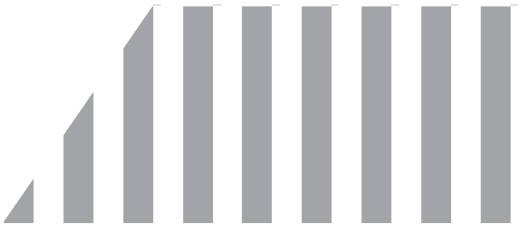
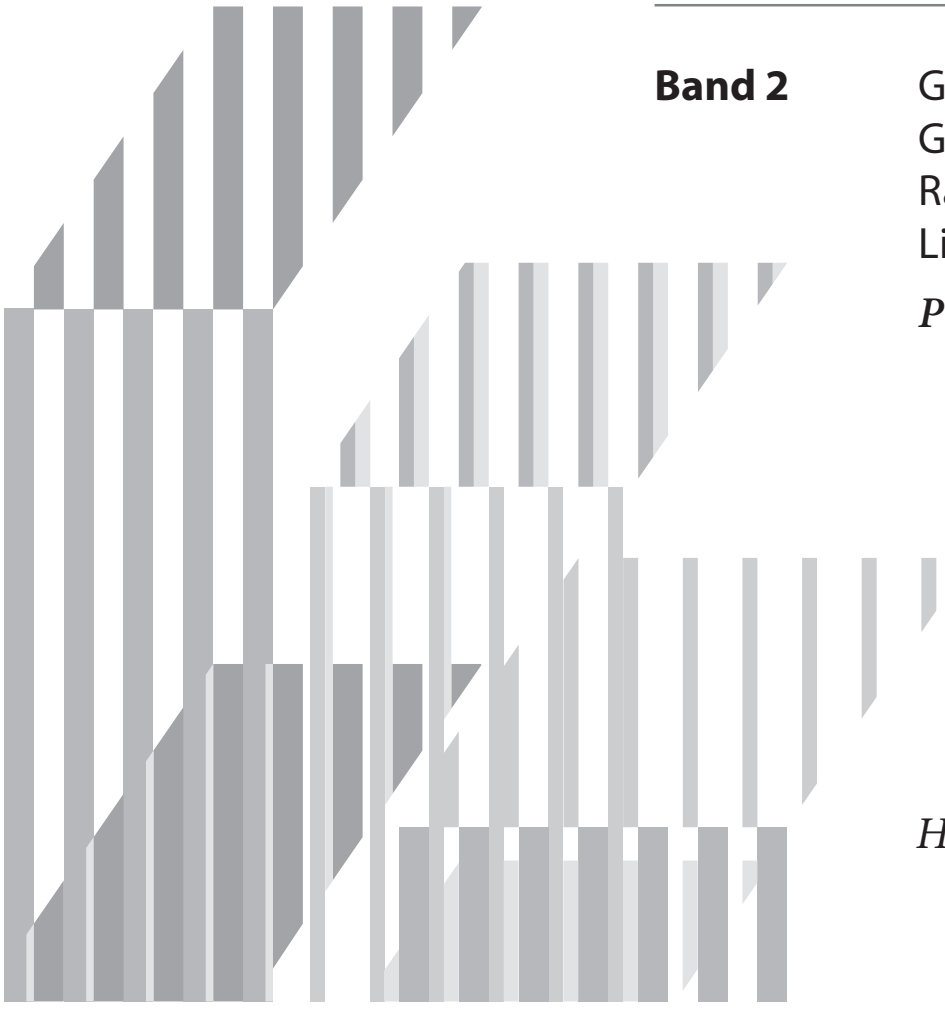


MODERNE STRUKTURIERTE PROGRAMMIERUNG

Objektorientiert und algorithmisch
ENTWERFEN | IMPLEMENTIEREN | TESTEN

2. Auflage



Band 2

Grundlagen
Gruppen
Ranggruppen
Link-Listen

Praxis

Horst van Bremen

Bibliografische Information der Deutschen Nationalbibliothek:
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.dnb.de abrufbar.

Gestaltung	Katharina Schmitt
Titel- und Einbandgrafik	Monika Sylvia Gomm † 1971
Englisch-Korrektorat	David Roseveare
Verlag	BoD · Books on Demand GmbH, Überseering 33, 22297 Hamburg, bod@bod.de
Druck	Libri Plureos GmbH, Friedensallee 273, 22763 Hamburg
Version / Datum	Version 2.1.1 vom 1.7.2026
Literaturverzeichnis	siehe Kapitel „11.1 Literaturverzeichnis“ auf Seite 441 f
Rechtliches	siehe Kapitel „11.2 Rechtliche Hinweise“ auf Seite 443 ff
ISBN des Hardcover-Buchs	978-3-6957-8393-9

Über den Autor

Horst van Bremen
Diplom 1972 an der TU Darmstadt – Elektrotechnik und Informatik
39 Berufsjahre in der IT, davon 18 als Freiberufler

IT-Systemarchitekt
Datenbankexperte

Programmiererfahrung seit 1969
Strukturierte Programmierung nach Jackson seit 1974
Freier Autor seit 2011

Vorwort

Der Band 2 der ersten Staffel dieser Buchreihe versammelt alle Programmbeispiele, die auf der Basis der JSP/MSP-Entwürfe im Band 1 vom Autor geschrieben wurden, sowie Testkapitel und Testprotokolle. Das ist zumindest unüblich. Normalerweise enthalten Bücher über Programmierung zwar Code-Beispiele in Form kurzer Abschnitte, die englisch Snippets – also Schnipsel oder Bruchstück – genannt werden, aber den Code ganzer Programme – und erst recht Tests hiervon – hat der Autor noch in keinem Fachbuch gesehen. Im vorliegenden Fall gibt es aber für das ungewöhnliche Vorgehen gute Gründe:

- Anfänger tun sich häufig schwer damit, etwas, das sie theoretisch verstanden haben, in ein funktionierendes Programm umzusetzen. Die dafür erforderliche Erfahrung braucht ihre Zeit, um sich zu entwickeln. Um so besser und schneller geht es, wenn Beispiele vorliegen, an denen sich die eigenen Entwürfe orientieren können. Die Download-Option auf www.jsp-msp.de erspart das Abtippen.
- In Jahrzehnten des Ausprobierens und des Scheiterns bildeten sich Kriterien dafür, wie eine professionelle Programmierung aussehen kann, nicht muss. Ein Beispiel: In den allerersten ALGOL-Programmen des Autors hießen die Variablen A1, A2, A3 und so weiter, was diesen Code von vornherein nahezu unverständlich machte. Wie wichtig es ist, passende Namen zu wählen, stellte sich erst nach und nach heraus. Hier möchte der Band 2 stilbildend sein.
- Große Unsicherheiten bestehen anfangs auch bei der Modularisierung, sowie bei der OO-Programmierung, ganz besonders aber bei der Zuordnung von Code-Snippets zu Methoden. Wie man hier geschickt vorgeht, das lehrt auch JSP/MSP nicht. Hinsichtlich dieses Aspekts wird im Band 2 angestrebt, zumindest gute Lösungen zu zeigen, ohne den Anspruch auf Vorbildlichkeit zu erheben.
- Vollständig ungewöhnlich ist es, JSP/MSP-Entwürfe in OO-Sprachen zu realisieren. Die strukturierte, datenbasierte Methode von Michael A. Jackson wurde zwar primär für die algorithmische Programmierung konzipiert, eignet sich aber auch für OO-Implementierungen. Das Zusammenspiel „echter" OO-Klassen mit den neu eingeführten Process Control-Klassen, die sich den JSP-typischen Baumstrukturen verdanken, kann anhand der hier vorgelegten Beispiele genau studiert werden.
- Die Aufteilung der Buchreihe in Paare von Methoden- und Praxis-Bänden ist nicht nur dem Umfang der Bände geschuldet, sondern sie dient hauptsächlich dazu, die Entwürfe und ihre Umsetzungen nebeneinanderlegen zu können. Durch die konsequente Übernahme der Operations- und Kontroll-Bezeichnungen in die Programme und durch die ausführliche Kommentierung kann die Ableitung des Codes aus den Struktogrammen gut nachvollzogen werden. Die Test-Abschnitte unterstützen das Verständnis.
- Code-externe Kommentare, Anmerkungen und teilweise recht detaillierte Darstellungen dienen dazu, den Implementierungen eine eigene Semantik zu geben. Auch gut kommentierter Code ist selten wirklich selbsterklärend. Die Zusatztexte sollen die verbleibende Verständnislücke schließen helfen.
- Um „den Wald" zu schonen, wurden Redundanzen nach Kräften vermieden. Wiederverwendeter Code wird nicht wiederholt, sondern es wird auf dessen erstes Auftreten verwiesen.

