren, die wir nicht ohne Änderung von *CSimpleTestInfo* erben können und deren zugehörige *ITestInfo* Methoden wir daher überschreiben müssen.

```
public CTestDescriptor getTestDescriptor() {
  CTestDescriptor test =
    new CTestDescriptor(OSMFPBinsearch.class, "Binsearch {simple}");
  test.setShortDescription("Implements the BINSEARCH algorithm to solve "+
                           "the ONE-MASS-FINDING-PROBLEM");
  return test;
}
```

Listing 8: Beispieltest Teil II: Beschreibung des Tests

Diese Zeilen sagen aus, dass es sich um einen Test handelt in Klasse *OSMFPBinsearch.class* und dieser als *Binsearch {simple}* im Display angezeigt wird. Zusätzlich setzten wir noch die optionale Beschreibung des Tests mittels der *setShortDescription(...)* Methode.

Nun folgen die Parameter. In unserem Fall ist dies lediglich die gesuchte Teilmasse *m*:

```
public CParameterDescriptor[] getParameterDescriptors() {
  try {
    CParameterDescriptor mass =
      new CParameterDescriptor(this.getClass(), "m", Double.TYPE);
    mass.setDisplayName("submass M");
    mass.setShortDescription("The submass we search for");

    CParameterDescriptor[] ret = {mass};
    return ret;

  }
  catch (EConventionException ce) {
    throw new Error(ce.toString());
  }
}
```

Listing 9: Beispieltest Teil III: Die verwendeten Parameter

OSMFPBinsearch.class referenziert auch hier die Klasse, wo sich der Deskiptor befindet. Der im Programm verwendete Variablenname heisst *m* und ist vom Typ *double*. Damit im Display nicht nur „m?“ erscheint, setzten wir den Diplaynamen auf *submass M* und liefern eine weiterführende Erklärung. Der *try-catch*-Block terminiert das System mit einem Error, falls die Membervariable *m* oder die dazugehörigen Zugriffsmethoden fehlen. Diese finden wir in Listing 10)

```
/** parameter: the submass we search for */
protected double m;

/***** do not forget: the getter and setter Methods for the parameters *****/

public double getM() {
  return this.m;
}

public void setM(double m) {
  this.m = m;
}
```

Listing 10: Beispieltest Teil IV: Variable, Setter- und Getter-Methoden

Definieren wir nun die angebotenen Varianten unseres Algorithmus.

```
public CAlgorithmDescriptor[] getAlgorithmDescriptors() {
  try {
    CAlgorithmDescriptor algoExists =
      new CAlgorithmDescriptor(OSMFPBinsearch.class, "massExists",
                                          "Does a given mass m exist?");

    CAlgorithmDescriptor algoFreq =
      new CAlgorithmDescriptor(OSMFPBinsearch.class, "massCount",
                                "How many different submasses exist?");

    CAlgorithmDescriptor algoList =
      new CAlgorithmDescriptor(OSMFPBinsearch.class, "massList",
                                                "List all submasses!");


    CAlgorithmDescriptor[] ret = {algoExists, algoFreq, algoList};

    return ret;
  }
  catch (EConventionException ce) {
    throw new Error(ce.toString());
  }
}
```

Listing 11: Beispieltest Teil V: Angebotene Algorithmenvarianten ...

...und die zugehörigen Methoden mit vorerst leerem Body.

```
/** Algorithm: Does the submass exist? */
public void massExists() { }                          // Variant (1)

/** Algorithm: How many different submasses occur? */
public void massCount() { }                           // Variant (2)

/** Algorithm: List all occuring submasses! */
public void massList() { }                            // Variant (3)
```

Listing 12: Beispieltest Teil VI: ...und ihre Signatur

An dieser Stelle fügt man den Namen der Testklasse in die Konfigurationsdatei *tests.txt*[37] ein und startet das System. Hätten wir etwas vergessen, würde eine Exception ausgelöst. Drückt der Benutzer nun auf den RUN Button (für den enstprechenden Test) wird der leere Methodenbody aufgerufen – es passiert also (noch) nichts besonders.

5.4.2 Der Algorithmus

Nachdem die erste Hürde schon geschafft ist, folgt nun das Implementieren obiger drei Prozeduren. Alle drei haben dabei etwas gemeinsam: sie greifen auf einen Datenstruktur zurück, die in einem Vorverarbeitungsschritt[38] berechnet wird. Beginnen wir mit der Deklaration dieser Datenstruktur:

```
/** preprocessed data: the calculated submasses */
protected double[] submasses;
/** the string, which we calculate the submasses for */
protected String curStr;
```

Listing 13: Beispieltest Teil VII: Variablen zur Speicherung vorverarbeiteter Daten

[37] Das Format der Datei ist im Kopf des Files beschrieben.

[38] Im folgenden wird davon ausgegangen, dass der Programmierer den BINSEARCH Algorithmus verstanden hat und die hier gewählte Implementierung nicht im Detail erklärt wird. Es geht an dieser Stelle vielmehr um die Implementierung eines Tests an sich, was sich am einfachsten an Hand eines Beispiels bewerkstelligen lässt.

Im Array *submasses* speichern wir alle gefunden Teilmassen. Die Variable *curStr* dient dazu festzustellen, ob sich der Eingabestring seit der letzten Eingabe verändert hat und repräsentiert gleichzeitig eine Kopie der Eingabe. Je nach Inhalt der Variablen wird das Preprocessing ausgeführt oder direkt auf die bereits früher berechneten Daten zugegriffen. Letzteres sieht dann wie folgt aus:

```
private boolean prepNeeded() {
  if (inString == null) return true;
  else return !inString.getAsString().equals(curStr);
}
```

Listing 14: Beispieltest Teil VIII: Ist das Preprocessing nötig?

Nachdem die benötigten „globalen“ Variablen definiert sind, berechnen wir alle vorkommenden Teilmassen und speichern sie im dafür vorgesehenen Array *submasses*:

```
protected void preprocess() {
  Set set = Collections.synchronizedSet(new HashSet());
  curStr = inString.getAsString();
  int n = curStr.length();
  if (n>0) {
    // calc weights between positions {0,j} | j is element of [0..n-1]
    double sum = 0;
    double val[] = new double[n];
    for(int j=0; j<n; j++) {
      sum += inString.getWeightAt(j);
      val[j] = sum;
      set.add(new Double(sum));
    }

    // calc remaining values by lookup and use of precalculated values
    for(int l=1; l<n; l++) {
      outputs.setProgress((int)(l*100/n));
      for(int r=l; r<n; r++) {
        set.add(new Double(val[r] - val[l-1]));
      }
    }

    Object[] masses = set.toArray();
    submasses = new double[masses.length];
    for(int i=0; i<submasses.length; i++) {
      submasses[i] = ((Double)masses[i]).doubleValue();
    }

    // sort array
    Arrays.sort(submasses);
  }
  else {
    curStr = null;
    submasses = new double[0];
  }
}
}
```

Listing 15: Beispieltest Teil IX: Der Preprocessing Algorithmus

Nachdem wir uns versichert haben, dass diese Methode das Array *submasses* mit allen vorhandenen Teilmassen füllt, generieren wir die verschiedenen darauf basierenden Varianten. Jede der drei Varianten ruft dabei zuerst *update()* aus der Superklasse *CSimpleTestInfo* auf, um die Eingaben auf dem neuesten Stand zu halten. und prüft anschliessend. ob das Preprocessing überhaupt ausgeführt werden muss.

```
/* Algorithm: Does the submass exist? */
public void massExists() {
  update();
  if (prepNeeded()) preprocess();

  int pos = Arrays.binarySearch(this.submasses, this.m);

  if (pos < 0)
    outputs.Console().println("Submass "+this.m+" not found!");
  else
    outputs.Console().println("Submass "+this.m+" found!");

  result[0] = new Boolean(pos >= 0);
}
```

Listing 16: Beispieltest Teil X: Implementation der Variante (1)

Praktisch analog entstehen nun auch die beiden anderen Varianten.

```
/* Algorithm: How many different submasses occur? */
public void massCount() {
  update();
  if (prepNeeded()) preprocess();

  int count = this.submasses.length;

  outputs.Console().println("Number of submasses = "+count);

  result[1] = new Integer(count);
}

/* Algorithm: List all occuring submasses! */
public void massList() {
  update();
  if (prepNeeded()) preprocess();

  outputs.Console().println("Found submasses:");
  outputs.Console().println(CStringUtils.arrayToString(this.submasses));

  result[2] = this.submasses;
}
```

Listing 17: Beispieltest Teil XI: Implementation der Varianten (2) und (3)

Schliesslich versichern wir uns noch, dass alle Membervariablen unserer Klasse *OSMFPBinsearch* einen initialen Wert besitzen. Wir erledigen dies in der von *CSimpleTestInfo* geerbten Methode *reset()*. Dort wird auch das *result* Array initialisiert, worin wir jeweils die Resultate der Tests speichern. An Index *i*-1 von *result* befindet sich das Ergebnis der Variante *i*. Die Grösse des Arrays *result* entspricht demzufolge der Anzahl Varianten eines Tests.

```
/***** set some features provided by the super class (e.g. to be resetable) *****/

public void reset() {
  // do here anything to reset your algorithm
  super.reset();
  submasses = null;
  curStr = null;
  this.result = new Object[3];
}
```

Listing 18: Beispieltest Teil XII: Die Methode einen Test zu (re-)initialisieren

Der Algorithmus in nun implementiert und kann gestartet werden. Die berechneten Ergebnisse werde ins Outputfeld (die Konsole) geschrieben. Länger andauernde Berechnungen werden in unserem Beispiel durch den Fortschrittsbalken mitkommentiert *outputs.setProgress(...)*. Damit ist die eigentliche Implementierung des Tests abgeschlossen.

Alles nun Folgende hat nur noch einen kosmetischen Charakter. Es gibt, wie eingangs bereits angedeutet, noch die Möglichkeit, die berechneten Daten visuell darzustellen.

5.4.3 Visualisierung der Ergebnisse

Gerade lange Listen, wie die in Variante (3) produzierte Auflistung aller vorhandener Teilmassen, kann schnell unübersichtlich werden. Die Konsole ist vergleichbar mit einer normalen Shell, die ausschliesslich im unformatierten Textmodus arbeitet. STRATO bietet aber noch weitere Möglichkeiten, ein Resultat zu präsentieren:

- als HTML-formatierte Seite
- als (selbstgezeichnete) Grafik

An dieser Stelle müssen zuerst zwei Interfaces eingeführt werden, welche diesen Umstand ermöglichen: *ITextResult*, *IGraphicResult*. Zum besseren Verständnis sind auch die JAVADOC Kommentare (in Englisch) enthalten.

```
/**
 * This interface has to be implemented by each test that would like to produce
 * some textual output.
 */
public interface ITextResult extends IResult{

  /**
   * This callback method does the raw writing job based on the updated
   * values (managed by the test class). Use HTML Tags <html> and </html> to
   * format the text: e.g. tc.setText("<html>Result = <i>result[i]</i></html>");
   *
   * @param tc       The text component to write into
   * @param algoID  The ID of the algorithm from which the result
   *                  should be displayed
   * @exception  EUpdateException  if the necessary result data is not already
   *             available or is <code>null</code>
   */
  public void write(javax.swing.text.JTextComponent tc, int algoID) throws EUpdateException;

} // eof ITextResult

/**
 * This interface has to be implemented by each test that would like to produce
 * some graphical output.
 */
public interface IGraphicResult extends IResult {

  /**
   * This callback method can be used to setup the data to be displayed next
   *
   * @param container   This is the container to drawing has to fit into. Use
   *                      the <code>getInsets()</code> method to get its extent.
   * @param algoID  The ID of the algorithm from which the result
   *                  should be displayed
   * @exception  EUpdateException  if the necessary result data is not already
   *             available or is <code>null</code>
   */
  public void setupPainting(java.awt.Container container, int algoID) throws EUpdateException;

  /**
   * This callback method does the raw drawing job based on the updated
   * values (managed by the test class)
   *
   * @param g       The graphics handle to paint on
   * @param algoID  The ID of the algorithm from which the result
   *                  should be displayed
   * @exception  EUpdateException  if the necessary result data is not already
   *             available or is <code>null</code>
   */
  public void paint(java.awt.Graphics g, int algoID) throws EUpdateException;

} // eof IGraphicResult
```

Listing 19: Interfaces ITextResult und IGraphicResult

Wir wollen die Anwendung dieser Methoden wieder an einem Beispiel erläutern, indem wir unsere BINSEARCH Klasse optisch tunen. Wir definieren die Darstellungsarten der drei Varianten (als HTML formatierter Text) in Tabelle 5-1:

	HTML formatierter Text
Variante (1)	Resultat in fetter Schrift gedruckt und in grüner oder roter Farbe
Variante (2)	Keine besondere Formatierung
Variante (3)	Eine HTML- formatierte Tabelle zu max. 10 Werten pro Zeile

Tabelle 5-1: Vorhandene Varianten des BINSEARCH Tests

Die jeweilige visuelle Zusatzinformation, die durch diese pathologischen Beispiele gewonnen wird, ist sehr unterschiedlich. Die Beispiele sollen bloss die Vielfalt an verschiedenen Möglichkeiten aufzeigen, die in einem anderem Kontext wesentlich sinnvoller eingesetzt werden können.

Unsere Klasse implementiert für diesen Fall das *ITextResult* Interface. Die beiden Parameter bezeichnen die Textkomponente (*tc*), wo wir die Ausgabe platzieren und die Variante (*algoID*), für welche wir den formatierten Output anfordern. Die Resultate können dabei irgendwo in der Klasse gespeichert sein, solange die *CReferenceDescriptor*en deren Ort bekannt geben. In allen unseren Beispiel-Implementationen ist dies immer noch das Array mit Namen *result*.

```
public void write(javax.swing.text.JTextComponent tc, int algoID) throws EUpdateException {
  // constants
  String BEGIN = "<html><font color=blue size=+2>RESULT:</font><p>";
  String END = "</html>";

  String htmlres = null;

  try {
    switch(algoID) {
      case 0: // Variant (1)
        htmlres = BEGIN + "<pre>" +
                ( ((Boolean)result[algoID]).booleanValue() ?
                  "<font color=green size=+4>YES, submass "+m+" found!" :    // if found
                  "<font color=red size=+4>NO, submass "+m+" not found!") +  // if not found
                  "</font></pre>" + END;
      break;

      case 1: // Variant (2)
        htmlres = BEGIN + "<ul>" +
               "<li>There are " + result[algoID] + " different submasses found,</li>" +
               "<li>in the String " + inString.getAsString() + "</li>" +
               "</ul>" + END;
      break;

      case 2: // Variant (3)
        double[] list = (double[])result[algoID];
        int row = list.length / 10;  if ((list.length % 10) > 0) row++;
        htmlres = BEGIN + "<table border=\"1\">";
        for (int i=0; i<row; i++) {
          htmlres += "<tr>";
          int col = ((i+1)*10 <= list.length) ? 10 : list.length % 10;
          for(int j=0; j<col; j++)
            htmlres += "<td>" + list[i*10+j] + "</td>";
          htmlres += "</tr>";
        }
        htmlres += "</table>" + END;
      break;

      default: // -> should not occur ;-)
        // do nothing
      break;
    }
```

```
      tc.setText(htmlres);  // display it
    }
    catch (Exception e) {
      throw new EUpdateException("");
    }
  }
```

Listing 20: Implementation des ITextResult Interfaces

Die Implementation erzeugt HTML Code[39]. Eingeschlossen in das HTML Tag Paar `<html> </html>` wird der Text formatiert. Dabei sind alle Tags bis und mit HTML Version 3.2 unterstützt. Insbesondere sind dies auch Tabellen oder andere Elemente zur Strukturierung der Daten. Geht irgend etwas schief, etwa durch das Auftreten einer *java.lang.NullPointerException*, wird eine *EUpdateException* ausgelöst. Diese Information wird ausgewertet und dem Benutzer als Dialogbox angezeigt.

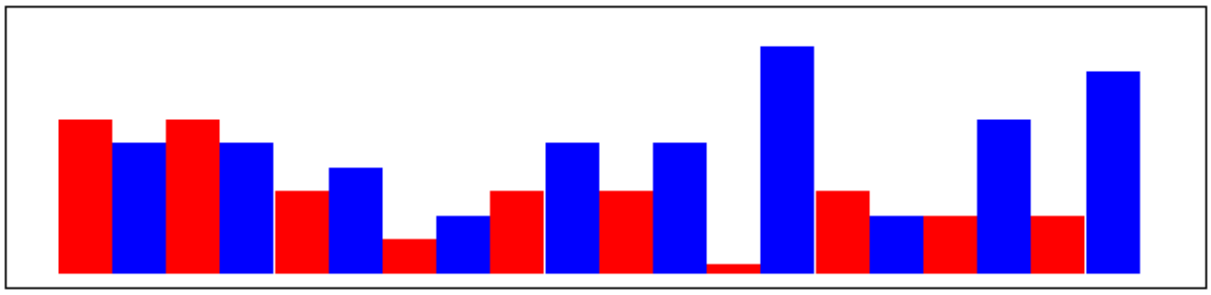

Abbildung 5-6: Grafische Ausgabe eines Testsresultates

Etwas komplizierter ist die grafische Ausgabe. Wir betrachten zur Veranschaulichung einen anderen Test. In diesem Fall handelt es sich um einen Statistiktest in der Klasse *Statistics*, der für einen String die absoluten und die relativen Auftretenshäufigkeiten der Buchstaben über dem zugeteilten Alphabet bestimmt. Das Interface für Grafikausgaben heisst *IGraphicResult* und umfasst, wie in Listing 19 bereits gezeigt, folgende zwei Methoden.

```
public void setupPainting(java.awt.Container container, int algoID) throws EUpdateException;
public void paint(java.awt.Graphics g, int algoID) throws EUpdateException;
```

Listing 21: Methoden des Interfaces IGraphicResult

Die erste der beiden wird aufgerufen, um die durch den Test berechneten Resultate in die richtige Form zu bringen. Der Parameter *container* enthält beispielsweise die Methoden *getWidth()* oder *getHeight()*, um die Ausmasse des zur Verfügung stehenden Zeichenbereichs zu ermitteln. Balkendiagramme und ähnliches können so, je nach vorhandenem Platz, kleiner oder grösser skaliert werden. Die konkreten Funktionen, um Striche, Punkte, Kreise oder andere geometrische Objekte zu zeichnen, werden schliesslich in der zweiten der beiden Methoden platziert. Dort wird als erster Parameter eine Referenz auf ein *Graphics*[40] Objekt zur Verfügung gestellt, welche die Zeichenfläche darstellt. Letztere Methode wird jedes Mal

[39] Eine sehr gute Referenz zu HTML bietet das SELFHTML Projekt: http://www.netzwelt.com/selfhtml/

[40] Falls der Programmierer das JAVA 2D Framework verwenden will, muss er das Objekt in ein Graphics2D Objekt *casten*. Danach steht ihm die gesamte Grafikfunktionsvielfalt, die Java zur Zeit bieten kann, zur Verfügung. Näheres zum JAVA 2D API findet sich unter: http://java.sun.com/docs/books/tutorial/2d/index.html

aufgerufen, wenn das Fenster aus irgendeinem Grund neu gezeichnet werden muss. Aus diesem Grund wurden die Vorgänge auch in zwei Methoden aufgeteilt[41].

	Java2DGrafik	Hilfsklassen
Absolutwerte	Histogramm	CBarPlotAssistant
Relativwerte	Liniendiagramm	CLinePlotAssistant

Tabelle 5-2: Vorhandene Varianten des Statistik Tests

```
private CPlotAssistant plotter = null;

public void setupPainting(java.awt.Container container, int algoID)
                                                        throws EUpdateException {
  try {
    // setup (x,y) pairs
    double[] x = new double[inAlphabet.getSize()];
    double[] y = new double[inAlphabet.getSize()];
    for(int i=0; i< x.length; i++) {
      x[i] = (double)i;
      switch (algoID) {
        case 0: // result is int[]
          y[i] = (double) ( ((int[])result[0])[i] ); break;
        case 1: // result is double[]
          y[i] = (double) ( ((double[])result[1])[i] ); break;
        default:  // only to avoid traps
          y[i] = 0;
      } // eof switch
    }

    // setup drawing aera (calculate viewport)
    double maxY = 0.0;
    for(int i=0; i< x.length; i++) { maxY = (y[i]>maxY) ? y[i] : maxY; }
    switch (algoID) {
      case 0:
        maxY += 1.0;
        plotter = new CBarPlotAssistant(container);
        break;
      case 1:
        maxY *= 1.5;   // there are all values < 1.0;
        plotter = new CLinePlotAssistant(container);
        break;
      default:
        maxY = 10.0;
    }

    // assign the calculated viewport and plot the values
    plotter.setOutputBounds(0.0,0.0,x.length,maxY);
    plotter.setupPlot(x, y, java.awt.Color.blue);
  }
  catch (Exception e) {
    throw new EUpdateException("");
  }
}

public void paint(java.awt.Graphics g, int algoID) throws EUpdateException {
  try {
    // delegate the painting job to the plotter
    plotter.paintContent(g);
  }
  catch (Exception e) {
    throw new EUpdateException("");
  }
}
```

Listing 22: Implementation des IGraphicResult Interfaces

[41] Wem das zu mühsam ist, der implementiert *setupPainting* mit einem leeren Body und programmiert alles in der *draw* Methode. Die *draw* Methode sollte aber nicht zu viel Zeit beanspruchen, da sonst der Aufbau der Oberfläche unnötig verlangsamt wird.

Wir speichern die Resultate eines Tests in zwei Arrays (x-Werte, y-Werte). Diese Daten nun zu visualisieren ist in den verschiedensten Arten möglich. Zur Veranschaulichung wurden zwei Typen von Charts, wie wir sie aus Tabellenkalkulationsprogrammen kennen, rudimentär implementiert (siehe Sourcecode von *CPlotAssistant*, *CBarPlotAssistant* und *CLinePlotAssistant* im Anhang). Eine Variante implementiert Balken- und eine Liniendiagramme. Die Funktionsweise dieser Hilfsklassen zu erkunden, wird dem Leser überlassen. Die Tabelle 5-2 zeigt, wie wir die Werte visualisieren, und die Abbildung 5-6 ein möglicher Output für die erste Variante.

5.5 User Interface

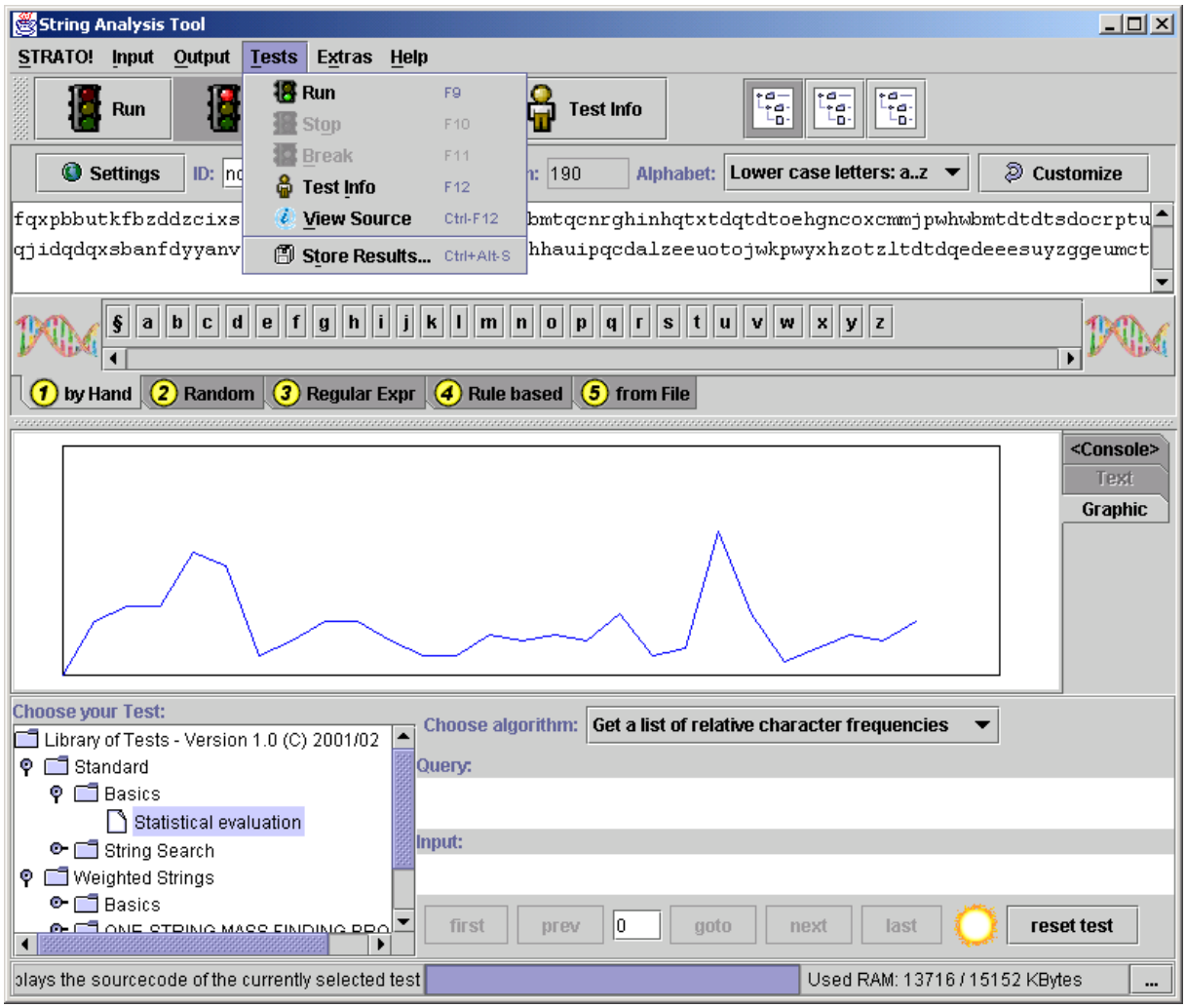

Abbildung 5-7: Snapshot: Die STRATO Applikation

Wir können, aus Gründen des grossen Umfangs[42], nicht im einzelnen auf die Klassen jener Packages eingehen, welche die Benutzerschnittstelle implementieren. Es folgt daher nur eine kurze Beschreibung deren Inhalts. Anschliessend werden zwei Themen kurz aufgegriffen. Dies sind der Entscheid betreffend der verwendeten GUI-Bibliothek und mögliche Einsatzgebiete der Kommandozeilen-Benutzerschnittstelle.

[42] Über das GUI-Package könnte, vom ihrem Umfang her, eine separate Diplomarbeit geschrieben werden!

Package com.strato.gui

Dies ist die im Rahmen der Diplomarbeit implementierte grafische Benutzerschnittstelle. Sie unterstützt nebst Eingabe- und Ausgabebereich einen zur Laufzeit dynamisch wechselbaren Manipulationsbereich (auswechselbares Modul im untersten Drittel der Oberfläche). Eine Menüzeile und schliesslich eine Toolbar sorgen für schnellen Zugriff auf häufig gebrauchte Funktionen.

Package com.strato.ui

Eine kommandobasierte Version, die mittels Scripting automatisierte Tests mit variierendem Input ermöglicht. Sie wurde nur zu Testzwecken implementiert und ist nicht in einem ausgereiften Zustand!

5.5.1 AWT oder Swing?

Zur Programmierung der graphischen Oberfläche stehen dem Javaprogrammierer standardmässig zwei Toolkits zur Verfügung. Einerseits das *Abstract Windowing Toolkit* (*AWT*), das für die Erzeugung der einzelnen Controls (Buttons, Sliders, Textfields, ...) auf die native Routinen des Betriebssystems zugreift und andererseits *SWING*. Letzteres ist eine 100% in Java geschriebene GUI-Bibliothek, die einen grossen Umfang besitzt. Durch *Memoryleaks* und eine nicht überzeugend schnelle Performanz, ist sie in letzter Zeit immer mehr ins Kreuzfeuer der Kritik geraten.

Die Entscheidung fiel trotzdem auf SWING. Bessere optische Verarbeitung und die grössere Auswahl auch an komplexer Controls. Vorhandene Baumstrukturen und bereits voll funktionstüchtige Editoren (zum Beispiel für HTML-Text), glichen die Performanceeinbussen beim Aufbau der Oberfläche aus.

5.5.2 Benutzerschnittstelle auf Kommandozeilenebene

Wird das STRATO Projekt irgendwann einmal mehrere hundert Tests umfassen, die ihrerseits noch einmal so viele Parameter verlangen, könnte eine Automatisierung der Tests bald der einzige Ausweg werden, umfangreiche Analysen von Strings (oder Klassen von Strings) sinnvoll durchzuführen. Dies wiederum verlangt den Einsatz von *Skripting*sprachen, die meist in jeder Shell vorhanden sind. Die Klassen *com.strato.cui.CStratoCmd* und *com.strato.cui.CStratoBatch* zeigen Ansätze in diese Richtung und erlauben die Ausführung eines Tests, in dem die Eingabestrings aus einem File stammen (in diesem Fall mit *com.strato.cui.CInputSet* erzeugt). Die Ausgabe kann auf die Konsole oder ein anderes File gelenkt werden (siehe 5.2.3 „Streams“).

5.6 Sonstige Bestandteile des Frameworks

Wir widmen uns dem letzten Teil des Frameworks: Der Ausführungseinheit für Tests. Nachdem ein Test geschrieben wurde, sollte er zweckmässigerweise auch gestartet werden können. Die Laufzeit mag in vielen Fällen im Millisekundenbereich liegen, kann aber für einzelne Tests und/oder überlange Eingabegrössen mehrere Tage dauern. Dies hängt ausserdem von der Komplexität der Aufgabe und dem Laufzeitverhalten der gewählten Implementierung ab. Doch gerade die Optimierung eines Algorithmus wird meist Schritt für Schritt erreicht und ein vorzeitiges Abbrechen langandauernder Tests wäre wünschenswert. Alternativ könnte man das Programm beenden. Dies bedingt jedoch nach dem Neustart auch die Neueingabe aller sonstiger Einstellungen (Parameterwerte, ...). Aus diesem Grund wurde versucht, die Ausführung des Tests (Algorithmus) von den restlichen *Threads* der Applikation

(dem Eventthread, dem GUI-Thread, ...) zu trennen. Soll nun ein Test abgebrochen werden, reicht es, diesen neuerzeugten Thread zu terminieren.