Es ist dem Autor bewusst, dass es vermutlich ebensoviele Meinungen über die „richtige" Programmierung wie Programmierer(innen) gibt. Daher ist es möglich, dass die hier gezeigten Implementierungen auf heftige Kritik oder gar auf völlige Ablehnung stoßen. Ganz sicher gibt es auch an den nicht immer konsequent umgesetzten Namenskonventionen und anderen Formalia dies und jenes zu bemängeln. Insofern hofft der Autor auf die Gnade derjenigen, die es damit ganz besonders streng halten. Allen anderen mögen die Kapitel des zweiten Bands hoffentlich von großem Nutzen sein.

Lemgo, den 2.5.2023

Übersichtsverzeichnis Band 2

Inhaltsverzeichnis Band 2

Band 2

Band 2

06/2022 oben / links: Rhetorica, Musica und Arithmetica an der Nordseite der Ratslaube
rechts: Ratslaube von 1565 des Lemgoer Rathauses mit 1589 aufgesetzter Kornherrenstube (NRW)

Example1 in COBOL II und C

1.1 COBOL-II-Code von Example1

Nach den ausführlichen Vorbereitungen am Anfang des Bands 1 fällt das algorithmische Programmieren leicht. Als Hommage an Jackson wurde zuerst die Sprache COBOL II für die Implementierung ausgewählt. Zum Vergleich folgt im Kapitel „1.3 C-Code von Example1" auf Seite 15 ff dasselbe Programm in der Sprache C.

COBOL II ist nicht „case sensitive", unterscheidet also – ausser in String-Konstanten – nicht zwischen Groß- und Kleinbuchstaben. Die Kleinschreibung wurde in diesem Beispiel nur wegen der besseren Lesbarkeit gegenüber der allgemein üblichen Codierung mit Großbuchstaben gewählt.

Das Programm lautet:

```
identification division.

program-id. example1.

environment division.

configuration section.

input-output section.

file-control.
    select textin assign textin
           access mode is sequential
           file status is textin-fs.

    select textout assign textout
           access mode is sequential
           file status is textout-fs.

data division.

file section.

fd textin                   block 0
                            recording f.
01  filler                 pic  x(55).

fd textout                  block 0
                            recording f.
01  out-rec                pic  x(77).
```

```
 working-storage section.

*    counters

 01  blanks                 pic s9(04) comp value zero.
 01  chars                  pic s9(04) comp value zero.
 01  ind                    pic s9(04) comp value zero.
 01  text-len               pic s9(04) comp value zero.
*    op. 9,10,11
 01  total-blanks           pic s9(04) comp value zero.
 01  total-chars            pic s9(04) comp value zero.
 01  total-len              pic s9(04) comp value zero.

*    switches

 01  char-flag              pic  x          value space.
     88  char-not-found                     value 'N'.
     88  char-found                         value 'F'.
 01  eof-flag               pic  x          value 'D'.
 01  textin-fs              pic  x(02)      value spaces.
     88  textin-fs-ok                       value '00'.
     88  textin-fs-end                      value '10'.
 01  textout-fs             pic  x(02)      value spaces.
     88  textout-fs-ok                      value '00'.

*    records

 01  in-string              pic  x(55)      value spaces.
 01  out-line.
     05  out-string         pic  x(55)      value spaces.
     05  filler             pic  x(02)      value spaces.
     05  chars-9            pic  z(03)9     value zero.
     05  filler             pic  x(04)      value spaces.
     05  blanks-9           pic  z(03)9     value zero.
     05  filler             pic  x(04)      value spaces.
     05  len-9              pic  z(03)9     value zero.
 01  header-line.
     05  filler             pic  x(55)      value 'Text'.
     05  filler             pic  x(22)      value
                            ' chars  blanks  length'.
 01  empty-line             pic  x(77)      value spaces.
 01  footer-line.
     05  filler             pic  x(55)      value spaces.
     05  filler             pic  x(22)      value
                            '  ====    ====    ===='.
```

```
*------------------*
 procedure division.
*------------------*
 main section.
*    op. 1 & 2: open input & output files
     open input  textin
     open output textout

*    op. 6: read 1st input record
     perform text-read

*    op. 7,8: generate header
     write out-rec from header-line
     write out-rec from empty-line

*    I1: build extended strings
     perform until eof-flag = 'E'
*       op. 12,13
        move zero to chars
                     blanks

*       I2: process input string & op. 15 / 16
        perform varying ind from 1 by 1
                until ind > text-len
*          S3: blank found?
           if in-string(ind:1) = space
*             op. 18
              add 1 to blanks
           else
*             op. 17
              add 1 to chars
           end-if
        end-perform

*       op. 19,20,21,22,23
        add  chars       to total-chars
        add  blanks      to total-blanks
        add  text-len    to total-len
        move in-string   to out-string
        move chars       to chars-9
        move blanks      to blanks-9
        move text-len    to len-9

*       op. 24
        write out-rec from out-line

*       op. 6: read next input record
```

```
           perform text-read
        end-perform

*       generate footer
*       op. 25
        move total-chars  to chars-9
        move total-blanks to blanks-9
        move total-len    to len-9

*       op. 26
        write out-rec from footer-line

*       op. 27
        move spaces   to   out-string (1:49)
        move 'totals' to   out-string (50:6)
        write out-rec from out-line

*       op. 3,4,5: close input & output files, stop
        close textin
        close textout
        stop run.

* The section text-read prepares the input data for the main section and
* it sets the end indicator when EOF is reached.
* For each existing record, text-read determines its length without
* trailing blanks.
* Read errors cause an error message to be written before the program
* terminates with the return code 16.

 text-read section.

        read textin into in-string

        if textin-fs-ok
*          op. 14 (initial)
           move zero           to text-len
           set  char-not-found to true

           perform varying ind from 55 by -1
                   until ind < 1 or
                         char-found
              if not in-string(ind:1) = space
*                op. 14 (final)
                 move ind        to text-len
                 set  char-found to true
              end-if
           end-perform
```

```
else
   if textin-fs-end
      move 'E' to eof-flag
   else
      display '++++ textin-fs = ' textin-fs
      move 16 to return-code
      stop run
   end-if
end-if
continue.
```

Die Ausgabe des Sonetts im Unterkapitel „Das erste JSP-Beispiel“ des Band-1-A1-Kapitels „JSP-Basismethode“ wurde von diesem Programm erzeugt. Aus Platzgründen wurden die Eingabesätze auf 55 Zeichen gekürzt. Diese Festlegung könnte natürlich geändert werden. Dabei wären jedoch die starr codierten Längen, Schleifendeabfragen etcetera anzupassen. COBOL kennt leider keine Konstantendeklarationen, die für Variable und Array-Dimensionierungen verwendet werden könnten. Das ist eine Quelle vieler Irrtumsmöglichkeiten.