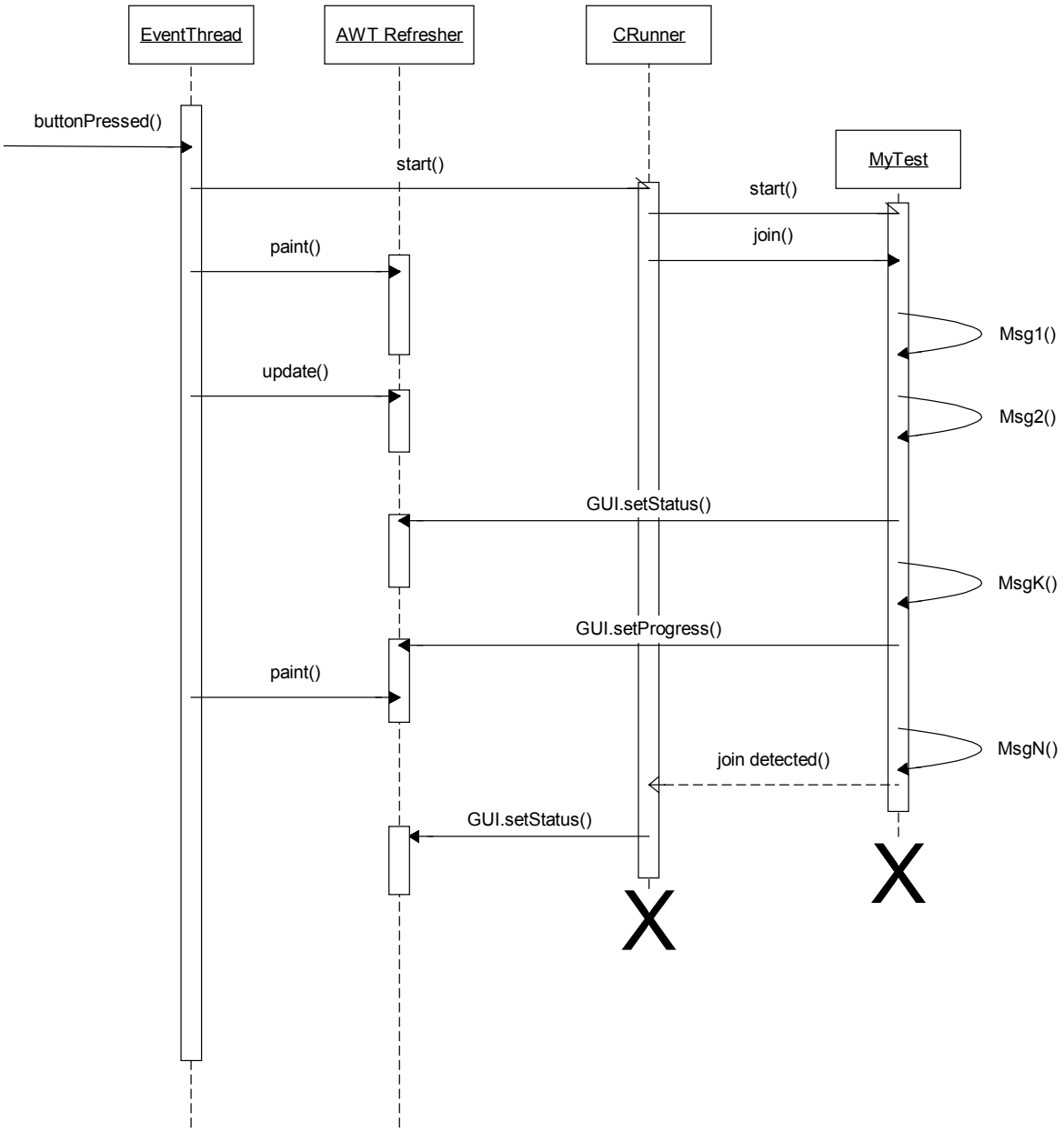

Abbildung 5-8: UML Sequenzdiagramm des Ablaufs beim Starten eines Tests

Weitere Vorteile sind:

- das sofortige (asynchrone) Abbrechen eines Tests (unabhängig davon, ob noch andere Tasks synchron anstehen);
- ein möglicher Fehler im Test verlangt keinen Abbruch der Applikation selbst (kann diese nicht zum Crash führen), da die Hauptthreads der Applikation nicht involviert sind;
- ein Test kann suspendiert (pausiert) und wieder aufgenommen werden;

- dem laufenden Test ist es erlaubt, Statusinformation an die GUI zu senden, so dass diese auch rechtzeitig angezeigt wird und nicht erst, nachdem der Test beendet wurde;
- dem Test ist es ausserdem erlaubt, die Ausführung seiner Berechungen selbst auf mehrere Threads zu verteilen. Der „Hauptthread“ muss am Schluss auf die Terminierung aller zusätzlich erzeugten Threads warten, bevor er die Kontrolle wieder ans System zurückgibt.

Das Sequenzdiagramm in Abbildung 5-8 verdeutlicht das Zusammenspiel. Damit sich die Situation etwas vereinfacht, sind nur die wichtigen Threads eines laufenden Java (*Heavyweight*-)Prozesses aufgelistet. Zum einen ist dies jener Thread, der für die Abhandlung von Ereignissen (engl. *Events*) – der Benutzer klickte oder drückte etwas – verantwortlich ist. Zum andern handelt es sich um den GUI-Thread, welcher für das stete Neuzeichnen der verschiedenen AWT/SWING GUI-Elemente zuständig ist.

5.7 Zusammenfassung

Dieses Kapitel zeigte uns einen Einblick in die Implementierung des Tools. Angefangen bei den Kernkomponenten *CString* und *CAlphabet* und deren Umsetzung in Java waren nebst sehr allgemeinen Themen (beispielsweise dem Datenaustausch) auch einzelne spezielle Themen aus dem vorangegangenen Kapitel auf ihre konkrete Implementation hin genauer durchleuchtet worden. Die Plug-in Schnittstelle wurde beispielsweise durch das Prinzip der Deskriptoren realisiert. Jeder Test gibt selbstständig darüber Auskunft, welche Funktionalität es gibt und in welcher Ausprägung diese angeboten wird. Ein kleiner Kurs anhand eines konkreten Beispiels erklärte Schritt für Schritt das Schreiben eines eigenen Tests. Designentscheide über ein graphische und/oder kommandzeilenbasierte Version fehlten ebenso wenig wie Erklärungen über die Abläufe eines Tests, nachdem der RUN-Button in der GUI gedrückt wurde. Aussagen zur Organisation der Klassen und verwendete Programmierkonventionen versuchten ihrerseits die Wartbarkeit des gesamten Projektes zu erleichtern.

6 STRATO Tutorial (Version 1.0)

6.1 Überblick – erste Schritte

Da es sich hier aber nicht um ein weiteres Produkt eines Textverarbeitungssystems oder um den x-ten Klon erfolgreicher Bildbearbeitungssoftware handelt, werden die grundlegenden Features sowie die Bedienung des Tools in der Form eines kompakten Tutorials wiedergegeben.

A) INSTALLATION

Kopieren Sie die Datei *strato.jar* in ein beliebiges Verzeichnis, und wechseln Sie in dieses Verzeichnis. Sollte bereits eine Javaversion[43] (1.3 oder höher) auf dem Computer installiert sein kann mit Schritt B weitergefahren werden.

Sollte sich auf dem System noch keine, oder eine frühere Version von Java befinden, installieren Sie die Software gemäss Anweisungen der jeweiligen Java Distribution. Näheres dazu finden Sie beispielsweise auf der Web-Seite von SUN™: http://java.sun.com.

B) STARTEN DES PROGRAMMS

Hier nun ein paar Möglichkeiten, wie STRATO™ auf diversen Plattformen zum Leben erweckt wird. Die Umgebungsvariable JAVAPATH bezeichnet dabei den Pfad des Javainterpreters, während CLASSPATH die Verzeichnisse der Klassenbibliotheken enthält.

Microsoft Windows™:

Via Kommandozeile (Eingabeaufforderung)

```
>%JAVAPATH%\java -cp %CLASSPATH%;strato.jar com.strato.gui.Strato
```

oder einfacher

```
>%JAVAPATH%\java –jar -cp %CLASSPATH% strato.jar
```

Aus dem Explorer / vom Desktop

Falls das *Java Development Kit* (JDK Version 1.3 oder höher) mit den Standardeinstellungen installiert wurde, sollte ein Doppelklick auf die JAR-Archivdatei strato.jar den Java-Interpreter mit der im Archiv vermerkten „Startklasse“ `com.strato.gui.Strato` automatisch starten.

Alternativ kann das Tool auch als Applet gestartet werden. Allerdings kann STRATO als Applet nur mit eingeschränkter Funktionalität, insbesondere was den Zugriff auf das Dateisystem betrifft, benutzt werden. Hierzu muss beim Internet Explorer™ das Java-Plugin

[43] Um die Version einer installierten Javaumgebung zu ermitteln, muss der Aufruf von „java –version“ von der Kommandozeile/Shell aus erfolgen.

(ebenfalls Teil der JDK Distribution) installiert sein. Ein Doppelklick auf die Datei *StratoPlugin.html*[44] bringt den gewünschten Effekt. Netscape™ und Opera™ User benötigen ebenfalls das Plug-in.

UNIX™ / LINUX™

Via Shell

```
>%JAVAPATH%\java -cp $classpath:strato.jar com.strato.gui.Strato
```

Für weitere Informationen betreffend Java-Interpreter und dessen Kommandozeilen-Schalter informiert der Aufruf des Interpreters (ohne irgendwelche Argumente).

6.2 Aufbau der graphischen Oberfläche

Das Konzept der drei Grundpfeiler Eingabe, Verarbeitung und Ausgabe, auf welchem die interne Architektur des Software basiert, spiegelt sich auch 1:1 auf der Benutzeroberfläche wieder (siehe Abbildung 6-1).

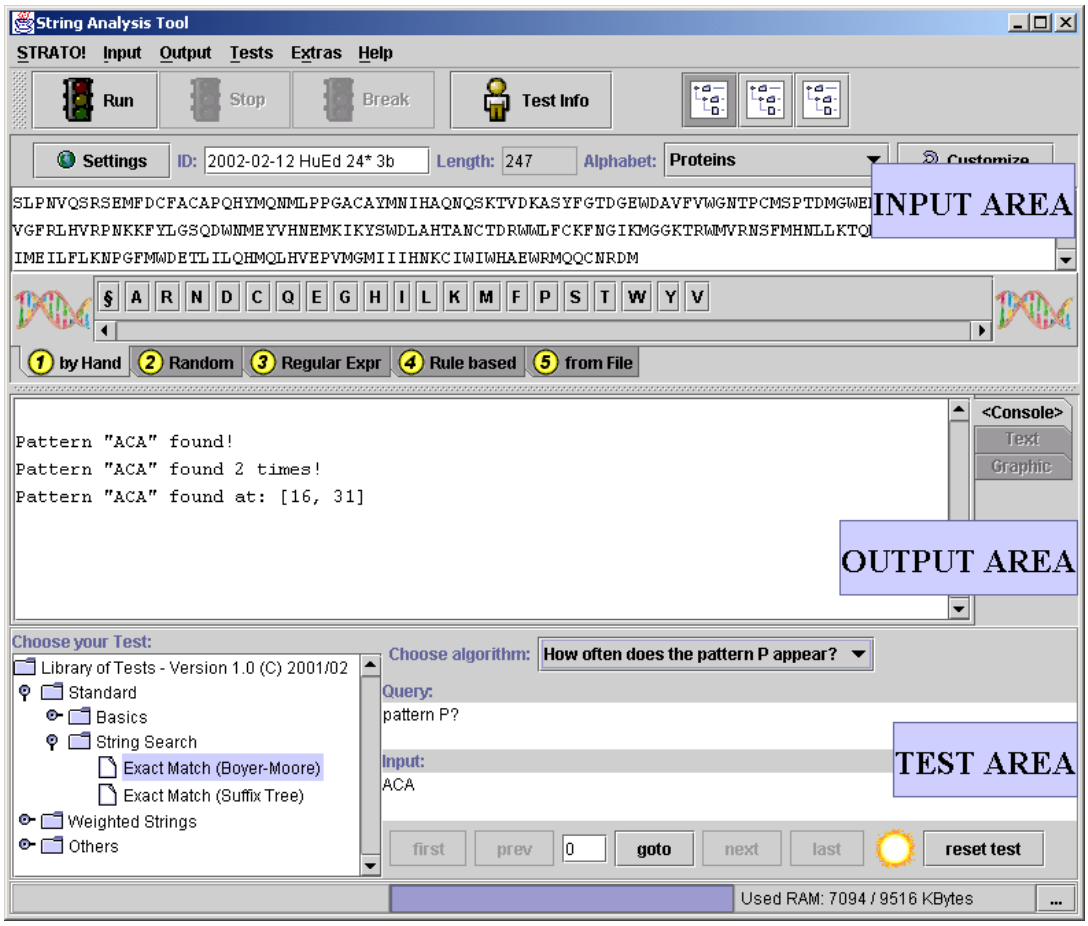

Abbildung 6-1: Die drei Hauptkomponenten: Eingabe-, Ausgabe- und Testbereich

[44] Diese Datei muss sich im selben Verzeichnis wie *strato.jar* befinden

6.2.1 Eingabebereich

Im Eingabebereich wählt der Benutzer die gewünschte Quelle (Erzeugungsmethode) des Eingabestrings. Hierbei hat er zur Zeit die Wahl zwischen fünf verschiedenen Methoden, die teils ihrerseits noch weiter unterteilt sind. Die gewünschte Herkunft der Eingabe wird durch einen Klick auf den entsprechenden Reiter (engl. *Tab*) getätigt bzw. durch den dazugehörigen Shortcut (CTRL-1 bis CTRL-5) festgelegt. Obwohl das Tool grundsätzlich per Maus gesteuert wird, finden sich diese wie auch viele andere Funktionen zusätzlich auch noch in der Menüzeile[45] wieder.

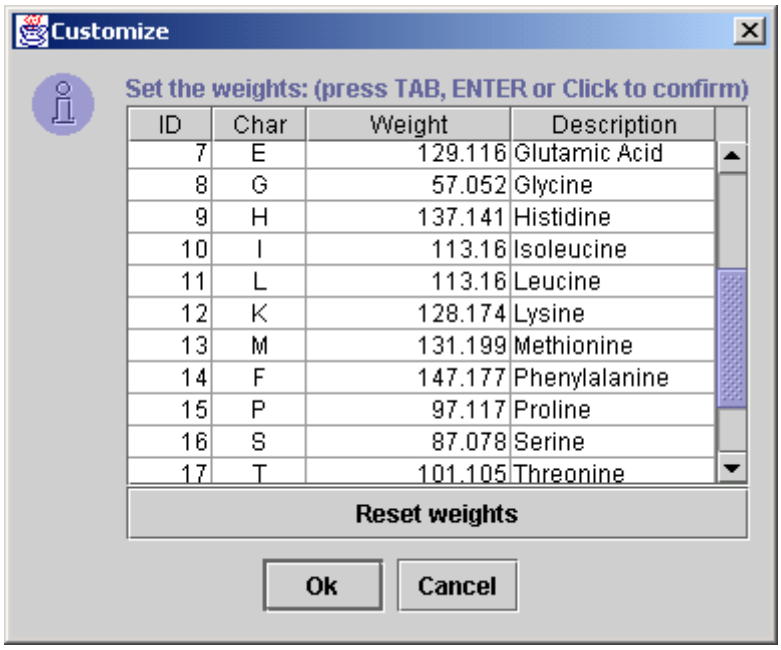

Abbildung 6-2: Snapshot: Dialogbox zur Änderung der Gewichte

Nach der Wahl der Quelle sollten die Basisvariablen eines Strings ausgewählt werden. Je nach Typ der Stringerzeugung kann das Alphabet, die Länge und/oder ein Bezeichner frei gewählt werden oder ergeben sich implizit. Konkrete Informationen dazu entnehme man der Tabelle 6-1. Die entsprechenden Controls befinden sich oberhalb des weissen 3-zeiligen Anzeigefensters für den String. Der Button (*Customize*) befindet sich direkt neben der Checkbox zur Auswahl des gewünschten Alphabetes und erlaubt dessen Zeichen-Gewichte nach den eigenen Bedürfnissen zu konfigurieren (siehe Abbildung 6-2). Dabei ist darauf zu achten, dass es zur Zeit zwei verschiedene Typen von Alphabeten gibt. Benutzerdefinierte und Standard-Alphabete. Erstere werden dynamisch auf Grund der im Eingabestring verwendeten Zeichen zur Laufzeit generiert, während letztere aus einer Menge von vorgegebenen Zeichen besteht (siehe Tabelle 4-1).

Sind die wenigen „globalen“ Anforderungen an den String einmal gestellt, können je nach gewählter Erzeugungsmethode noch weitere Einstellungen getroffen werden. Wer detailliert an den Ideen und Umsetzungen der einzelnen Stringerzeugungsmethoden interessiert ist, wird auf das Kapitel 4.2 verwiesen. Es folgt eine kurze Zusammenfassung, welche Einstellungen bei den einzelnen Methoden I) – V) vorgenommen werden können.

[45] Ein Kurzreferenz zu allen Menübefehlen finden Sie in Kapitel 6.3.

ATTRIBUT	WERTEBEREICH	I	II	III	IV	V
Stringlänge	$\in IN_0$	✓[46]	✓		✓	
Alphabet	$\in X$, X:= {Menge der Alphabete}	✓	✓	✓	✓	
Id	([a..Z], [0..9], [-,.])*	✓				

Tabelle 6-1: Die veränderbaren Attribute geordnet nach Erzeugungsart

I) By Hand (manuelle Erzeugung)

Hier werden, je nach Wahl des Alphabetes, die verwendbaren Zeichen als Buttonarray aufgelistet. Sie können entweder durch direktes Eintippen via Tastatur oder durch Drücken eines dieser Buttons eingegeben werden (siehe Abbildung 6-3).

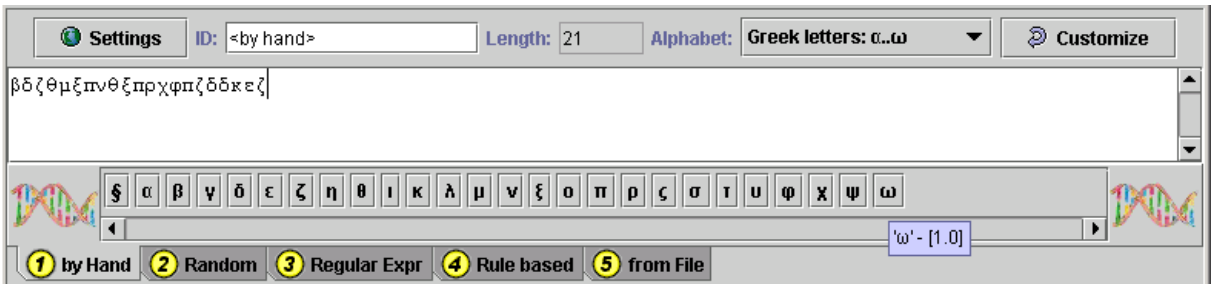

Abbildung 6-3: Snapshot: Globale Attribute sowie die Buttons der Eingabe per Hand

II) Random (zufällige Erzeugung)

Die Auswahl eines Zufallsgenerators und gegebenenfalls Angaben zu einem initialen Seed, werden bei diesem Reiter vorgenommen. Es muss bei diesem String-Generator immer auch die Länge (≥1) des zu erzeugenden Strings angegeben werden. Dann drücke man *GO!* (siehe Abbildung 6-4).

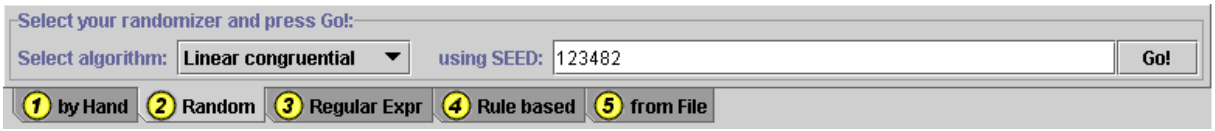

Abbildung 6-4: Snapshot: Zufällige Stringerzeugung

III) Regular (Erzeugung aus regulärem Ausdruck)

Sehr lange, sich selbstähnliche und/oder repetitive Muster enthaltende Strings, erzeugt man am besten durch Eingabe eines regulären Ausdrucks (siehe Abbildung 6-5). Die Syntax der verwendeten Sprache Regula wurde bereits in Kapitel 4.2.3 erklärt.

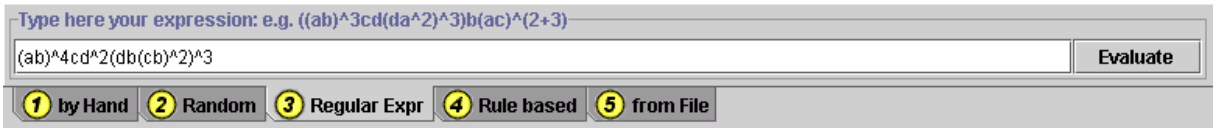

Abbildung 6-5: Snapshot: Erzeugung aus regulärem Ausdruck

[46] Die Länge kann hier durch Eingabe oder Löschen eines oder mehrerer Zeichen variiert werden, da die Anzeige mit dem Eingabefeld synchronisiert ist.

IV) Rule based (Erzeugung nach einer Regel)

Das ist die flexibelste Art, einen String zu erzeugen. Das System erlaubt dem Benutzer die Wahl einer Generierungsfunktion (siehe Abbildung 6-6). Die zugehörigen Parameter werden dabei wie bei Javasourcecode, also durch Komma getrennte und in Java Syntax eingegebene Listen, übergeben. D.h. Strings ins Anführungszeichen, Floats mit einem Suffix f etc. (siehe Tabelle 6-2).

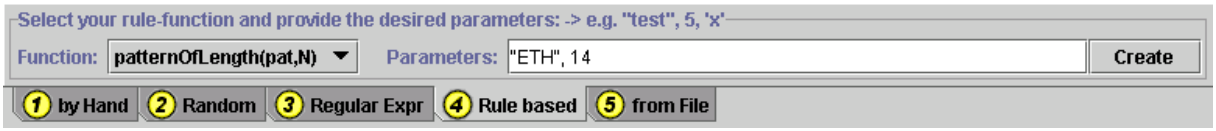

Abbildung 6-6: Snapshot: Erzeugung aus Regel

TYPE	SYNTAX	BEISPIEL
String	Text mit zwei umschliessenden Anführungszeichen	“mein Text“
char	Ein Zeichen mit zwei umschliessenden Apostrophen	’X’
byte	Suffix b	12b
short	Suffix s	-15s
int	Optionaler Suffix I	129
long	Suffix L	545605635L
float	Suffix f	345.34f
double	Optionaler Suffix d	-1.23
boolean	true | false	true

Tabelle 6-2: Formatierungsregeln für Basistypen

V) From File (Erzeugung durch Auslesen aus einer Datei)

Die letzte Methode ist schliesslich das Laden eines vorher abgespeicherten Files, bzw. eines Strings, der in einem vorgegebenen Format als Textdatei vorliegt (siehe dazu Kapitel 4.2.5). Das sich öffnende Dialogfenster verweist dabei standardmässig stets auf das HOME Verzeichnis des Benutzers.

Unter Microsoft Windows 2000™ wäre dies beispielsweise:

```
C:\Dokumente und Einstellungen\<username>\
```

Unter UNIX™ wäre dies z.B. :

```
\home\<username>\
```

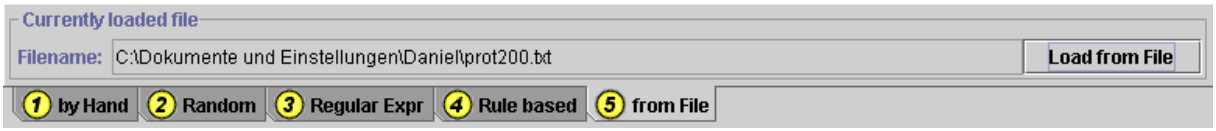

Abbildung 6-7: Snapshot: Auslesen der Eingabe aus einer Datei

Algemeine Einstellungen:

Hinter dem Button *Settings* verbirgt sich unter anderem die Möglichkeit, die Tastatur so zu konfigurieren, dass das Eintippen von Zeichen verunmöglicht wird, falls diese nicht dem aktuellen Alphabet angehören sollten. Standardmässig ist diese Funktion ausgeschaltet.

Falls extrem lange Strings erzeugt werden sollten, empfiehlt es sich, unter *Settings* eine obere Grenze für der Stringlänge anzugeben, bis zu welcher das „berechnete“ Resultat noch vollständig angezeigt werden soll. Als Vorgabe ist dieser Wert auf 10000 gestellt, kann aber einen beliebigen Wert im Intervall [100..MaxINTEGER] annehmen. Wird diese Einstellung zu hoch gewählt, dauert die Anzeige des Erzeugnisse eventuell ein mehrfaches an Zeit, die benötigt wurde, um den String zu erzeugen (siehe Abbildung 6-8). Wichtig hierbei ist noch anzumerken, dass diese Funktion bei einem Wechsel zum Reiter *by Hand* vorübergehend deaktiviert wird und zu Wartezeit führen kann, bis der String komplett angezeigt wird. Dies erlaubt es dem Benutzer, den halb- oder vollautomatisch generierten String durch Wechsel zu diesem Reiter noch zu bearbeiten, bevor dieser einem Test als Eingabe übergeben wird.

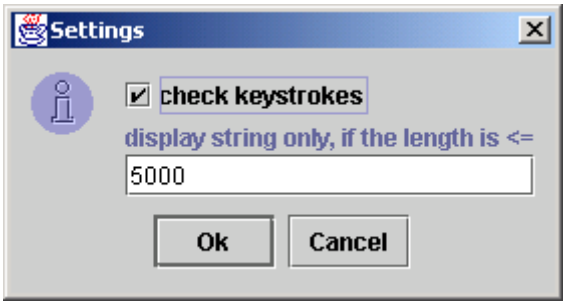

Abbildung 6-8: Snapshot: Dialogbox zur Änderung von Einstellungen

6.2.2 Tests

Die wichtigste Komponente befindet sich im untersten Drittel des Applikationsfensters. Dieser Bereich ist zuständig für die Auswahl des Tests (der Analysefunktion), der auf den soeben erzeugten oder eingegebenen Eingabestring angewandt werden soll. Es gibt zur Zeit ein Modul (anwählbar via Toolbar), welches die Auswahl und Konfiguration des Tests erlaubt. Dass es zur Zeit nur ein Modul gibt, ist aber nicht weiter schlimm. Wir können uns diesen modularen Mechanismus schon jetzt zu Nutze machen, indem wir dasselbe Modul mehrfach laden – d.h. mehrere unabhängige Instanzen erzeugen. So können beispielsweise zwei ähnliche Tests sehr einfach miteinander verglichen werden, indem sie auf denselben Eingabestring angewandt werden. Der Benutzer kann seine Algorithmen nun nicht nur strikte nacheinander, sondern abwechslungsweise nebeneinander testen, ohne durch den Wechsel zu einem andern Test bereits vorgenommene Parametereinstellungen zu verlieren (siehe Abbildung 6-9, rechter Bereich).

Abbildung 6-9: Snapshot: Die Toolbar zum Starten, Stoppen und Unterbrechen von Tests

Modul Tests

Im linken Bereich befindet sich eine baumartige Struktur, die sämtliche vorhandenen Tests kategorisch auflistet. Soll ein Test ausgewählt werden, klickt man auf ein Blatt dieser aufklappbaren Struktur, was dann im rechten Teil des Moduls weitere Einstellungen ermöglicht. Dort wird zuerst die Variante des ausgewählten Algorithmus selektiert. Diese Auswahlbox wurde eingeführt, damit viele Tests auf ein und demselben Algorithmus basieren können und am Schluss jeweils bloss einen anderen Teil der berechneten Strukturen als Resultat zurückliefern. Ein gutes Beispiel hierzu liefert das zuvor schon mehrmals erwähnte Exact Pattern Matching, bei welchem für einen gegebenen Eingabestring alle

Auftretenspositionen des Patterns aufgezählt werden. Kann diese Aufgabe beantwortet werden, sind auch folgende Fragestellungen sinnvoll: Kommt das Pattern überhaupt vor und wenn ja wie oft? An welcher Stelle geschieht dies zum ersten Mal? Zur Beantwortung dieser Zusatzfragen, können stets dieselben Kernroutinen des Algorithmus verwendet werden.

Im unteren Teil der rechten Seite lassen sich schliesslich noch die Parameter der Tests in ein Textfeld eingeben. Mittels Buttons wird navigiert. Jeder Parameter hat einen Defaultwert, der als Voreinstellung erscheint. Zur Zeit werden sämtliche Basistypen von Java – inklusive der Klasse *java.lang.String* – als Parameter unterstützt. Nebenan gibt es noch einen *Reset*-Button, um den aktuellen Test (inklusive der Parameter) in den ursprünglichen Zustand[47] zu versetzen. Schliesslich folgt noch ein Klick auf den RUN-Button in der Toolbar (siehe Abbildung 6-9). Nachdem der Test beendet wurde, erscheint das Resultat in der Konsole des Ausgabebereiches. Sollte die Ausführung eines Tests sehr lange gehen, kann dieser durch Drücken des Buttons STOP frühzeitig beendet werden. Für einen vorübergehenden Unterbruch wird die BREAK Taste benutzt. Der Tests kann anschliessend durch erneutes Drücken der RUN-Taste wieder in Gang gebracht oder mit STOP beendet werden. Für jeden Test kann zusätzlich eine INFO-Dialogbox angezeigt werden. Sie enthält allgemeine Informationen zum Test (siehe Abbildung 6-10).

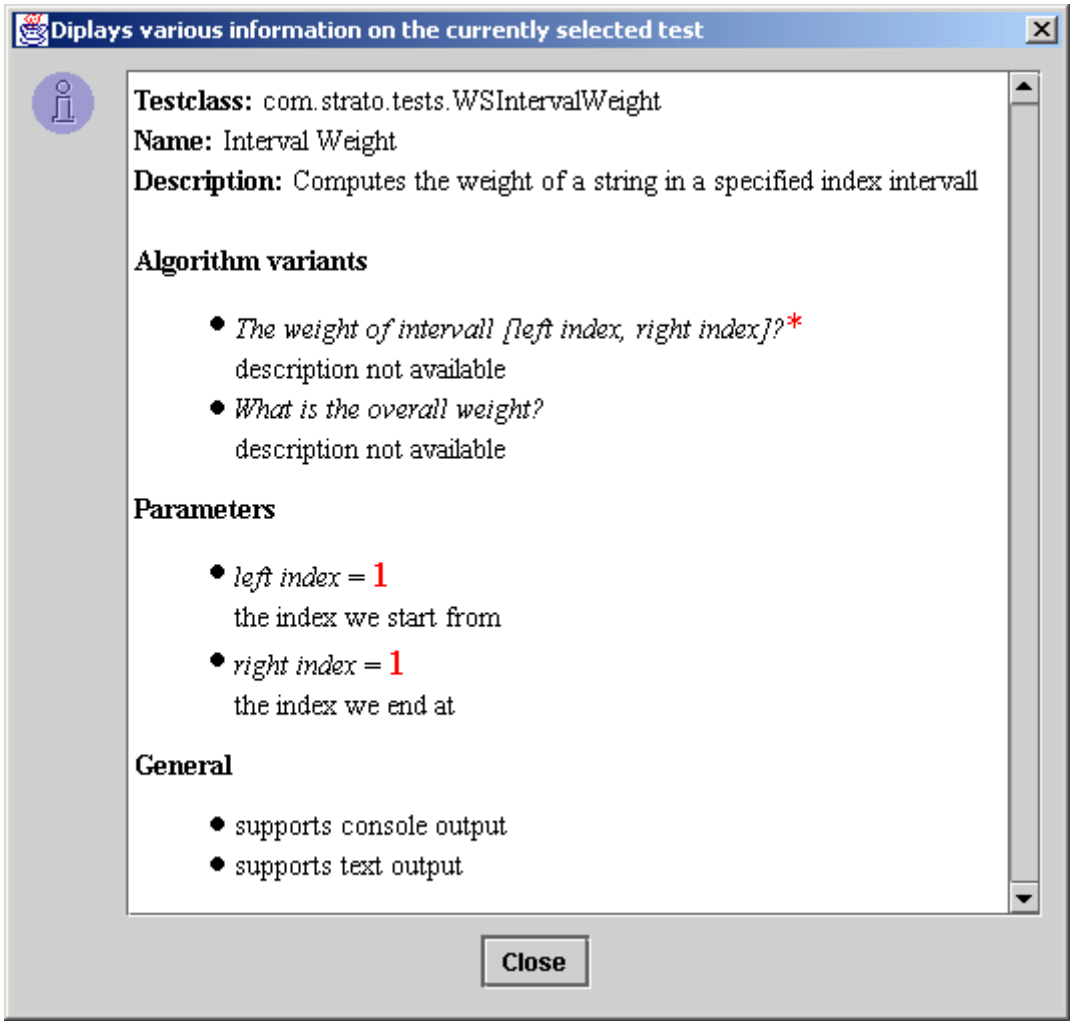

Abbildung 6-10: Snapshot: Generelle Informationen zum Test

[47] Dies versetzt den Test in jenen Zustand, wie er vor der ersten Benutzung des Tests war (nach dem Hochfahren des Systems). D.h.: Parameter mit Vorgabewerten, keine gespeicherten Preprocessing-Daten, etc.

Nur via Menü ist schliesslich noch eine weitere Funktionen im Zusammenhang mit Tests verfügbar: Die Anzeige des Quelltextes (*Sourcecode*) des aktuellen ausgewählten Tests (siehe Abbildung 6-11).