1.2 COBOL-II-Code-Durchgang

Für COBOL-Unkundige oder -Ungeübte werden nachfolgend die Bestandteile des Programms und die Ableitung aus der Programmstruktur mit Operationen – respektive dem Flussdiagramm oder dem Pseudocode – dargestellt. Geübte COBOL-Programmierer(innen) werden dagegen vermutlich diverse Stilkritiken anbringen, weil sich in Jahrzehnten der COBOL-Programmierung viele norm-ähnliche Codiervorschriften herausgebildet haben, die allerdings oftmals von Installation zu Installation verschieden sind.

1.2.1 Programmkopf

Das Programm beginnt mit

```
identification division.

program-id. example1.

environment division.

configuration section.
```

Die Einteilung in Divisions und Sections ist der COBOL-I-Tradition geschuldet. Wichtig ist hier nur die Program ID `example1`. Es sind maximal 8 Zeichen für den Programmnamen zulässig, was erheblich dazu beiträgt, Unterprogrammaufrufe in der Form `CALL 'xxxxxxxx'` kryptisch aussehen zu lassen und Namensverwechslungen zu erleichtern. „Sprechende“ Programmnamen waren zu Zeiten knappen Plattenplatzes nicht erwünscht. Unterprogramme können in COBOL II allerdings unter Verwendung von Data Names als Langnamen aufgerufen werden. Der Compiler erzeugt eine Lademodul-Vorform namens `EXAMPLE1` für den Linkage Editor.

1.2.2 Input-Output Section

In der `input-output section` werden die Ein- und Ausgabedatenbestände beschrieben, soweit diese für die Beziehungen zum Trägersystem relevant sind. Dieses kennt die Dateien unter dem rechts neben `assign` stehenden Namen, während links davon andere Bezeichnungen verwendet werden können – eine weitere Quelle möglicher Irrtümer:

```
input-output section.

file-control.
    select textin assign textin
           access mode is sequential
           file status is textin-fs.

    select textout assign textout
           access mode is sequential
           file status is textout-fs.

data division.

file section.

fd textin                  block 0
                           recording f.
01  filler                 pic  x(55).

fd textout                 block 0
                           recording f.
01  out-rec                pic  x(77).
```

Die Details dieser Anweisungen können hier übergangen werden. Zur Vertiefung empfiehlt sich für Anfänger ein COBOL-Lehrbuch, weil diese Sprache ziemlich gewöhnungsbedürftig ist. Der einzige hier interessierende Punkt ist, dass COBOL verlangt, die Strukturen der Ein- und Ausgaben unabhängig von deren programminterner Verwendung in der `file section` zumindest grob zu definieren, wobei jeder links von `assign` stehende Name in einer File Definition = `FD` vorkommen muss.

Im vorliegenden Fall ist für `textin` eine anonyme Variable mit der Standardbezeichnung `filler` angegeben, der ein String von 55 Zeichen Länge zugeordnet ist (hierzu unten mehr). Diese redundante Deklaration ist eine der häufigen Fehlerquellen bei Programmänderungen, welche die Längen der Ein- und Ausgabebereiche betreffen. `filler` kann übrigens innerhalb von Datendeklarationen beliebig häufig verwendet werden, um anonymen Speicherbereichen Formate – und initiale Arbeitsspeicherinhalte – zuzuweisen.

Im ausführbaren Code können diese anonymen Variablen und Felder jedoch nicht angesprochen werden. Sie sind reine Platzhalter. `filler` darf in COBOl II auch weggelassen werden:

```
01                         pic  x(55).
```

In COBOL existiert eine sonderbare Asymmetrie zwischen den Ein- und Ausgabeanweisungen. Beim Lesen aus einer sequentiellen Datei wird die Form

```
READ <input file> [INTO <in rec>]
```

verwendet, wobei `<input file>` den FD-Namen meint. Dadurch wird der nächste Satz in den FD-Bereich gebracht oder der EOF-Indikator im File-Status auf `'10'` gesetzt, wenn der letzte Satz gelesen wurde. Der optionale Zusatz `INTO <in rec>` bewirkt, dass der Satzinhalt zusätzlich in dasjenige Feld in der `working-storage` gebracht wird, das der Platzhalter `<in rec>` benennt. Dabei brauchen die Feldlängen nicht übereinzustimmen, was manchmal nützlich, meistens aber eine weitere Fehlerquelle ist.

Im vorliegenden Programmbeispiel wird die folgende Anweisung verwendet:

```
read textin into in-string
```

Nach einem Laufabbruch kann in einem Dump der `working-storage` der aktuelle Eingabesatz lokalisiert und für die Fehlersuche verwendet werden, während der Inhalt des FD-Bereichs eventuell nicht mehr greifbar ist. Unter anderem deshalb empfiehlt es sich, mit `INTO` zu arbeiten.

Dagegen lautet die Syntax zum Schreiben wie folgt:

```
WRITE <output record> [FROM <out rec>]
```

Anstelle des Ausgabe-FD-Namens ist der deklarierte Name des Ausgabepuffers anzugeben. Je nach der bei der Namensvergabe waltenden Fantasie kann es schwierig sein, in einem großen Programm festzustellen, welcher FD-Name eigentlich gemeint ist. Lässt man den optionalen `FROM`-Teil weg, so erwartet COBOL, dass die auszugebenden Daten im Ausgabepuffer stehen, wohin sie normalerweise mit `MOVE` gebracht werden.

Mit dem Zusatz `FROM <out rec>` wird COBOL angewiesen, zuerst den Inhalt des damit bezeichneten Felds aus der `working-storage` in den Ausgabepuffer zu kopieren und dann diesen in die Datei – oder einen Betriebssystempuffer – zu übertragen. Auch hier können die Längen sich unterscheiden: Zu kurze Sendefelder werden mit Blanks aufgefüllt, zu lange abgeschnitten.

Ein Beispiel hierfür ist:

```
write out-rec from out-line
```

Beim Abbruch des Programmlaufs mit nachfolgender Dump-Ausgabe gelten hier dieselben Hinweise wie für `READ ... INTO`. Zudem können unterschiedliche Ausgabefelder verwendet werden, wovon das Programmbeispiel Gebrauch macht. (Beim Lesen dürfte es eher für Verwirrung sorgen, wenn verteilte `READ`-Anweisungen verschiedene Eingabefelder nach `INTO` angeben. Hier empfiehlt es sich sehr, ein gemeinsames Lesemodul zu verwenden.)

Folglich muss in der FD zur Ausgabedatei ein Data Name für den Ausgabepuffer angegeben werden. `filler` darf hier nicht stehen.

1.2.3 Working Storage

Die `working-storage section` enthält die Deklarationen der Zähler, Schalter und Felder. Jede Deklaration beginnt mit einer Level Number, wobei 01 die oberste Ebene darstellt. Dort liegen die voneinander unabhängigen Variablen und dort beginnen die Felder, die eine mehr oder minder komplexe innere Struktur besitzen können, oder – wie `in-string` – einfach aus einer Menge gleichartiger Datenelemente bestehen.

Unterstrukturen werden mit höheren Level-Nummern als 01 eingeleitet, wobei sich die Nummerierung 05, 10, 15 etcetera allgemein durchgesetzt hat. Die Level-Nummer 88 bezeichnet ausnahmsweise etwas völlig

anderes: Hier werden Zuordnungen zwischen Schalterinhalten und Namen vorgenommen (Condition Names). Das ist sehr praktisch, erfordert aber sorgfältiges Arbeiten.