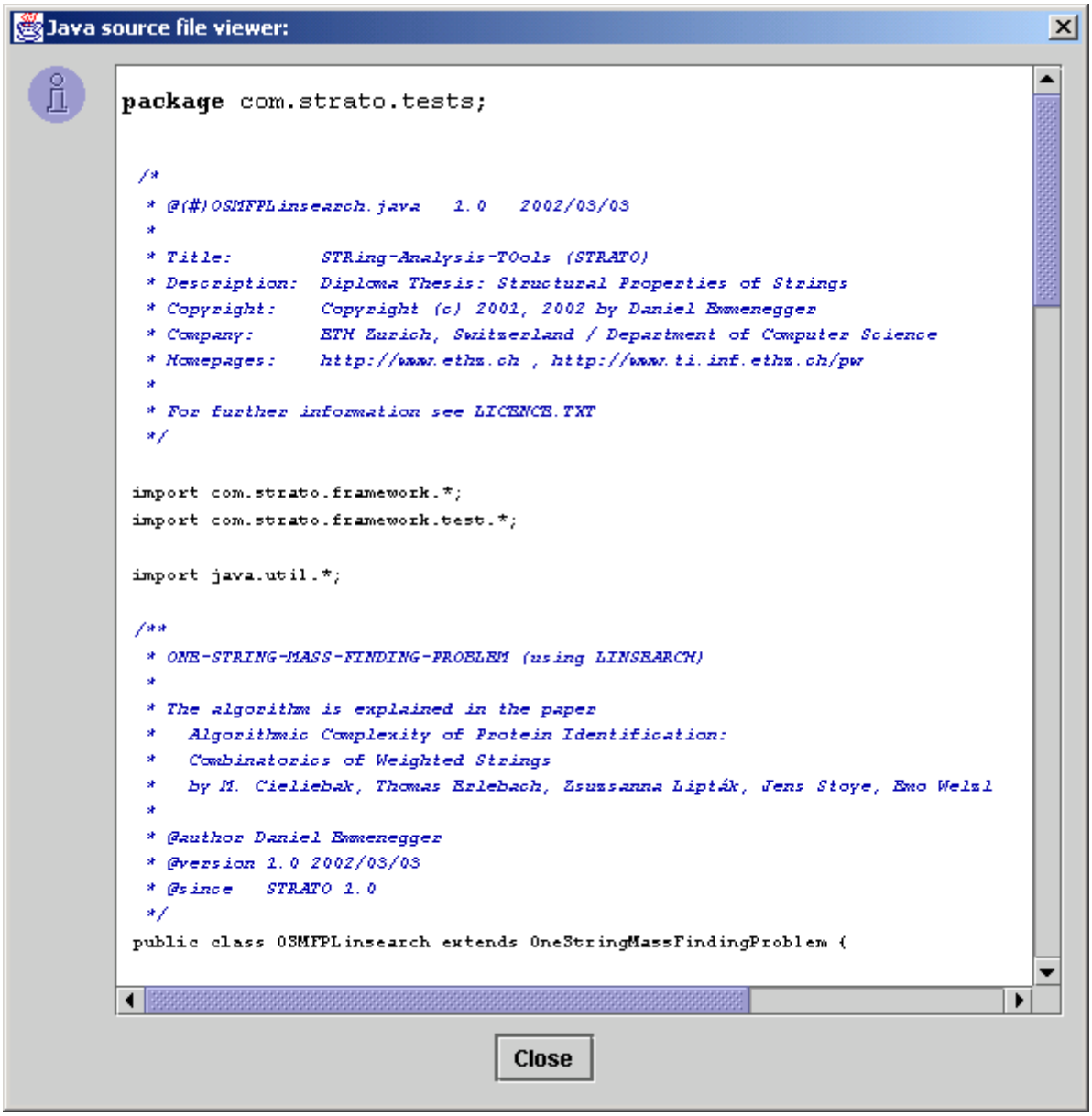

Abbildung 6-11: Snapshot: Anzeige des Sourcecodes eines Tests

6.2.3 Der Ausgabebereich

STRATO unterstützt in Version 1.0 drei verschiedene Arten von Ausgaben. Konsolenoutput, HTML Textoutput und Grafiken. Dies ist an den 3 Reitern des Ausgabebereiches zu erkennen. Welche dieser drei Möglichkeiten vom derzeitigen Test angeboten werden, ist daran erkennbar, welche der Reiter aktiv sind. Dabei können die Ausgabearten in zwei Kategorien unterteilt werden: Aktive und passive Inhalte. Die Konsole stellt stets eine aktive Einheit dar, die ähnlich zu einer Shell Ausgaben kontinuierlich auf den Bildschirm schreibt[48]

[48] Hierbei sei erneut anzumerken, dass die Ausgaben auf die Konsole aus Gründen der Effizienz zwischengepuffert werden und erst nach Beendigung des Tests (oder falls der Puffer voll ist) ins Ausgabefenster geschrieben werden.

– und zwar in der Reihenfolge, in welcher sie vom Algorithmus erzeugt wurden. Die Information zur Anzeige der passiven Inhalte wird ebenfalls durch den Test selbst bereitgestellt, indem das System bei dieser Variante die gewünschten Informationen durch Aufruf einer genau definierten Methode vom Test selbst bezieht.

Da wir uns im Zeitalter des World Wide Web (WWW) befinden, ist bei STRATO eine HTML-formatierte Ausgabe des Resultates implementiert. Dies hat den Vorteil, dass für jegliche Art von Ausgabetypen, z.B. Boolean, Skalar, Vektor, Matrix, Text, .., eine geeignete Visualisierung des Ergebnisses ermöglicht wird. Zur Darstellung einer 3x3-Matrix können beispielsweise HTML Tabellen verwendet werden, bei einer booleschen Aussage vom Typ ja/nein oder gut/schlecht eine geeignete Grafik (grünes Pluszeichen vs. rotes Minuszeichen). Ob und wie ein Tests ein berechnetes Resultat darstellt, ist vollständig Aufgabe des Programmierers des jeweiligen Tests (siehe Abbildung 6-12).

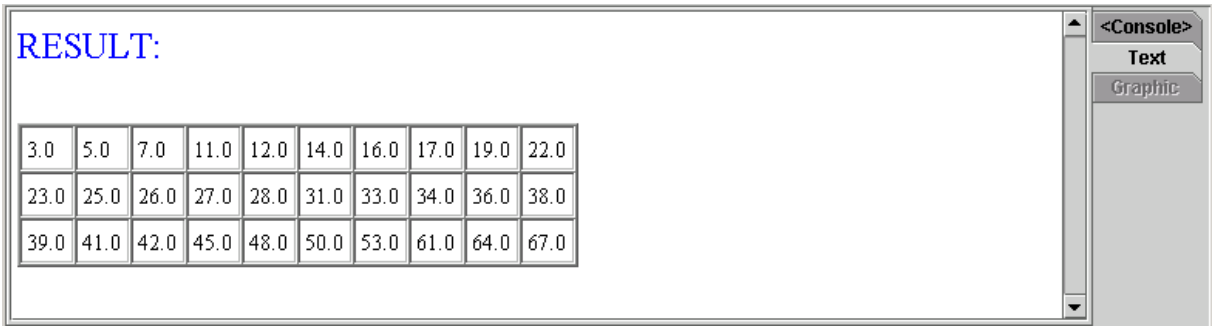

Abbildung 6-12: Snapshot: HTML-formatierte Ausgabe einer Liste von Werten

Tests, die statistische Auswertungen einer Eingabe vornehmen, liefern oft lange Vektoren. Hierbei kann eine zweite Art der Ergebnisvisualisierung eingreifen – eine Grafikausgabe. Sehr bekannt sind hierbei wohl die diversen Arten von Charts (Diagrammen), die wir aus Tabellenkalkulationen kennen. Balkendiagramme, Kreisdiagramme, Punktwolken und ähnliches sind denkbar. Aber auch ganz andere Darstellungen wie Fraktale, Richtungsfelder etc. können dazu beitragen, das Resultat aussagekräftig darzustellen. Auch diese Art der Visualisierung steht ganz in der Verantwortung des Programmierers des jeweiligen Tests. Er kann hierbei auf Bibliotheken für Grafikroutinen zurückgreifen, die einerseits Bestandteil von STRATO (oder Java selbst) sind oder gratis im Internet angeboten werden.

6.2.4 Manipulation von Eingabe- und Ausgabedaten

Sowohl Eingabe- als auch Ausgabebereich besitzen die Möglichkeit, die dort angezeigten Daten zu Löschen, Speichern und Drucken. Zu einem früheren Zeitpunkt abgespeicherte Daten, können auch wieder geladen werden. (siehe Abbildung 6-13).

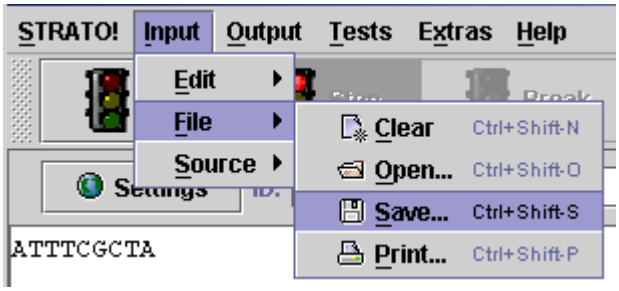

Abbildung 6-13: Snapshot: Menü zur Manipulation von Eingabedaten

6.3 Diverses

Wohl eher aus Freude am Programmieren, denn aus wirklich essentiellen Gründen ist die Möglichkeit vorgesehen, das Aussehen (*Look & Feel*) der Oberfläche frei zu wählen. Wird der „Java-Look“ selektiert, kann zudem ein geeignetes Motiv (*Theme*) ausgesucht werden. Die Palette reicht von saftgrün bis zu monochromer schwarz/weiss Darstellung der GUI Elemente.

STRATO!		
	About	Dialogfeld mit Lizenzhinweisen anzeigen
	Exit	Verlassen der Applikation
Input		
	Edit	
	Cut	Markierten Textteil ausschneiden
	Copy	Markierten Textteil in die Zwischenablage kopieren
	Paste	Textteil aus Zwischenablage einfügen
	Clear	Gesamten Text löschen
	Select All	Alles markieren
	File	
	Clear	Eingabestring löschen
	Open	Eingabestring aus Datei laden
	Save	Aktuellen Eingabestring speichern
	Print	Aktuellen Eingabestring drucken
	Source	Auswahl der Eingabequelle
Output		
	File	
	Clear	Angezeigte Ausgabe löschen
	Open	Ausgabe aus Datei laden
	Save	Angezeigte Ausgabe speichern
	Print	Angezeigte Ausgabe drucken
Tests		
	Run	Test starten
	Stop	Test stoppen (definitiv)
	Break	Test unterbrechen (vorübergehend)
	Info	Informationen zum aktuellen Test
	View Source	Quellcode des aktuellen Tests anzeigen
	Store Test Result	Aktuelle Testergebnisse speichern
Extras		
	Look & Feel	Art der Darstellung wählen
	Themes	Farbschema auswählen
Help		
	Help Overview	Einstieg zu diversen Hilfeseiten

Tabelle 6-3: Ein Auflistung aller Menüs (im Applikationsmodus)

Vielerorts geben sogenannte *Tooltips* Auskunft über die genaue Funktionsweise eines bestimmten Controls. Dazu muss lediglich mit der Maus über einer Schaltfläche etwas verweilt werden. Auch eine Online-Hilfe ist integriert und gibt über diverse Themen Auskunft.

Der Benutzer wird an dieser Stelle aber nochmals darauf hingewiesen, dass alle Funktionen, die in irgendeiner Weise auf das Dateisystem zugreifen, in der Appletvariante **nicht** funktionieren und daher im Menü gar nicht erst erscheinen!

7 Gewichtete Strings

Nachdem wir nun über ein Hilfstool zur Stringverarbeitung verfügen, wollen wir dieses für eine etwas komplexere Fragestellung im Zusammenhang mit gewichteten Strings einsetzen. Betreffend Notation und Aufgabenstellung stützen wir uns dabei auf den Technical Report (TR) *Algorithmic Complexity of Protein Identification* [5].

Die im Verlaufe der Diplomarbeit entstandenen Beispiel-Implementationen verschiedenster Tests sind meist nur zum dem Zwecke erstellt worden, ein bestimmtes Feature des Frameworks (bzw. der Plug-in Schnittstelle) zu demonstrieren (siehe Abbildung 5-5) Wir widmen uns nun einem Problem, dessen Definitionen wir im obigem TR finden und hier zusammenfassend wiedergeben:

Gegeben sei ein String $\tau = z_1z_2z_3...z_n$ der Länge n über einem gewichteten Alphabet $\Sigma = \{a_1, ..., a_c\}$ der Grösse c. Die Position i eines Zeichens z im String τ bezeichnen wir fortan als Index ($1 \leq i \leq n$). Der String τ hat beim Index j folglich das Zeichen z_j.

Wir definieren darauf drei Mengen: ($\subseteq$ steht für „ist ein nichtleerer Teilstring von“)

Definition F: Menge der: Teilstrings, Parikh-Vektoren, Teilmassen

- $S_{set}(\tau) := \{ \sigma \mid \sigma \subseteq \tau\}$, die Menge der verschiedenen Teilstrings von τ,
- $P_{set}(\tau) := \{mult(\sigma) \mid \sigma \subseteq \tau\}$, die Menge verschiedener Multiplizitätsvektoren (auf unter dem Namen *Parikh-Vektoren* bekannt) von τ, und
- $M_{set}(\tau) := \{\mu(\sigma) \mid \sigma \subseteq \tau\}$, die Menge verschiedener Teilmassen von τ.

Uns interessiert in erster Linie die Grösse (Kardinalität) dieser Mengen. Wir stellen die zugehörigen kombinatorischen Funktionen wie folgt dar: $S(\tau) = |S_{set}(\tau)|$ (Anzahl verschiedener Teilstrings von τ). Analog dazu: $P(\tau) = |P_{set}(\tau)|$ und $M(\tau) = |M_{set}(\tau)|$. Im genannten TR wird ein Lösungsansatz gegeben, den wir an dieser Stelle genauer untersuchen und erweitern. Dazu benötigen wir jedoch eine spezielle Datenstruktur für die Repräsentation des Eingabestrings, den sogenannten *Suffix-Tree* [1].

7.1 Suffix-Trees

Definition G: Suffix-Tree

Ein *Suffix-Tree* für einen String σ der Länge n (kurz: $ST(\sigma)$) ist ein (gerichteter) Baum mit exakt n Blättern, durchnummeriert von 1 bis n. Jeder interne Knoten (ausgenommen die Wurzel), hat mindestens zwei Kinder und jede Kante ist mit einem nichtleeren Teilstring aus σ beschriftet. Es gibt keine zwei Kanten (vom selben Knoten ausgehend), welche Beschriftungen (Labels) besitzen, die mit demselben Zeichen (Buchstaben) beginnen. Das Hauptmerkmal des *ST* ist, dass für jedes Blatt i gilt: Die Konkatenation der Kantenlabels auf dem Pfad von der Wurzel bis zum Blatt i ergibt exakt den Suffix von σ, welcher an Position i beginnt (also den Teilstring $\sigma[i..n]$).

Die Anwendungsgebiete für *Suffix-Trees* in der Stringverarbeitung sind enorm gross und vielfältig. Trotzdem führt diese Datenstruktur in der Lehre der Informatik, meines Erachtens

zu Unrecht, immer noch ein Schattendasein. Seit ihrer Entdeckung im Jahre 1973 sind viele Autoren bemüht, effiziente und einfache Implementationen zu finden. Ein Suffix-Tree kann in Zeit O(n) erstellt werden. Je nach Anwendung kann auch ein O(n^2) *worst-case* respektive ein O(n log n) *expected-case* Verfahren sinnvoll sein. Linear muss in der Praxis nicht heissen, dass es auch wirklich schneller läuft als O(n log n)-Verfahren. Implementationen in populären objektorientierten Sprachen mit aufwendiger Speicherverwaltung (Löschen nicht mehr benötigter Knoten etc.) tragen ihren Teil zu diesen Feststellungen bei.