Je nach COBOL-Implementierung auf dem jeweiligen Zielsystem gibt es Empfehlungen, aus Adressierungsgründen nicht zu viele 01-Level zu definieren oder für einzelne Variable respektive strukturlose Felder den Level 77 zu verwenden. Bei anderen Implementierungen sind genau gegenteilige Empfehlungen anzutreffen. Daher sollten Sie die für Ihre Installation geltenden Regeln erfragen.

```
 working-storage section.

*    counters

 01  blanks                 pic s9(04) comp value zero.
 01  chars                  pic s9(04) comp value zero.
 01  ind                    pic s9(04) comp value zero.
 01  text-len               pic s9(04) comp value zero.
*    op. 9,10,11
 01  total-blanks           pic s9(04) comp value zero.
 01  total-chars            pic s9(04) comp value zero.
 01  total-len              pic s9(04) comp value zero.

*    switches

 01  char-flag              pic  x          value space.
     88  char-not-found                     value 'N'.
     88  char-found                         value 'F'.
 01  eof-flag               pic  x          value 'D'.
 01  textin-fs              pic  x(02)      value spaces.
     88  textin-fs-ok                       value '00'.
     88  textin-fs-end                      value '10'.
 01  textout-fs             pic  x(02)      value spaces.
     88  textout-fs-ok                      value '00'.

*    records

 01  in-string              pic  x(55)      value spaces.
 01  out-line.
     05  out-string         pic  x(55)      value spaces.
     05  filler             pic  x(02)      value spaces.
     05  chars-9            pic  z(03)9     value zero.
     05  filler             pic  x(04)      value spaces.
     05  blanks-9           pic  z(03)9     value zero.
     05  filler             pic  x(04)      value spaces.
     05  len-9              pic  z(03)9     value zero.
 01  header-line.
     05  filler             pic  x(55)      value 'Text'.
     05  filler             pic  x(22)      value
                            ' chars  blanks  length'.
```

```
01  empty-line              pic  x(77)      value spaces.
01  footer-line.
    05  filler              pic  x(55)      value spaces.
    05  filler              pic  x(22)      value
                            '  ====    ====    ===='.
```

Die Namen rechts neben den Level-Nummern werden im ausführbaren Code verwendet. Sie unterliegen etlichen Regeln, die hier nicht referiert werden.

`pic` ist die Abkürzung für „picture". Condition Names haben keine Pictures, sondern sind Namen für Zustände ihrer Schaltervariablen.

Rechts daneben befinden sich die Formatangaben, die zu lesen einige Übung erfordert:

- `s9(04) comp` ist eine Zahl im internen Format mit maximal 4 Dezimalstellen und einem Vorzeichen (s = Sign). Die 9(04) könnte auch 9(4) oder 9999 geschrieben werden. Die Zahl in der Klammer ist also ein Wiederholfaktor. „Comp" bedeutet „computational", also zum Rechnen bestimmt.
- `Z(03)9` ist dagegen eine externe Zahldarstellung mit ein bis vier druckbaren Ziffern. Das z bedeutet Zero Suppression, also die Ausgabe von Blanks an diesen Stellen, wenn die Zahl weniger als vierstellig ist. Das erledigt COBOL automatisch. `z(03)9` könnte auch `z(3)9` oder `zzz9` geschrieben werden.
- `x` ist ein einzelnes alphanumerisches Zeichen.
- `x(02)` oder `xx` bedeutet dagegen zwei alphanumerische Zeichen. Rückgemeldete File-Status-Werte in `textin-fs` oder `textout-fs` sind immer numerisch mit zwei Ziffern. Somit wäre auch `99` oder `9(2)` oder `9(02)` richtig. Jedoch bedeutet eine mit zwei Leerzeichen gefüllte File-Status-Variable, dass mit dieser Datei in diesem Lauf noch nichts geschah. Daher wurde das Format `x(02)` = `xx` gewählt.
- `x(55)` ist ein String aus 55 Zeichen. Es wäre ziemlich unpraktisch, 55 mal x nacheinander zu schreiben… Analog bedeutet `x(22)` einen String aus 22 Zeichen etcetera.

Die `VALUE`-Klauseln weisen in allen Leveln ausser 88 den Variablen oder Feldern Werte zu:

- `value zero` veranlasst die Initialisierung mit Null(en) im jeweiligen Format. Comp-Variablen erhalten den Inhalt „binär Null". Alphanumerische Variablen werden mit so vielen druckbaren Nullen gefüllt, wie das Format angibt, sowie eventuell mit führenden Leerzeichen. `zeroes` wäre auch zulässig. `zero` und `zeroes` sind figurative Konstanten, die auch im ausführbaren Code zulässig sind.
- `value space` oder auch `value spaces` bewirkt die Initialisierung mit einem oder mehreren Leerzeichen (Blanks), je nach Länge der Variablen oder des Felds. Dem Compiler ist die Ein- oder Mehrzahl dieser figurativen Konstanten egal.
- `value '00'` bedeutet, dass der Condition Name `textin-fs-ok` den Wert `true` annimmt, wenn in `textin-fs` zwei Nullen stehen. In dieser Statusvariablen meldet die COBOL-eigene Lese-Routine, die zum Sprachpaket dazugehört, ob ein Satz eingelesen wurde (Wert `'00'`), das Dateiende erreicht ist (Wert `'10'`), oder ein Fehler auftrat. Im vorliegenden Beispiel wurde eine knappe Fehlerbehandlung implementiert.
- Es ist gute Praxis, möglichst allen Elementen des Arbeitsspeichers per `VALUE`-Klausel Initialwerte zuzuweisen. Erinnern sich sich noch an die nicht initialisierte Variable in der veränderten Schleifenende-Abfrage (siehe Unterkapitel „Der intuitive Ansatz" des Band-1-A1-Kapitels „JSP-Basismethode")? Diese verursachte ein völlig unnötiges Problem. Es kostet etwas Mühe, die Initialwerte anzugeben, aber dann ist hier Ruhe.

Jede Deklarationszeile endet mit einem Punkt. Fehlt dieser, so erwartet der Compiler weitere Angaben auf den folgenden Zeilen. Meist reichen ein oder zwei Zeilen für die Deklarationen der Datenelemente respektive Datengruppen. Aus Platzgründen sind zweimal die Value-Strings auf Folgezeilen angegeben.

Unterstrukturen wurden hier für die Ausgabedeklarationen verwendet. In diesen Fällen erhalten die 01-Level keine Picture-Angabe, weil sie ja aus der Zusammenfassung der Unterstrukturen bestehen. Diese liegen sämtlich auf dem Level 05, sind also gleichrangig und folgen daher sequentiell aufeinander.

Mit `FILLER`-Definitionen findet hier die Ausrichtung der Spalten im vorgegebenen Ergebnisformat statt. Mehrfach sind Textkonstante definiert, die nicht nur aus Leerzeichen bestehen. Das Ziel ist hier, möglichst wenige Konstanten im ausführbaren Code angeben zu müssen. Änderungen können so viel zuverlässiger stattfinden, weil nicht überall nach davon betroffenen Strings gesucht werden muss. Ausserdem sind die vom Compiler umgesetzten Konstanten erheblich effizienter nutzbar als Strings, die erst im Programmablauf erzeugt werden.

Nun fragen Sie sich möglicherweise noch nach dem Unterschied zwischen Variablen und Feldern. Hier gibt es keine scharf gezogene Grenze. Unter Feldern versteht man allerdings eher längere Speicherbereiche, die eine zusammengesetzte Struktur aufweisen oder zu solchen Strukturen gehören und nicht elementar sind. Elementar sind einfache Variable, die beispielsweise in Berechnungen verwendet werden.

Sie werden möglicherweise eine Array-Definition für den Eingabe-String vermissen. Darauf wurde verzichtet, weil die Abfrage eines einzelnen Zeichens hier mit Hilfe des sogenannten Reference Modifiers gelöst ist: `in-string(ind:1)` ist dasjenige Zeichen, das im Feld `in-string` auf der Position `ind` steht. 1 ist hier die Länge des Teilstrings. In COBOL beginnen Indizes mit 1.

Abschließend ist noch der Unterschied zwischen den Variablen wie `blanks` und `blanks-9` anzusprechen. Diese sind nicht nur durch den nahezu gleichen Namen einander zugeordnet: `blanks-9` dient dazu, den in `blanks` enthaltenen binären Wert in eine druckbare Ausgabe umzuwandeln, was die Anweisung

```
            move blanks      to blanks-9
```

implizit erledigt. In COBOL muss also keine Formatierungsroutine aufgerufen werden, um einen Variablen- bzw- Feldinhalt von einer Darstellung in eine andere zu transformieren, solange die dafür geltenden Regeln eingehalten werden. Das ist eine der Stärken dieser Programmiersprache, die ihr vor allem in der kommerziellen Programmierung rasch eine weite Verbreitung sicherten.

1.2.4 Procedure Division

Mit den folgenden Zeilen beginnt der ausführbare Code. Er besteht aus zwei Sections, nämlich dem Hauptprogramm und einer Lese-Routine. Das Hauptprogramm wird nun abschnittsweise erläutert. Die einleitenden und abschließenden Zeilen lauten:

```
      *-------------------
       procedure division.
      *-------------------
       main section.
      *    op. 1,2,6
           open input  textin
           open output textout
           perform text-read

      *    generate header: op. 7,8
```

```
           write out-rec from header-line
           write out-rec from empty-line

       *   I1: build extended strings
           perform until eof-flag = 'E'
              === Schleifenkörper ===
           end-perform

       *   op. 25
           move total-chars  to chars-9
           move total-blanks to blanks-9
           move total-len    to len-9

       *   op. 26
           write out-rec from footer-line

       *   op. 27
           move spaces   to   out-string (1:49)
           move 'totals' to   out-string (50:6)
           write out-rec from out-line

       *   close input & output files, stop: op. 3,4,5
           close textin
           close textout
           stop run.
```

Eingangs werden die Dateien geöffnet (Operationen 1,2). Es folgen das Vorlesen mit Hilfe der Routine `text-read` (Op. 6) und die Ausgabe des Listenkopfes (Op. 7,8).

Die Operationen 9,10,11 müssen nicht codiert werden, weil die `VALUE`-Klauseln diese Initialisierungen bereits bewirkt haben.

Danach beginnt mit

```
           perform until eof-flag = 'E'
```

die I1-Schleife, die mit

```
           end-perform
```

endet. `perform until` repräsentiert in COBOL die abweisende Schleife.

Die drei folgenden `MOVE`-Anweisungen entsprechen der Operation 25. Es folgt die Operation 26. Bevor die Ausgabe der Summenzeile stattfinden kann, muss `out-string` vorbereitet werden. Das ergibt zusammen die Operation 27.

Danach werden die Dateien geschlossen (Op. 3,4). Mit `stop run` endet der Lauf (Op. 5). Dieser Anweisung folgt der einzige Punkt in der `main section`. Dieser Punkt ist ein COBOL-I-Relikt, ebenso wie die von einem Punkt gefolgte Anweisung `continue` am Ende der Lese-Routine.

`continue` ist eine NOP-Anweisung (NOP = no operation), die beispielsweise verwendet werden kann, um komplexe Negationen in Abfragen zu vermeiden. Am Ende einer `section` dient diese Anweisung nur dazu, einen formalen Schlusspunkt zu setzen. Man könnte anstelle von

```
end-if
continue.
```

auch wie folgt codieren:

```
end-if.
```

Damit handelt man sich aber eine Verpflichtung zu zusätzlicher Aufmerksamkeit ein.

Der Schleifenkörper lautet:

```
*       op. 12,13
        move zero to chars
                     blanks

*       I2: process input string & op. 15 / 16
        perform varying ind from 1 by 1
                until ind > text-len
*          S3: blank found?
           if in-string(ind:1) = space
*             op. 18
              add 1 to blanks
           else
*             op. 17
              add 1 to chars
           end-if
        end-perform

*       op. 19,20,21,22,23
        add  chars       to total-chars
        add  blanks      to total-blanks
        add  text-len    to total-len
        move in-string   to out-string
        move chars       to chars-9
        move blanks      to blanks-9
        move text-len    to len-9

*       op. 24
        write out-rec from out-line

*       op. 6
        perform text-read
```

Er beginnt mit den Operationen 12 und 13, wobei die Abkürzung auffällt, dass `move zero to` zwei Zielvariable anspricht. Die Op. 14 wird in der `section text-read` realisiert.

Danach ist I2 als Zählschleife – `perform varying` bis `end-perform` – implementiert, die den Index `ind` mit 1 initialisiert (`from 1`) (Op. 15) und mit 1 fortschaltet (`by 1`) (Op. 16). Die Endebedingung ist erfüllt, wenn `ind > text-len` gilt. Diese Schleife endet bei Leerzeilen sofort, das heißt ohne Durchlaufen des Schleifenkörpers, weil dann 1 > 0 abgefragt wird.

Im Schleifenkörper findet die Zählung der Leerzeichen und der anderen Zeichen statt, wobei die Implementierung von S3 den Reference Modifier nutzt (siehe oben).

Die Operationen 18 und 17 repräsentieren die Additionen. Die Operation 16 musste nicht eigens als Sequenz nach S3 herausfaktorisiert werden, da die Zählschleife das erledigt.

Nach `end-perform` folgen die Operationen 19,20,21,22 und 23. Es folgt die Ausgabe einer Ergebniszeile (Op. 24) und das Nachlesen (Op. 6).

Nun fehlen nur noch die Implementierung der Operation 6 und der LENGTH-Funktion:

```
 text-read section.

     read textin into in-string

     if textin-fs-ok
*       op. 14 (initial)
        move zero           to text-len
        set  char-not-found to true

        perform varying ind from 55 by -1
                until ind < 1 or
                      char-found
           if in-string(ind:1) not = space
*             op. 14 (final)
              move ind        to text-len
              set  char-found to true
           end-if
        end-perform
     else
        if textin-fs-end
           move 'E' to eof-flag
        else
           display '++++ textin-fs = ' textin-fs
           move 16 to return-code
           stop run
        end-if
     end-if
     continue.
```

Dafür wurde eine eigene `section` definiert. Darin liest das Programm den ersten oder den nächsten Satz aus dem FD-Bereich in das `working-storage`–Feld `in-string`, das dieselbe Länge besitzt.

Der Condition Name `textin-fs-ok` ist `true`, wenn ein Satz eingelesen wurde. Dann setzt das Programm einen anderen Condition Name, nämlich `char-not-found`, zu `true`. Warum wird nicht einfach `false` dem

Condition Name `char-found` zugewiesen? COBOL lässt das nicht zu, weil die Value-Bereiche von Condition Names einander überlappen können. Diese repräsentieren nur deshalb disjunkte Werte der zugrundeliegenden Schaltervariablen `char-flag`, weil das im vorliegenden Beispiel so codiert ist – aber das muss nicht zwingend so sein. Es ist durchaus möglich, dass in den `VALUE`-Klauseln solcher Condition Names Wertlisten angegeben sind, die gemeinsame Elemente enthalten. Also könnte auch nicht immer eindeutig entschieden werden, was eine `false`-Zuweisung zur Folge hätte.

Die `SET`-Anweisung bewirkt hier, dass der Wert `'N'` der Variablen `char-flag` zugewiesen wird. Sind mehrere Literale in der `VALUE`-Klausel aufgelistet, bringt `SET <condition name> TO TRUE` den ersten dieser Werte in die zugehörige Variable.