Einen kurzen kompakten Vergleich verschiedener Methoden findet wir in [6], wobei gleichzeitig eine Implementation eines speichersparenden Algorithmus *Write-Only Top-Down (WOTD)* vorgestellt wird. Dieser Algorithmus hat eine *expected-case* Laufzeit von O(n log n). Er benötigt deutlich weniger Arbeitsspeicher als alle bis anhin bekannten linearen Verfahren.

Da wir besonders daran interessiert sind, unsere obigen drei Funktionen für möglichst lange Strings τ zu berechnen, konzentrieren wir uns auf eine speichereffiziente Implementationen wie diesen WOTD Algorithmus. Ist dieser einmalige Vorverarbeitungsschritt – das Generieren des Suffix-Trees – erledigt, können zusätzlich zu den Funktionen S, M und P vielfältigste Fragen (engl. Queries) an den String τ sehr effizient beantwortet werden. Eine Exact Pattern Matching Query nach allen k Vorkommen eines Pattern P der Länge m ist beispielsweise in O(m + k) möglich, d.h. in andern Worten, dass die Laufzeit der Anfrage komplett unabhängig von n ist! Falls wir also viele Anfragen an den einen selben Eingabestring τ stellen, fällt die Zeit für das Preprocessing nicht so stark ins Gewicht (unter der Bedingung, dass die Vorberechnungen nicht exponentielle Laufzeit haben!).

7.2 Teilstrings

Nehmen wir im folgenden an, wir hätten eine speichersparende Implementation zur Erstellung eines Suffix-Trees zur Hand. Wie können wir nun *S(τ)* berechnen?

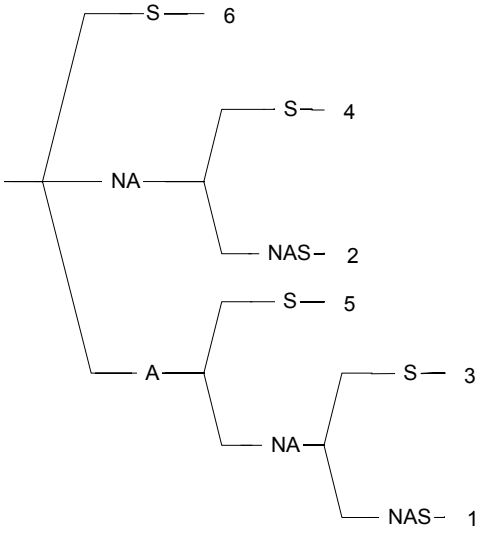

Abbildung 7-1: Suffix-Tree für den String „ANANAS“

Wir veranschaulichen uns die Vorgehensweise an einem konkreten String τ = ANANAS. Den zugehörigen Suffixbaum *ST(τ)* finden wir in Abbildung 7-1. Wir bezeichnen im folgenden Pfade von der Wurzel zu einem beliebigen Blatt als *vollständige Pfade*. Wie aus der Definition bekannt ist, entspricht somit die Menge der Konkatenationen der Labels aller

vollständigen Pfade exakt der Menge aller Suffixe des Strings τ (im weiteren als $F_{set}(\tau)$ bezeichnet).

Durch ein einfaches Traversieren des Baumes (und Besuchen aller Blätter) können wir nun alle Suffixe des Eingabestrings τ aufzählen. In unserem Beispiel ist $F_{set}(\tau)$ = {ANANAS, ANAS, AS, NANAS, NAS, S}. Die Grösse von F_{set} entspricht dabei wie nicht anders erwartet der Länge n des Eingabestrings τ.

Die Menge $F_{set}(\tau)$ ist eine Teilmenge von $S_{set}(\tau)$. Um alle Teilstrings zu erhalten, müssen wir bei der Traversierung nicht nur die vollständigen Pfade (pro memoria: Pfade von der Wurzel zu den einzelnen Blättern) aufzählen, sondern auch alle Teilpfade. Unter einem *Teilpfad* verstehen wir einen Pfad, der bei der Wurzel beginnt und nach einer beliebigen Anzahl Zeichen endet. Dieses Ende muss sich nicht exakt bei einem Konten befinden, sondern kann auch „mitten auf einer Kante“ sein. Betrachten wir den Teilbaum unter der Kante A (ausgehend von der Wurzel), ergeben sich durch Traversieren des Baumes mittels *Tiefensuche* (*preorder*)[49] folgende Teilstrings: {A, AN, ANA, ANAN, ANANA, ANANAS, ANAS, AS}. Nehmen wir die Teilstrings aus den andern beiden Teilbäumen unterhalb der Wurzel dazu, ergibt dies total 15 Teilstrings für unsern Beispielstring τ. Wir bezeichnen diesen Algorithmus zum Aufzählen aller Teilstrings im fortan mit ENUM-SUBSTRINGS (siehe Tabelle 7-1). Wir gehen zusätzlich noch davon aus, dass ENUM-SUBSTRINGS die Kinder eines Knotens des $ST(\tau)$ in alphabetischer Reihenfolge (bezüglich der Kantenbeschriftungen) besucht. Aufgrund der Definition der Kantenlabels eines Suffix-Trees, können bei ENUM-SUBSTRINGS keine zwei Teilpfade denselben Teilstring liefern.

Teilstring	**Länge**	**Startindex**	**Endindex**	**Bemerkung**
A	1	1	1	
AN	2	1	2	
ANA	3	1	3	
ANAN	4	1	4	
ANANA	5	1	5	
ANANAS	6	1	6	Suffix von τ an Index 1
ANAS	4	3	6	Suffix von τ an Index 3
AS	2	5	6	Suffix von τ an Index 5
N	1	2	2	
NA	2	2	3	
NAN	3	2	4	
NANA	4	2	5	
NANAS	5	2	6	Suffix von τ an Index 2
NAS	3	4	6	Suffix von τ an Index 4
S	1	6	6	Suffix von τ an Index 6

TOTAL	= 46	Zeichen, die ENUM-SUBSTRINGS speichern muss		

Tabelle 7-1: Teilstrings von ANANAS und ihre Eigenschaften

Eine erste Methode um $S(\tau)$ zu erhalten besteht nun einfach darin, $S_{set}(\tau)$ zu berechnen und anschliessend die Grösse der entstandenen Menge als Resultat zurückzuliefern. Verwenden wir dazu ENUM-SUBSTRINGS, beansprucht dies allerdings enorm viel Speicherplatz für das Zwischenspeichern aller Teilstrings. Offensichtlich gilt $n \leq S(\tau) \leq n \cdot (n+1)/2$. Unter der

[49] Jede Kante wird dabei Zeichen für Zeichen abgearbeitet!

Annahme, dass jeder String der Länge n Platz $\Theta(n)$ benötigt, ergäbe dies im schlechtesten Fall[50] einen Gesamtspeicherbedarf von $\Theta(n^3)$.

Anstatt alle Strings komplett zu notieren, können wir uns auch nur Start- und Endindex (respektive Start- und Endposition) des Teilstrings im Bezug auf den ursprünglichen Eingabestring τ abspeichern. Damit sink der Speicherhunger schon einmal auf $\Theta(S(\tau))$ respektive $\Theta(n^2)$.

Um nun noch weitere Arbeitsspeicher zu sparen, machen wir uns eine spezielle Eigenschaft der Suffix-Trees und unseres ENUM-SUBSTRINGS Algorithmus zu Nutze. Wenn wir die Menge der Teilstrings von τ betrachten, entdecken wir, dass viele durch das Anfügen eines weiteren Zeichens auseinander hervorgehen: {A, AN, ANA, ANAN, ...}. Betrachten wir die dazugehörigen Start- und Endindexpaare {(1,1), (1,2), (1,3), (1,4), ...} bemerken wir, dass bei den aufeinanderfolgenden Paaren der Startindex identisch ist und der zugehörige Endindex stets um eine Position inkrementiert wird (siehe erneut Tabelle 7-1). Wir könnten uns daher die Startposition auch nur einmal merken und anschliessend die verschiedenen Endindizes (siehe Tabelle 7-2). Der Startindex ist dabei identisch mit der Bezeichnung des zugehörigen Blattes im Suffix-Tree (siehe Definition des Suffix-Tree und Abbildung 7-1).

Start-index	**Endindex 1**	**Endindex 2**	**Endindex 3**	**Endindex 4**	**Endindex 5**	**Endindex 6**
1	1	2	3	4	5	6
2	2	3	4	5	6	-
3	6	-	-	-	-	-
4	6	-	-	-	-	-
5	6	-	-	-	-	-
6	6	-	-	-	-	-

Tabelle 7-2: Start- und Endindizes der Teilsstrings von τ in der Übersicht

Da wir aber wissen, welche Startindizes überhaupt je vorkommen können (nämlich alle Blätter des Suffix-Trees 1..n) können wir eine komprimierte Darstellung obiger Matrix einführen. Wir erzeugen ein Array X mit Länge n. Weil für jeden Startindex eine kontinuierliche Folge von Endindizes vorliegt[51], würde es reichen, pro Startindex nur das Intervall der Endindizes zu speichern. Im Falle von Startindex *2* wäre dies beispielsweise das Intervall [2..6], für Startindex *5* das Intervall [6..6]. Weil jedes dieser Intervalle stets denselben Endindex besitzt (in unserem Beispiel 6), reicht es, die nur Länge des Intervalls zu speichern – die restliche Information ist implizit enthalten.

1		***2***		***3***		***4***		***5***		***6***		**begin**
6	*3*	5	*4*	1	*5*	1	*6*	1	*2*	1	-	*1*

Tabelle 7-3: Darstellung der Teilstrings von τ in komprimierter Arrayform

Damit wir diese alphabetische Reihenfolge beim Abspeichern der Intervallgrenzen nicht verlieren, merken wir uns im Array X während der Tiefensuche (zusätzlich zur Intervalllänge) noch einen Pointer zum nächsten besuchten Blattknoten. Ebenso merken wir uns den Einstiegspunkt (*begin*), welches Blatt wir als erstes besucht haben. Jetzt können wir den Suffix-Tree löschen.

[50] Dies ist der Fall, wenn die Grösse des Alphabets n entspricht. Nur dann sind alle Teilstrings verschieden.
[51] Diese Tatsache ergibt sich direkt aus dem ENUM-SUBSTRINGS Algorithmus.

Wollen wir schliesslich alle Teilstrings (ausgehend von *X*) aufzählen, beginnen wir beim zuerst besuchten Blattknoten und Durchlaufen Array *X* gemäss den Pointern. In unserem Beispiel ist dies die Reihenfolge {*1, 3, 5, 2, 4, 6*}. Pro Element (bzw. Suffix-Tree Blatt) *i* zählen wir dabei alle Teilstrings für den Startindex *i* und die Endindizes *n*-*X*[i]+1 bis *n* auf.

Bei der Komplexitätsanalyse sehen wir, dass unsere Datenstruktur *X* (im Vergleich zum Suffix-Tree) asymptotisch immer noch linearen Platz O(2*n* + 1) = O(*n*) benötigt. Dies ist nicht besser als die speichereffiziente WODT Implementierung. Reicht uns die Information über die vorhandenen Teilstrings eines Strings τ, können wir, im Gegensatz zum WODT Suffix-Tree, die Konstante vor *n* um einen Faktor 6 reduzieren.

Um schliesslich *S(τ)* zu berechen, summieren wir sämtliche Intervalllängen im Array *X* auf. Weil für das Zusammenzählen keine Pointer benötigt werden, können sie in diesem Fall weggelassen werden, um nochmals etwas Speicher zu sparen.

Sind wir allerdings nicht angewiesen $S_{set}(\tau)$, sondern nur *S(τ)* zu berechen, gibt es eine noch viel schnellere Methode, ohne das Array *X* überhaupt aufzubauen. Es reicht dabei aus, die Längen der Beschriftungen aller Kanten von *ST(τ)* aufzuaddieren. Die entstandene Summe entspricht genau *S(τ)*. Der Beweis ergibt sich aus der Definition des Suffix-Trees. Gibt es in einem String τ mehrere Vorkommen des selben Teilsstrings σ, so werden alle auf den gleichen Teilpfad in *ST(τ)* abgebildet. Die identischen Teilsstrings unterscheiden sich lediglich durch den Startindex, wo sie in String τ erscheinen.

7.3 Multiplizitätsvektoren

Das erste Teilproblem gelöst ist, können wir uns dem zweiten widmen: der Berechnung von *P(τ)*. Wir entnehmen aus [5] eine weitere Tatsache, dass folgende offensichtliche Beziehung $M(\tau) \leq P(\tau) \leq S(\tau)$ gilt. Aus dieser Relation und der Definition von Multiplizitätvektoren ist es naheliegend, $P_{set}(\tau)$ aus $S_{set}(\tau)$ herzuleiten. Wir berechen dazu die Parikh-Vektoren für jeden Teilstring, ordnen diese und löschen eventuell entstandene Duplikate bis auf einen Repräsentanten. Dieser Vorschlag wie er in [5] kurz erwähnt wird, bedarf nun allerdings einer geschickten Verwaltung der Vektoren, um nicht dasselbe Schicksal des Speicherüberlaufs wie bei den Teilstrings zu erfahren.

Eine erste triviale Variante speichert für jeden Teilstring aus τ einen c-elementigen Vektor (pro memoria: $c = |\Sigma|$). D.h. wir benötigen im $c \cdot P(\tau)$ Platz → O(*P(τ)*). Die Laufzeit beträgt O(*P(τ)*·log *P(τ)*). Der logarithmische Faktor rührt übrigens vom Sortieren der Vektoren her und nicht etwa von der Wahl der Suffix-Tree Implementation, die wir in unseren Analysen stets als linear in *n* betrachten. Die bei diesem Vorgehen entstehende Menge ist $P_{set}(\tau)$.

Leider können wir bei den Parikh-Vektoren für einen Teilstring τ – im folgenden mit *PV*(τ) bezeichnet – nicht so einfach Intervalle definieren, wie dies bei den Teilstrings möglich war. Wir können allerdings wiederum die Ähnlichkeit zwischen den „benachbarten" Teilstrings der geordneten Menge $S_{set}(\tau)$ nutzen, um die Vektoren innerhalb eines Teilbaumes zumindest aus den vorhergehenden abzuleiten. Dies bringt erhebliche Gewinne bei der Laufzeit und bei der Speicherplatzbeanspruchung. Für die Differenz zwischen zwei Parikh-Vektoren für „benachbarte" Teilstrings (beispielsweise *PV*(„ANA") und *PV*(„ANAN")) kann anstelle eines *c*-dimensionalen Differenzen-Vektors – in unserem Beispiel der Vektor *PV*(„N") der Form (0,...,0,1,0,...0) – auch nur eine Zahl abgespeichert werden. Dies ist der Alphabetindex des Zeichens „N".

Wir beenden mit dieser Erkenntnis die Untersuchung zur Abspeicherung von Parikh-Vektoren.

7.4 Teilmassen

Die letzte der drei Fragestellungen ist schliesslich nur noch Formsache. Sind einmal alle Multiplizitätsvektoren des Eingabestrings τ aufgezählt, kann für jeden die Gesamtmasse berechnet werden und die gefundenen Werte in einer Variablen *s* vom Datentyp *Set*[52] einfügt werden (beispielsweise in einen Binärbaum). Ist dies für alle Vektoren vollbracht, entspricht der Inhalt der Variablen *s* exakt $M_{set}(\tau)$. Für den Wert von $M(\tau)$ wird gemäss Definition die Anzahl Elemente dieser Menge zurückgegeben. Dieser Algorithmus zur Berechung aller Telmassen läuft in Zeit $O(P(\tau) + M(\tau) \cdot \log M(\tau))$ und das Resultat benötigt $O(M(\tau))$ Platz.

Alternativ dazu, könnten wir die Menge $M_{set}(\tau)$ auch direkt aus der Menge $S_{set}(\tau)$ (respektive unserem Array *X*) berechen.

7.5 Implementation

Die vorher erläuterten Algorithmen sind, bis auf einen[53], alle implementiert. Sämtliche Implementationen machen dabei Gebrauch von der neuentwickelten Datenstruktur, dem Array *X*. Für den Suffix-Tree wurde eine existierende Java-Implementation verwendet. Näheres zu den Implementationsdetails findet der Leser in [7].