Die anschließende absteigende Zählschleife beschafft die String-Länge ohne schleppende Leerzeichen. Die Indexvariable `ind` wird im Schleifenkopf mit 55 initialisiert (`from 55`), also der letzten Zeichenposition im Eingabe-String und nach jedem Durchlauf des Schleifenkörpers um 1 dekrementiert (`by -1`). Das Ende dieser Schleife ist erreicht, wenn der Index unter 1 sinkt oder ein druckbares Zeichen gefunden wurde.

Innerhalb der Schleife prüft das Programm jedes Zeichen mittels Reference Modifier, ob sein binärer Wert nicht `space` ist. Es könnten durchaus Sonderzeichen in der Eingabe enthalten sein, die nicht druckbar sind, aber das spielt hier keine Rolle.

Trifft das Programm ein Zeichen an, das kein Leerzeichen ist, so führt es folgende Zuweisungen aus:

```
move ind        to text-len
set  char-found to true
```

Die String-Länge ist gleich der Position des gefundenen Zeichens. Da nicht weiter gesucht werden muss, setzt das Programm den Condition Name `char-found` auf `true`. Der Wert `'F'` wird der Variablen `char-found` zugewiesen. Nun ist die Schleifenendeabfrage erfüllt und die `text-read section` wird verlassen.

Besteht der gesamte Eingabe-String aus Leerzeichen, so endet die Schleife mit Erreichen der Bedingung `ind < 1`. `text-len` behält dann seinen Initialwert 0.

Diese Implementierung der LENGTH-Funktion wurde als Beispiel für viele andere Suchschleifen aufgenommen. Anstatt – wie hier – einen Schalter abzufragen, werden in vielen Programmen Tricks wie derjenige verwendet, den Schleifenabbruch durch Manipulation des Zählers zu erzwingen. Das mag vor Jahrzehnten mit der damals wesentlich langsameren Hardware halbwegs akzeptabel gewesen sein, kann aber heute wegen der Fehlergefahr bei Änderungen nicht mehr toleriert werden. Im übrigen funktionieren solche Tricks nur bei bestimmten Compilern, können also nach der Portierung auf ein anderes System versagen.

Alternative Implementierungen sind zum Beispiel folgende:

- Ist die Eingabe ein `VARCHAR`-String, der in einer relationalen Zeile enthalten ist, so liefert das Datenbanksystem die Länge in einem zusätzlichen Datenelement mit.
- In Java® meldet die Methode `in_string.length()` die String-Länge zurück.
- In C ergibt `strlen(in_string)` die tatsächliche Zeichenanzahl *ohne* Terminierungs-Null (Vorsicht Falle!), während `sizeof in_string` immer den Wert 56 liefern würde. `sizeof` ermittelt die allokierte Länge. C-Programmierer(innen) meinen übrigens häufig, `sizeof` sei eine Funktion. Das trifft aber nicht zu. Nur bei der Abfrage von Datentyp-Längen muss das Argument geklammert werden: `sizeof(int)`

Übrigens: Die COBOL-II-Klausel `length of in-string` würde hier stets 55 liefern, also die Maximallänge. In COBOL gibt es keine Terminierungs-Null. Variable Längen werden dem Compiler durch Deklaration mit `OCCURS DEPENDING ON <variable>` signalisiert. In der angegebenen Variablen muss die tatsächliche Länge abgelegt werden, die natürlich die jeweilige Obergrenze nicht überschreiten darf. Dies wird allerdings nur dann zur Laufzeit geprüft, wenn die Compile-Option SSRANGE angegeben ist, was wegen der deut-

lich längeren Laufzeiten in Produktion normalerweise nicht zulässig ist. SSRANGE veranlasst, dass auch Indexüberschreitungen und unzulässige Reference Modifiers erkannt werden, was natürlich an vielen Stellen zusätzlichen – generierten – Code erfordert und alle Indexrechnungen verlangsamt.

1.2.5 Schlussbemerkungen zur COBOL-II-Version

- Dies ist das einzige ausführlich kommentierte COBOL-II-Beispiel in dieser Buchreihe. Es erzeugt die geforderte Ausgabe des Gedichts im Unterkapitel „Das erste JSP-Beispiel" des Band-1-A1-Kapitels „JSP-Basismethode".
- Wenn Sie sich mit der strukturierten Programmierung nach Jackson vertraut machen wollen, aber mit anderen Sprachen als COBOL II, C oder Java arbeiten, sollten Sie dieses einfache Beispiel selbst implementieren, um das Beifahrer-Gefühl loszuwerden, das sich beim Betrachten fertiger Lösungen einstellt.
- Experimentieren Sie mit Erweiterungen. So könnten Sie beispielsweise die Satzzeichen getrennt zählen und dafür eine eigene Ausgabespalte vorsehen. Machen Sie die Eingabe breiter als 55 Zeichen und passen die Ausgabe daran an. Zählen Sie die Ein- und Ausgabesätze und geben eine kleine Statistik aus. Was müssen Sie dabei alles beachten?
- Wenn zuvor von „dem Compiler" die Rede ist, so ist ein weit verbreiteter COBOL-II-Compiler gemeint, der unter z/OS® auf IBM®-Großrechnern läuft.
- Das z/OS stellt (COBOL-II-)Batchprogrammen eine vollständig andere Ausführungsumgebung zur Verfügung, als dies unter anderen verbreiteten Betriebssystemen der Fall ist. Solche Programme laufen unter der Regie des Job Entry Subsystem (JES), dessen Kommandosprache Job Control Language (JCL) heisst. Ein Job enthält diejenige Folge von exakt normierten JCL-Anweisungen, die erforderlich ist, um das Programm mit seinen Datenquellen zu verbinden und es zur Ausführung zu bringen. Dazu gehört, dass alle erforderlichen Dateien allokiert werden; die Operationen zum Öffnen und Schliessen sind dagegen Sache des Programms. Das Erlernen und sinnvolle Anwenden der JCL erfordert zwar einen erheblichen Aufwand, aber es ist *die* Grundvoraussetzung für die automatisierte Stapelverarbeitung unter z/OS – zumindest aus der Sicht der Anwender und der Systemadministratoren.

1.3 C-Code von Example1

Die Implementierung von Example1 in C erfolgte gemäß dem ISO-Standard C99 auf einem iMac®, also einem Apple® Macintosh® unter macOS®. Als integrierte Entwicklungsumgebung diente die Lightweight IDE, die ein komfortables Editieren und Testen ohne den Aufwand der macOS-IDE Xcode® ermöglicht (siehe „C-Programmierung auf dem Mac" von Detlef Schulz [12], Seiten 30 & 31). Die aktuelle Version der Lightweight IDE finden Sie im Web unter folgender URL:

http://ragnemalm.se/lightweight/

Übrigens: Die Lightweight IDE färbt den Code ein, damit die Syntax besser erkennbar ist. Der hier gezeigte Code und die weiteren C-Beispiele werden allerdings aus drucktechnischen Gründen schwarz wiedergegeben.

Hinweis: Leider unterstützt Apple die „alten" 32-Bit-Programme wie die Lightweight IDE ab macOS Version 10.15 / Catalina nicht mehr. Der Entwickler der IDE, Ingemar Ragnemalm, arbeitet zwar an einer 64-Bit-Version, hat aber seit Anfang 2021 keinerlei Fortschritte damit vermeldet. Es gibt glücklicherweise mehrere Alternativen zu Lightweight, z.B. Eclipse.

Die C-Version von Example1 sieht vollständig anders aus, leistet aber dasselbe wie die COBOL-II-Version, wie ein Vergleich der Laufergebnisse zeigt. Sie beginnt mit den folgenden globalen, also programmweit gültigen Deklarationen:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Example1 Program

#define TRACE_MODULE 0   // trace module names
#define TRACE_DETAIL 0   // trace logic details

extern int errno;

struct I_F               // input feedback structure
{
   char read_status;
   int  text_len;
};

struct I_F readTextin(FILE * pF_textin,
                      char   textin[]);

void itoa(int value, char result[], int digits);

void substr(char dest[], char src[], int offset, int len);
```

In diesem Programm ist erstmals ein Tracing mithilfe von `printf()`-Anweisungen implementiert, die durch konditionales Kompilieren ein- oder ausgeschaltet werden können. Die Anweisungen

```
#define TRACE_MODULE 0   // trace module names
#define TRACE_DETAIL 0   // trace logic details
```

steuern, ob Trace-Code erzeugt oder unterdrückt wird. Beispielsweise wird die `printf()`-Anweisung in

```
#if TRACE_MODULE
   printf("\n readTextin()\n");
#endif
```

nur dann in ausführbaren Code übersetzt, wenn der Schalter `TRACE_MODULE` den Wert `1` – oder einen anderen Wert als Null – besitzt.

1.3.1 main()

Die `main()`-Funktion – kurz: `main()` – ist der Startpunkt des Programms. Sie enthält ihrerseits etliche lokale Definitionen, denen der ausführbare Code unmittelbar folgt. Als Funktion darf sie bezeichnet werden, weil sie einen Integer-Rückgabewert abliefert, den die Laufzeitumgebung als OK (Wert = 0) oder als Fehlerrückmeldung (Wert ≠ 0) interpretiert. Das Programm enthält einige Abfragen auf Fehler, zum Beispiel auf zu viele oder zu wenige Parameter, die eine Fehlerrückmeldung auslösen, indem sie einen Return-Wert > 0 abliefern.

Wegen dem von COBOL abweichenden Indexstart mit 0 muss übrigens die Operation 15 angepasst werden:

Op.	Bedeutung
15	ind := 0

Der `main()`-Code lautet:

```
int main(int argc, char *argv[])
{
   char   textin[BUFSIZ] = {' '};
   FILE * pF_textin;

   char   tprint[BUFSIZ] = {' '};
   FILE * pF_tprint;

   char   header[]       = "Text"
                           "                              "
                           "                              "
                           "chars  blanks  length";
   char   blanks_55[]    = "                              "
                           "                                 ";
   char   filler[56]     = {' '};
   char   blank_line[]   = {'\r', '\n', '\0'};
   char   blanks_9[]     = {"    "};
   char   chars_9[]      = {"    "};
   char   len_9[]        = {"    "};

   int    total_chars    = 0;   // op. 9
   int    total_blanks   = 0;   // op. 10
   int    total_len      = 0;   // op. 11

   struct I_F IF_New     = {' ', 0};

   printf("*** Example1 ***\n");

   switch(argc)
   {
   case 1:  printf("\n Path 1 (textin) & path 2 (tprint) missing! \n");
            return 2;
   case 2:  printf("\t Path 1:  %s \n", argv[1]);
```

```
          printf("\n Path 2 (tprint) missing! \n");
          return 1;
case 3:   /* for (int i = 1; i < argc; i++)
               { printf("\t Path %d:  %s \n", i-1, argv[i]); } */
          break;
default: for (int i = 1; i < argc; i++)
            { printf("\t Path %d:  %s \n", i-1, argv[i]); }
          printf("\n Too many paths, or apostrophes missing! \n");
          return 3;
}

// op. 1: open input file
pF_textin = fopen(argv[1], "r");

if (pF_textin == NULL)
{
   printf("Textin OPEN failed\n");
   printf("Error: %d (%s)\n", errno, strerror(errno));
   return 4;
};

// op. 2: open output file
pF_tprint = fopen(argv[2], "w");

if (pF_tprint == NULL)
{
   printf("Tprint OPEN failed\n");
   printf("Error: %d (%s)\n", errno, strerror(errno));
   return 5;
};

// op. 6: read 1st input record
IF_New = readTextin(pF_textin, textin);

/* Does the (existing) first line end with \n only?
 * If there is no \r before \n, setup blank_line. */

if (IF_New.read_status == 'D'                  &&
    ((strlen(textin) <= 1                   ||
     (textin[strlen(textin)-1] == '\n' &&
      textin[strlen(textin)-2] != '\r'))))
{
   blank_line[0] = '\n';
   blank_line[1] = '\0';
};

// op. 7,8: generate header
```

```
   fputs(header, pF_tprint);
   fputs(blank_line, pF_tprint);
   fputs(blank_line, pF_tprint);

   // I1: build extended strings
   while (IF_New.read_status == 'D')
   {
      if (IF_New.text_len > 55)
      {
         printf("Record lengths > 55 not supported!\n");
         return 6;
      }

      // op. 12,13
      int chars  = 0;
      int blanks = 0;

      // I2: process input string & op. 15 / 16
      for (int ind = 0; ind < IF_New.text_len; ind++)
      {
         // S3: blank found?
         if (textin[ind] == ' ')
            { blanks++; }   // op. 18
         else
            { chars++; }    // op. 17
      }

#if TRACE_DETAIL
      printf("\n chars        = %d", chars);
      printf("\n blanks       = %d\n", blanks);
#endif

      // op. 19,20,21
      total_chars  += chars;
      total_blanks += blanks;
      total_len    += IF_New.text_len;

      // op. 22: copy input string (with trailing filler blanks)
      substr(tprint, textin, 0, IF_New.text_len);
      substr(filler, blanks_55, 0, 55-IF_New.text_len);
      strcat(tprint, filler);

      // op. 23: convert string counters
      itoa(chars, chars_9, 4);
      itoa(blanks, blanks_9, 4);
      itoa(IF_New.text_len, len_9, 4);
      strcat(tprint, "  ");
```

```
      strcat(tprint, chars_9);
      strcat(tprint, "    ");
      strcat(tprint, blanks_9);
      strcat(tprint, "    ");
      strcat(tprint, len_9);

      // op. 24
      fputs(tprint, pF_tprint);
      fputs(blank_line, pF_tprint);

      // op. 6: read next input record
      IF_New = readTextin(pF_textin, textin);
   }

   // generate footer (op. 25 & 26 reversed)
   // op. 26
   strcpy(tprint, blanks_55);
   strcat(tprint, "  ====    ====    ====");
   fputs(tprint, pF_tprint);
   fputs(blank_line, pF_tprint);

   // op. 25
   itoa(total_chars, chars_9, 4);
   itoa(total_blanks, blanks_9, 4);
   itoa(total_len, len_9, 4);
   substr(tprint, blanks_55, 0, 49);
   strcat(tprint, "totals  ");
   strcat(tprint, chars_9);
   strcat(tprint, "    ");
   strcat(tprint, blanks_9);
   strcat(tprint, "    ");
   strcat(tprint, len_9);

   // op. 27
   fputs(tprint, pF_tprint);
   fputs(blank_line, pF_tprint);

   // op. 3,4: close input & output files
   fclose(pF_textin);
   fclose(pF_tprint);

   printf("*** Successful completion ***\n");

   // op. 5: stop
   return 0;
}
```

Die Unterprogramme folgen der `main()`, könnten ihr aber ebensogut vorangehen. Der C-Compiler der Lightweigth IDE, in welcher das Programm entwickelt, übersetzt und getestet wurde, akzeptiert auch Vorabdeklarationen von Unterprogrammen (also ohne Code-Inhalt). Wenn es viele Unterprogramme sind, sollten sie beispielsweise alphabetisch oder in zusammengehörigen Gruppen organisiert auftreten.

1.3.2 readTextin()