[52] Beim Datentyp Set existiert von jedem Element (in unserem Fall Teilmassen) nur eine Instanz.
[53] Es handelt sich dabei um die Implementation der speichereffizienten Verwaltung von Parikh-Vektoren in Kapitel 7.3.

8 Weiterführende Themen

STRATO ist ein abgeschlossenes Projekt. Trotzdem kann an verschiedenen Stellen noch weitergeforscht und programmiert werden. Dabei ist nicht von der Implementierung neuer Tests oder neuer Stringerzeugungsmechanismen die Rede. Vielmehr meinen wir damit den Ausbau des Frameworks selbst. Das Kapitel versucht einige möglicher Erweiterungen zu thematisieren um zumindest Anregungen zu schaffen, wie der Kern der Software weiter ausgebaut werden könnte.

8.1 Weitere Testarten

Beim Implementieren unzähliger neuer Tests kann durchaus auch die Frage aufkommen, ob das Framework wirklich jede denkbare Art von Tests vollständig zu unterstützen vermag. Bis anhin jedenfalls wurde noch keine entdeckt, die nicht umzusetzen wäre, ohne das Framework selbst zu verändern. Einzig im Falle von Testparametern könnte vielleicht der Gedanke auftauchen, dass ausser den zur Zeit unterstützten Basistypen (*int*, *double*, *boolean*, etc.) und der Klasse *java.lang.String* auch noch andere Typen vonnöten wären. Dabei spielt nebst der internen Verwaltung dieser Datentypen (oder Klassen) auch die Interaktion zwischen dem Benutzer, der die Parameter manipulieren will, und der GUI, welche die Modifikation der Parameter ermöglichen soll, eine zentrale Rolle. Zur Zeit ist es nur möglich, die Parameterwerte in ein Textfeld einzugeben. Die Stringrepräsentation der Daten wird dabei von oder in den gewünschten Typ mittels Methoden der sogenannten *Wrapperklassen*[54] konvertiert. Soll ein Parameter ein komplexeres Gebilde wie ein Baum, eine Matrix oder was ähnliches darstellen, so muss auch das GUI eine entsprechende Schnittstelle bereitstellen, um diesen darzustellen und zu verändern. Eine Repräsentation als simpler Javastring fällt hier aus Benutzerfreundlichkeitsgründen wahrscheinlich weg.

8.2 Hilfstool für das Schreiben von Tests

Ein Zusatztool für die halbautomatische Sourcecode-Erzeugung eines Tests, könnte vielen Programmierneulingen helfen, selbstständig einen Test zu implementieren. Dieses Hilfstool würde dem Programmierer (z.B. mittels Dialogboxen) eine Reihe von Fragen zum Test stellen. Beispiele hierzu sind: Wie heisst der Test? Wie viele Parameter besitzt der Test? usw. Aus den Antworten würde „automatisch“ das Gerüst eines Tests erstellt werden – insbesondere die Implementation des *ITestInfo* Interfaces. Der Programmierer könnte sich also voll auf das Implementieren des Algorithmus selbst konzentrieren, ohne zusätzlich noch alle nötigen Deskriptoren von Hand ausprogrammieren zu müssen.

8.3 Mehrere gleichzeitige Inputs

Zur Zeit (STRATO Version 1.0) wird genau ein einziger Eingabestring pro Testausführung verarbeitet. Wie wäre es, wenn nun mehrere Inputs gleichzeitig erlaubt sind? Erste Versuche auf Kommandozeilenbasis zeigen eine Vorstufe davon. Dabei wird dem Tests weiterhin pro Durchgang nur eine Eingabe präsentiert, dafür aber mehrere hundert oder gar Tausende nacheinander. Die untereinander verschiedenen Eingabedaten sind dabei vorgängig

[54] Für jeden Basistyp gibt es eine Wrapperklasse, die es ermöglicht, die Daten als unveränderliche Objekte zu verwalten: z.B. Die Klasse *Integer* für den Datentyp *int.*

automatisch oder von Hand erzeugt und in ein File abgespeichert worden. Dieses wird nun eingelesen und liefert dabei Input für Input an den Test.

Eine nächste Stufe sieht nun vor, das Axiom aufzugeben, nur einen Eingabestring pro Durchgang zu verarbeiten. In erster Linie würden zwei Eingänge einen ganz neuen Zweig von Untersuchungen einläuten. *Editierdistanzen* und andere Untersuchungen, die zwei Strings miteinander vergleichen, könnten implementiert werden.

Noch einen Schritt weiter ginge die Idee, dass die Ausgabe eines Tests gar wieder als Input für einen anderen Test verwendet wird. Sei dies nun als berechneter Eingabestring oder gar als Parameter. Abbildung 8-1 versucht das in Kapitel 4 entwickelte Architekturmodell Eingabe / Verarbeitung / Ausgabe an einem möglichen Beispielsszenario mit drei Haupteingängen und einem Hauptausgang zu generalisieren.

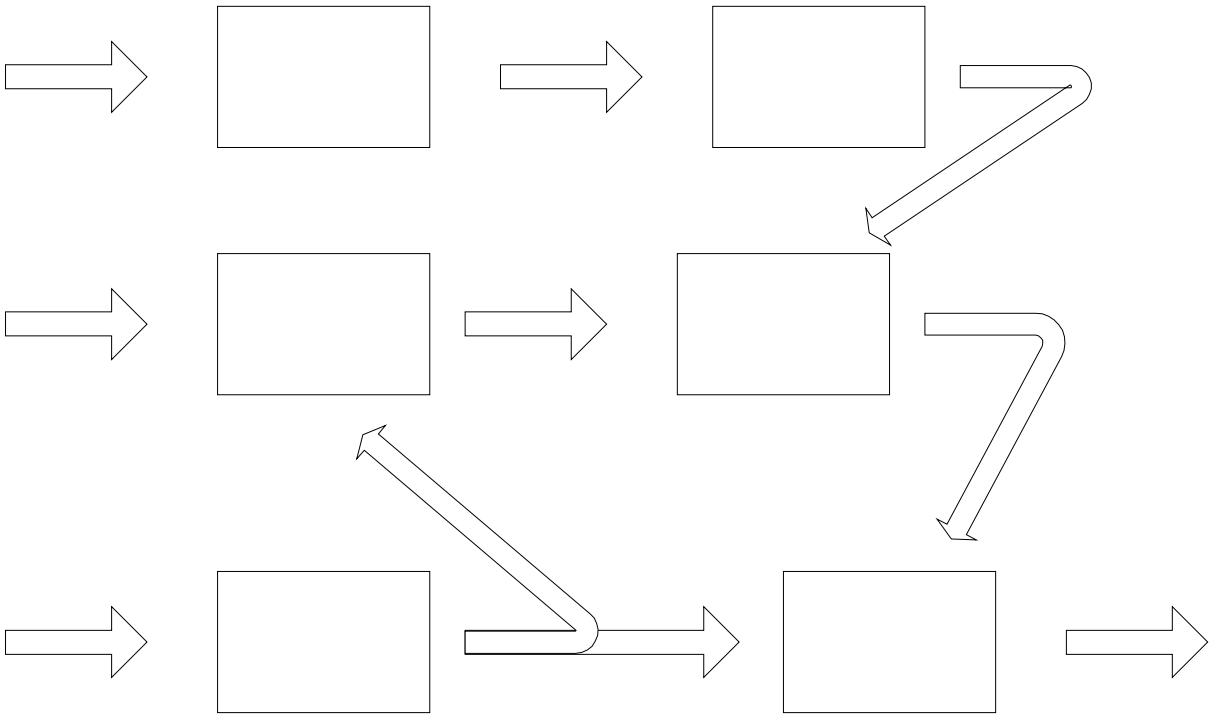

Abbildung 8-1: Mögliche Vernetzungsvariante mehrerer Eingänge und eines Ausgangs

Die Verknüpfungsmöglichkeiten von Eingängen, Ausgängen und Parameter-Eingängen wäre in schier beliebiger Vielfalt möglich.

9 Persönliche Bemerkungen

Die folgenden Unterkapitel versuchen den Ablauf der Diplomarbeit, die erzielten Resultate und die gewonnene Erfahrung aus verschiedenen Blickwinkeln zu betrachten und werden schlussendlich durch ein Gesamturteil abgerundet. Die wiedergegebenen Aussagen sind sehr subjektiver Natur (aus der Sicht des Diplomanden) und daher teils in ICH-Form niedergeschrieben.

9.1 ...zur Programmieraufgabe

Einleitend ein paar objektive Statistiken zum STRATO Tool, die für sich sprechen sollten.

Reiner Programmieraufwand:

- Anzahl Klassen: > 120
- Anzahl Zeilen Code: > 15'000
- Coding: ~ 450h

In der Literatur finden sich immer wieder Übersichten und statistische Werte, wie lange denn der Durchschnittsprogrammierer für ein korrekt funktionierendes (hoffentlich optimiertes) Programm von *N* Zeilen Code benötige. In der Industrie sind 5-10 Zeilen Code pro Stunde ein gebräuchlicher Richtwert. Dabei eingerechnet sind sämtliche Arbeitsschritte vom ersten Design, über die Implementierung bis zur Testphase und dem Bereinigen der Software von lästigen Bugs. Rechnen wir diesen „Pro-Stunde-Wert“ für unser Projekt aus, so ergibt dies einen Wert von mehr als 30 Zeilen pro Stunde. Dieser relativ hohe Wert erlaubt nun zwei Interpretationen, deren zweite mir nicht ganz unwichtig erscheint. Einerseits gehöre ich wohl der Gruppe der etwas flinkeren Programmierer an. Andererseits muss aber ganz klar festgehalten werden, dass diese Diplomarbeit ganz spezielle Eigenschaften mit sich bringt. Einige der Teilprobleme sind solche, die schon in einem andern Kontext gelöst wurden und daher hauptsächlich in die Kategorie „ToDo“ fallen. Je mehr Erfahrung ein Informatiker mitbringt, desto schneller löst er auch solche Standardaufgaben oder passt vorhandene Ressourcen an ein neues Problem an. Insbesondere sind Mechanismen, welche die graphische Oberfläche (GUI) betreffen, eigentlich reine Formsache – aber doch gehören sie vollumfänglich zu den zu erledigenden Tasks.

Doch zurück zur Implementation als Ganzes. Dieses Tool vereint in sich sehr viele Design-Pattern. Ich habe absichtlich nicht mal eines im Ansatz erklärt, da dies den Rahmen dieser Arbeit bei weitem sprengen würde und sehr gute Fachliteratur im Handel ist. *MVC/Observer*, *Strategy*, *Template*, *Singleton* und *Prototype* [2] sind ein paar Vertreter aus dieser Gilde. Auch Ideen aus der Ecke der Komponenten (JavaBeans™, COM™, ...) leisteten ihren Beitrag. Gerade diese facettenreiche Komposition verschiedenster Techniken, die seit langen in vielen Applikation erfolgreich verwendet werden, verhalfen in meinen Augen zum Erfolg dieser Diplomarbeit. STRATO ist zu einem Tool gereift, das sich weitflächig (im Rahmen des Anwendungsgebietes, das ihm zu Grunde liegt) erweitern lässt. Beispielsweise können selbst Teile der GUI dynamisch (nach-)geladen werden.

Meines Erachtens ist es gelungen, ein Framework zu schaffen (zwar sehr spezifisch im Einzelnen, dafür aber in ihrem Kontext extrem allgemein), das sich sehr einfach benutzen und erweitern lässt. Sollte sich das Tool definitiv durchsetzen und um viele weitere Tests ergänzt werden, würde es mich natürlich freuen.

9.2 ...zum Ablauf der Diplomarbeit

Damit diese über 15'000 Zeilen Code überhaupt erst entstehen konnten, waren noch ganz andere Arbeiten vonnöten. Sie setzten sich aus den gängigen Sparten zusammen:

- Ideen, Konzepte und Designvorschläge entwickeln und evaluieren;
- bei Meetings diese besprechen und anschliessend das Beste daraus wählen;
- eine ausführliche Literaturrecherche (Bücher, WWW, ...) betreiben;
- die vorliegende schriftliche Arbeit schreiben und den Schlussvortrag planen.
- usw.

Die Einarbeitung bestand hauptsächlich aus intensiver Lektüre von Literatur zum Thema Strings. Doch schon bald gewann ich einen Einblick in dieses spannende Thema und begann mit der Implementation des Grundgerüstes meiner Software. Nebenbei entstanden bereits erste Prototypen einer Benutzeroberfläche. Alles immer im Stile von Extreme Programming, zu welchem ich in Kapitel 9.4 noch ein paar Worte verlieren werde. Doch stets mit dem Hintergedanken im Kopf, nicht allzu viel Zeit für das Design einer Benutzeroberfläche zu verschwenden, wurde nach und nach Funktionalität hinter diesem *Frontend* programmiert. Nach rund 3 Monaten war ein „Produkt“ entstanden, das bereits sämtliche geforderten Features (inklusive fast aller optionalen) auswies. Um schliesslich die restliche Zeit noch sinnvoll zu nutzen, habe ich, wie in der Einleitung erwähnt, mich noch einem andern Thema gewidmet (siehe Kapitel 7). Nachdem ein kleiner Exkurs zu Neuronalen Netzen leider nicht die gewünschten Ergebnisse brachte, befasst ich mich etwas intensiver mit dem Suffix-Tree.

Wenn man diesen vorherigen Abschnitt so liest, könnte man fast glauben, dass alles total reibungslos lief und keinerlei Probleme auftragen. Doch weit gefehlt! Stundenlanges Nachdenken zu diversen Fragestellungen und ein an stets neue Anforderungen angepasstes Design forderten ihren Tribut, dass sich mein Arbeitstag manchmal bis in die späten Abendstunden oder gar bis Mitternacht hinzog. Im Gegenzug gab’s aber auch Probleme, die nicht einmal die Hälfte der für sie reservierten Zeit in Anspruch nahmen, was dann auch dafür sorgte, dass ich von Anfang bis am Schluss eigentlich immer im Zeitplan lag.

9.3 ...zu meinen Erwartungen an die Benutzer des Tools

Bevor die vorliegende Software STRATO der freien Welt übergeben wird, sollten noch einige Dinge klargestellt werden. Der Diplomand/Autor/Programmierer dieser mittlerweile deutlich über 100 Klassen und Interfaces hofft, dass sein Projekt in einer gewissenhaften Art weiterhin so geführt und gepflegt wird, wie es vom ersten Tag an gemacht wurde. Sei dies durch Einhaltung von „Coding Conventions“ oder bloss dem Dokumentieren von public deklarierten Methoden und Variablen. Gerade hier empfiehlt es sich die JAVADOC Richtlinien[55] zu verwenden. Dies ist keine Vorschrift, aber so wird meist das Wichtigste nicht vergessen.

Ich habe mir während der letzten vier Monate ein paar goldene Regeln zusammengestellt, die ich stets befolgt habe und hiermit gerne weitergebe. Es geht dabei nicht darum, den Programmierer zu schikanieren, sondern bloss, ihm zu helfen, und vor allem jenen, die dessen Arbeit später einmal verwenden und/oder erweitern. Denn alle Personen, die in ein Projekt eintreten, müssen sich innert kürzester Zeit mit der schon bestehenden Software vertraut machen. Es folgt daher eine Auflistung von *DOs & DON’Ts*, deren Einhaltung erbeten ist und sich in den letzen vier Monaten als wichtig herauskristallisiert hat!

[55] Sozusagen der Quasi-Standard, um Klassen und ihre Funktionalität zu dokumentieren. Aus diesem Grund ist auch eine breite Javaprogrammierer Gilde damit vertraut. Näheres unter http://java.sun.com/j2se/javadoc/

Kategorie *Programmieren*
- Halte Dich an die „Coding Conventions“.
- Dokumentiere wo nötig deinen Code (siehe nächste Kategorie)

Kategorie *Dokumentation*
- Schreib alles Wichtige auf, was eventuell nur in Deinem Geist existiert und nicht aus dem Sourcecode ersichtlich ist.
- Es reicht die Stellen im Code zu dokumentieren, die *public* sind. Falls jemand die privaten Internas nicht versteht, wird er sie auch nicht anrühren.
- Dokumentiere alles, das in irgendeiner Form als erweiterbar gedacht ist (Code: Abstrakte Klassen, Interfaces; Listen: Tabellen, Konventionen, ...) in minuziösem Detail, ansonsten wird es niemand verwenden oder gar erweitern.

Kategorie *Allgemein*
- Das Programmieren muss Spass machen ☺
- Dieses Tool ist für Forscher aller Arten und Richtungen gedacht und nicht nur für eine eingefleischte Java-Programmierer-Community. Jeder noch so kleine Beitrag an das STRATO Projekt stellt daher ein Bereicherung dar.
- Neuhinzugefügte String-Analyse-Funktionen (Tests) oder String-Erzeugungs-Mechanismen (SEMs) müssen nicht höchsten Performancetests genügen. Sie sollten in einem ersten Schritt lediglich korrekt sein. Laufzeitverbesserungen und speicherplatzoptimierende Massnahmen werden erst in einem zweiten Schritt erwartet.

9.4 ...zu diversen Themen (Extreme Programming, Assistenten, ...)

Extreme Programming (XP) [3] ist ein Schlagwort in aller Munde, wenn es um neue professionelle Softwareentwicklungsmethoden für Teams zwischen 2-10 Leuten geht. Termingerecht eine Applikation im geforderten Umfang abzuliefern, war nur einer der Gründe, wieso sich immer mehr Leute darüber Gedanken machen, wie bekannte Projektentwicklungsrichtlinien (Wasserfallmodell, Rational Unified Process (RUP), ...) verbessert werden könnten. Dass dabei die Qualität nicht auf der Strecke bleiben sollte ist in gewissem Masse selbsterklärend. Zumal auch bei einer Diplomarbeit die drei Variablen eines jeden Projektes (Zeit, Qualität und Umfang)[56] alle voneinander abhängig sind, lassen sich die Prinzipen von XP auch auf Ein-Mann-Diplomarbeits-Projekte anwenden. An dieser Stelle jetzt über die Vor- und Nachteile von XP zu philosophieren, ist jedoch nicht Gegenstand dieser Arbeit. Für jene die sich mehr dafür interessieren, finden eine gute Einführung in XP in Büchern seines Erfinders Kent Beck.

Wer sich nun fragt, was dieser vorangegangene kurze Abschnitt mit dem STRATO Projekt zu tun hat, bekommt eine kurze aber prägnante Antwort. STRATO wurde *nicht* strikte nach XP produziert, versucht aber, die im Rahmen dieser Diplomarbeit sinnvollen Aspekte dieser neuen und erfolgsversprechenden Entwicklungstechnik umzusetzen. Der Entscheid total auf die althergebrachten Phasen eines Informatikprojekts zu verzichten und sich von Anfang an in die Programmierung zu stürzen, ist sicher nicht hinderlich, wenn wöchentlich entschieden wird, was denn als nächstes zu tun sei. Oder um es anders zu formulieren: Assistenten sind manchmal die anspruchvollsten Kunden bezüglich Wünschen und insbesondere regen Änderungswünschen an eine neu entstehende Software! Im Gegenzug bringt es auch dem Diplomanden eine gewisse Befriedigung, wenn er sieht, dass die neuhinzugefügten

[56] Eine Vierte wäre noch die „finanziellen“ Kosten, die man als Diplomand aber nach Erhalt des Diploms getrost auf Null setzen kann!

Funktionen in den beinahe wöchentlichen durchgeführten Demos als gut befunden werden. Somit kann jeder wöchentliche Release auch als Endversion mit nicht hundertprozentigem Umfang bezeichnet werden. Falls sich das Projekt einmal in eine Sackgasse bewegen sollte, kann jederzeit auf eine alte Version zugegriffen werden und ein neuer Weg eingeschlagen werden, ohne dabei wieder bei Null zu beginnen. Denn nur wer bereit ist, auch einmal ein Stück Programmcode wegzuwerfen, holt das Bestmögliche aus der Sache heraus!

9.5 Schlussbemerkungen

Der Trend *alles einfach*, um nicht zu sagen *geradezu simpel* zu gestalten, hält derzeit immer noch an. In Buchläden reihen sich Bände mit Titeln *Windows™ für Dummies*, *Java™ in 21 Tagen* usw., dass man meinen könnte, das Informatikstudium liesse sich problemlos auf ein Jahr kürzen. Dass dies nicht möglich ist, dazu brachen wir hier keine Worte zu verlieren. Glücklicherweise konnte die Komplexität des umfangreichen STRATO-Projektes relativ gut unter Kontrolle gehalten werden. Sind die verwendeten Prinzipien einmal klar verstanden, steht Dutzenden von neuen Tests (Plug-ins) und anderen *Add-ons* nichts mehr im Wege.

Die ursprüngliche Aufgabenstellung der Diplomarbeit war sehr offen formuliert (siehe Ausschreibung im vordersten Teil dieser Arbeit). In zahlreichen Meetings wurden im Verlaufe der Zeit die Einzelheiten (meist in spannenden Diskussionen) erarbeitet. Ich war anfangs nicht besonders glücklich, dass die Aufgabenstellung so generisch verfasst wurde. Rückblickend sehe ich aber die Tatsache, die konkreten Teilaufgaben mitgestalten zu können, als besonders wertvolle Erfahrung an. Wenn nicht alles bis in letzte Detail vorgegeben ist, lässt dies viel mehr freien Spielraum für gute und spannende Ideen.

10 Anhang A

10.1 Referenz: Conventions

Grundsätzlich verwende man die Standard Java Konventionen, welche unter http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html zu finden sind. Es folgen nun die wichtigsten Punkte daraus und ein paar kleine Abweichungen, die vor allem daher eingeführt wurden, um Namenskonflikte zwischen Klassen im STRATO Projekt und denen der Standard-Bibliothek zu vermeiden.

Namensgebung

- Klassenpakete (*package*) werden durchgehend klein geschrieben
- Klassenbezeichner (*class*) beginnen mit einem Prefix C und dann 1. Buchstabe gross
  - Ausnahme: GUI Komponenten haben den Prefix STRATO
- Interfacebezeichner beginnen mit dem Prefix I und dann 1. Buchstabe gross
- Klassen, welche die Funktion eines struct in C/C++ resp. eines RECORD in Oberon einnehmen, beginnen mit dem Prefix T und dann 1. Buchstabe gross
- Methoden beginnen stets mit einem Verb (1. Buchstabe klein); jedes weitere „innere“ Wort ist grossgeschrieben
- Variablen haben **alle** sinnvolle Namen (ohne Unterstriche). Ausnahme: Zähler von Schleifen oder ähnliches
- Konstanten sind durchgehend mit Grossbuchstaben geschrieben. Mehrere Worte umfassende Bezeichner sind durch '_' zu trennen
- Alle verwendeten Bezeichner entspringen der englischen Sprache

Dateistruktur

- Pro Datei gibt es „normalerweise“ nur eine Klasse
  - Ausnahme 1: Lokale Hilfsklassen werden ins gleiche File geschrieben
  - Ausnahme 2: Innere Klassen sind stets erlaubt
- Aufteilung der Klassenfiles in Packages erhöht die Übersichtlichkeit und gehört zum guten Ton.

Kommentare

- Kommentare werden in /** ... */ eingeschlossen und für JAVADOC Interpreter mit Information gefüllt. Es können und sollen auch mehrzeilige Kommentare verfasst werden.
- Innerhalb von Methoden wird mittels C++ Style kommentiert ➔ // any comment
- Alle *public* Methoden müssen dokumentiert werden ➔ /** ... */
- Es wird ausschliesslich in Englisch dokumentiert.

Darstellung – Layout des Codes

- Linien sollten maximal 80 Zeichen lang sein und sinnvoll umgebrochen werden.
- Bei Kontrollstrukturen wie if-then-else, while, ... wird die öffnende Klammer '{' auf dieselbe Line geschrieben wie das Schlüsselwort, die schliessende Klammer in eine eigene Zeile.

10.2 Verzeichnisse

10.2.1 Abbildungsverzeichnis

Abbildung 4-1: Das 3-Schichtenmodell: Eingabe, Verarbeitung, Ausgabe 8
Abbildung 4-2: Verschiede mögliche Quellen für die Eingabe 9
Abbildung 4-3: Dateiformat eines Eingabestrings 18
Abbildung 4-4: Eine Beispieldatei für unser Dateiformat 18
Abbildung 4-5: Basismodell einer Test-Box 19
Abbildung 4-6: Erste Ausbaustufe des Basismodells durch Einführung von Parametern 20
Abbildung 4-7: Finale Version der Test-Box mit integrierter Plug-in Funktionalität 21
Abbildung 4-8 Das berechnete Resultat 21
Abbildung 4-9: Gesamtübersicht zur Architektur von STRATO 22
Abbildung 5-1: UML Diagramm der Kernkomponenten CString, CAlphabet, IString 26
Abbildung 5-2: Der Datenfluss zwischen den einzelnen Programmkomponenten 28
Abbildung 5-3: UML Klassendiagramm der Deskriptoren 34
Abbildung 5-4: Interface zur Veröffentlichung der Funktionalität eines Tests 34
Abbildung 5-5: UML Klassendiagramm aller in STRATO 1.0 verfügbaren Tests 38
Abbildung 5-6: Grafische Ausgabe eines Testsresultates 46
Abbildung 5-7: Snapshot: Die STRATO Applikation 48
Abbildung 5-8: UML Sequenzdiagramm des Ablaufs beim Starten eines Tests 50
Abbildung 6-1: Die drei Hauptkomponenten: Eingabe-, Ausgabe- und Testbereich 53
Abbildung 6-2: Snapshot: Dialogbox zur Änderung der Gewichte 54
Abbildung 6-3: Snapshot: Globale Attribute sowie die Buttons der Eingabe per Hand 55
Abbildung 6-4: Snapshot: Zufällige Stringerzeugung 55
Abbildung 6-5: Snapshot: Erzeugung aus regulärem Ausdruck 55
Abbildung 6-6: Snapshot: Erzeugung aus Regel 56
Abbildung 6-7: Snapshot: Auslesen der Eingabe aus einer Datei 56
Abbildung 6-8: Snapshot: Dialogbox zur Änderung von Einstellungen 57
Abbildung 6-9: Snapshot: Die Toolbar zum Starten, Stoppen und Unterbrechen von Tests 57
Abbildung 6-10: Snapshot: Generelle Informationen zum Test 58
Abbildung 6-11: Snapshot: Anzeige des Sourcecodes eines Tests 59
Abbildung 6-12: Snapshot: HTML-formatierte Ausgabe einer Liste von Werten 60
Abbildung 6-13: Snapshot: Menü zur Manipulation von Eingabedaten 60
Abbildung 7-1: Suffix-Tree für den String „ANANAS“ 64
Abbildung 8-1: Mögliche Vernetzungsvariante mehrerer Eingänge und eines Ausgangs 70

10.2.2 Tabellenverzeichnis

Tabelle 4-1: Liste der vordefinierten Alphabete 11
Tabelle 5-1: Vorhandene Varianten des BINSEARCH Tests 45
Tabelle 5-2: Vorhandene Varianten des Statistik Tests 47
Tabelle 6-1: Die veränderbaren Attribute geordnet nach Erzeugungsart 55
Tabelle 6-2: Formatierungsregeln für Basistypen 56
Tabelle 6-3: Ein Auflistung aller Menüs (im Applikationsmodus) 61
Tabelle 7-1: Teilstrings von ANANAS und ihre Eigenschaften 65
Tabelle 7-2: Start- und Endindizes der Teilsstrings von τ in der Übersicht 66
Tabelle 7-3: Darstellung der Teilstrings von τ in komprimierter Arrayform 66

10.2.3 Beispielverzeichnis

Beispiel I: Ein gewichtetes Alphabet 10
Beispiel II: Ein einfacher String 10
Beispiel III: Spezielle Arten von Teilstrings 10
Beispiel IV: Ein verschachtelter Ausdruck 15
Beispiel V: Komplexe Exponenten → kein Problem 15
Beispiel VI: Spezialzeichen mit führendem Escape '\' oder als Sequenz "..." 16
Beispiel VII: Ein Test zu Exact Pattern Matchings 18
Beispiel VIII: Ein Test mit gewichteten Strings 19
Beispiel IX: Ein Test zur Erkennung komplementierter Palindrome 19

10.2.4 Verzeichnis der Definitionen

Definition A: Alphabet, Gewichtsfunktion, Gesamtgewicht, Teilmasse 10
Definition B: String 10
Definition C: Teilstring, Prefix, Suffix 10
Definition D: Palindrom 10
Definition E: Komplementiertes Palindrom 11
Definition F: Menge der: Teilstrings, Parikh-Vektoren, Teilmassen 63
Definition G: Suffix-Tree 63

10.2.5 Verzeichnis der Listings

Listing 1: Interface IString 27
Listing 2: Interfaces IAccess, IInput und IOutput 28
Listing 3: Interface IPersistence 30
Listing 4: Interface IStreamable 31
Listing 5: Beispielprogramm zur Reflection API 33
Listing 6: Die Klasse CParameterDescriptor 37

Listing 7: Beispieltest Teil I: Importklauseln und Konstruktor 39
Listing 8: Beispieltest Teil II: Beschreibung des Tests 40
Listing 9: Beispieltest Teil III: Die verwendeten Parameter 40
Listing 10: Beispieltest Teil IV: Variable, Setter- und Getter-Methoden 40
Listing 11: Beispieltest Teil V: Angebotene Algorithmenvarianten 41
Listing 12: Beispieltest Teil VI: ...und ihre Signatur 41
Listing 13: Beispieltest Teil VII: Variablen zur Speicherung vorverarbeiteter Daten 41
Listing 14: Beispieltest Teil VIII: Ist das Preprocessing nötig? 42
Listing 15: Beispieltest Teil IX: Der Preprocessing Algorithmus 42
Listing 16: Beispieltest Teil X: Implementation der Variante (1) 43
Listing 17: Beispieltest Teil XI: Implementation der Varianten (2) und (3) 43
Listing 18: Beispieltest Teil XII: Die Methode einen Test zu (re-)initialisieren 43
Listing 19: Interfaces ITextResult und IGraphicResult 44
Listing 20: Implementation des ITextResult Interfaces 46
Listing 21: Methoden des Interfaces IGraphicResult 46
Listing 22: Implementation des IGraphicResult Interfaces 47

10.2.6 Literaturverzeichnis

1. Dan Gusfield, *Algorithms on Strings, Trees and Sequences*, CUP, 1997
2. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns*, Addison-Wesley, 1994
3. Kent Beck, *Exreme Programming (XP)*, Addison-Wesley, 2000
4. Robert Sedgewick, *Algorithmen*, Princeton University, Addison-Wesley, 1992
5. Mark Cieliebak, Thomas Erlebach, Zsuzsanna Lipták, Jens Stoye, Emo Welzl, *Alorithmic Complexity of Protein Identification: Combinatorics of Weighted Strings*, ETH Zurich, Dept. of Computer Science; Submitted to DAM (Discrete and Applied Mathematics), December 21, 2001
6. Robert Giegerich, Stefan Kurtz and Jens Stoye, *Efficient Implementation of Lazy Suffix Trees*; to appear in Proc. of 3rd Workshop Of Algorithm Engineering London, UK, July 1999
7. B. Dorohonceanu and C. G. Nevill-Manning, *A Practical Suffix Tree Implementation for String Search*, Algorithm Alley, Dr. Dobb's Journal 314, 133-140, 7/2000
8. URL: http://www-igm.univ-mlv.fr/~lecroq/string
9. URL: http://www.mills.edu/ACAD_INFO/MCS/CS/S00MCS125/String.Matching.Algorithms/stringmatching.html
10. URL: http://www.cs.ucr.edu/~stelo/pattern.html
11. URL: http://csweb.haifa.ac.il/library/
12. URL: http://cbrg.inf.ethz.ch/Darwin/

10.3 Danksagung

Zum Schluss möchte ich allen danken, die mich während dieser Diplomarbeit unterstützt haben. Allen voran meinen beiden Assistenten Zsuzsanna Lipták und Mark Cieliebak für die gute Betreuung in den vergangen vier Monaten.

11 Anhang B