```
/**
 * readTextin() prepares the input data for the main() and it sets the
 * end indicator in read_status when EOF is reached.
 * For each existing record, readTextin() determines its length without
 * trailing blanks / CR / LF.
 * The return value of the module consists of the read_status and the text_len
 * data elements, packed in an I_F struct.
 */
struct I_F readTextin(FILE       * pF_textin,
                      char         textin[])
{
   char     * ps_textin;
   struct I_F IF_Crt     = {'D', 0};   // op. 14 (initial)

#if TRACE_MODULE
   printf("\n readTextin()\n");
#endif

   ps_textin = fgets(textin, BUFSIZ, pF_textin);

   if (ps_textin == NULL)
   {
      IF_Crt.read_status = 'E';
      textin[0]          = '\n';
      textin[1]          = '\0';
   }
   else
   {
      IF_Crt.text_len = strlen(textin);

#if TRACE_DETAIL
   printf("\n textin       = %s", textin);
   printf(" read_status  = %c", IF_Crt.read_status);
   printf("\n text_len raw = %d\n", IF_Crt.text_len);
#endif

      // scan backward for 1st non-blank / non-delimiter character
      for (int i = IF_Crt.text_len-1; i >= 0; i--)
      {
         // printf("\n textin[%d] = %c = x%X", i, textin[i], textin[i]);
```

```
         if (textin[i] == ' '  ||
             textin[i] == '\r' ||
             textin[i] == '\n')
            { IF_Crt.text_len--; }
         else
            { break; }
      };
   };

#if TRACE_DETAIL
   printf("\n text_len     = %d\n", IF_Crt.text_len);
#endif

   return IF_Crt;   // op. 14 completed
}
```

1.3.3 itoa()

```
/**
 * Simple number conversion from int to decimal string w/o sign
 * The result format depends on the parameter <digits> (<= 9):
 * digits = 0: no leading blanks, variable length
 * digits > 0: up to digits-1 leading blanks, length = digits
 * Assumption: <int> has up to 9 digits
 */
void itoa(int value, char result[], int digits)
{
   char digit[11]  = {'0', '1', '2', '3', '4', '5', '6', '7', '8', '9'};
   char stack[10]  = {' '};

   int  remainder = 0;
   int  i         = 0;
   int  k         = 0;
   int  save      = value;

   if (value < 0)
   {
      printf("\t value param < 0: %d\n", value);
      exit(1);
   };

   if (digits < 0 || digits > 9)
   {
      printf("\t wrong digits param: %d\n", digits);
      exit(2);
   };
```

```
   if (value <= 9)
   {
      if (digits <= 1)
      {
         result[0] = digit[value];
         result[1] = '\0';
      }
      else
      {
         for (; k < digits-1; k++)
            { result[k] = ' '; };

         result[digits-1] = digit[value];
         result[digits]   = '\0';
      };

      return;
   };

   // determine decimal digits from right to left
   for (; value != 0; i++)
   {
      remainder = value % 10;
      stack[i]  = digit[remainder];
      value    /= 10;
   };

   if (digits > i)
   {
      for (; k < digits-i; k++)
         { result[k] = ' '; };
   }
   else
      if (digits > 0 && digits < i)
      {
         printf("\t <digits> too small: %d, value = %d\n", digits, save);
         exit(3);
      };

   // reverse the stacked digits order
   for (i--; i >= 0; i--, k++)
      { result[k] = stack[i]; };

   result[k] = '\0';
}
```

1.3.4 substr()

```
/**
 * There is no elegant substr() function in the C99 standard.
 * This local function checks whether the <offset> points
 * behind the string terminator. It copies up to <len> characters
 * until '\n' or '\r' or  '\0', i.e., it removes CR / LF.
 */
void substr(char dest[], char src[], int offset, int len)
{
   int  i = 0;

   if (offset >= strlen(src))
   {
      printf("\t offset %d >= strlen(src): %lu\n", offset, strlen(src));
      exit(4);
   };

   if (len < 0)
   {
      printf("\t len param < 0: %d\n", len);
      exit(5);
   };

   for (; i < len; i++)
   {
      if (src[offset + i] == '\0' ||
          src[offset + i] == '\n' ||
          src[offset + i] == '\r')
          { break; };

      dest[i] = src[i + offset];
   };

   dest[i] = '\0';
}
```

1.4 C-Code-Durchgang

Da die Sprache C eine weit größere Verbreitung als COBOL besitzt und sich als Quasi-Standard für die algorithmische Programmierung etabliert hat, muss der Code-Durchgang nicht ganz so detailliert ausfallen.

1.4.1 Programmvorspann und main()-Funktion

Das Programm beginnt mit den üblichen Präprozessoranweisungen zum Einbinden der gängigen Header-Libraries. Danach folgen `#define`-Anweisungen für den Präprozessor, die das bedingte Übersetzen von Trace-Aufrufen erforderlich sind. Diese Schalter sind nach dem Test ausgeschaltet, verbleiben aber im Programm, damit die Trace-Aufrufe bei Bedarf wieder aktiviert werden können.

Erst danach beginnt der eigentliche C-Code mit der Definition der Rückmeldungsstruktur für die Lese-Funktion `readTextin()`, die sowohl das generelle Zugriffsergebnis (Daten wurden gelesen / Ende der Daten erreicht) als auch die jeweilige Satz- oder Zeilenlänge dem aufrufenden Code meldet. Wenn das Eingabe-Ende erreicht ist, baut der Code eine Pseudo-Eingabe auf, die beim Tracing nicht stört.

Anschliessend folgen die Header der internen Unterprogramme. Das COBOL-II-Programm ist dagegen ein Monolith, dessen Leseoperation in der `text-read section` codiert ist. Sections sind Codeabschnitte, die keine eigenen Parameter besitzen und unmittelbar auf den Speicher desjenigen Moduls zugreifen, in dem sie sich befinden. Sie können von mehreren Stellen innerhalb dieses Moduls aus angestoßen werden, was normalerweise für die Auslagerung gemeinsam genutzten Codes genutzt wird. Dazu gibt es in C keine Analogie.

Beginn der main()

Die `main()` besitzt die üblichen Parameter zur Übergabe von Strings, die extern definiert und von der Laufzeitumgebung beim Start angeliefert werden. Im aktuellen Fall sind das die Pfad- und Dateinamen der Eingabedatei und der Ausgabedatei. Am Anfang der `main()` stehen Puffer- und `FILE`-Pointer-Deklarationen, sowie vorbelegte String-Arrays, die zum Teil Konstante sind, ohne dass dies gesondert angegeben wäre. Es folgen die Summenzähler-Deklarationen und die lokale Deklaration der Lese-Rückmeldungsstruktur.

Beim Programmstart prüft Example1 seine Parameter auf Vollzähligkeit. Es weist falsche Parameter-Anzahlen mit Fehlermeldung ab und endet dann unter Rückgabe eines Return-Wert > 0. Falls Fehler beim Öffnen der Eingabedatei oder der Ausgabedatei auftreten, wird ebenso verfahren.

Ermitteln des Zeilenende-Formats bei gefüllter Eingabedatei

Der anschließende Abschnitt mit den `textin`-Abfragen dürfte für eine(n) COBOL-Programmierer(in) ohne C-Kenntnisse verwirrend sein. Eine sequentielle Eingabedatei enthält entweder Sätze starrer Länge, oder variabel lange Sätze. Letztere sind in COBOL relativ umständlich zu definieren und zu handhaben; zudem liegt zumindest auf Großrechnern eine Gewöhnung an starre Satzformate vor, die durch die Lochkarten-Ära geprägt wurde. So basieren Quelldateien für die Compiler, sowie Parameterdateien dort generell auf dem 80-Zeichen-pro-Satz-Format, während bei anderen Dateien im Allgemeinen darauf geachtet wird, einerseits keinen Platz zu verschwenden, andererseits die Nutzdaten möglichst mit fester Satzlänge zu speichern.

Die COBOL-Variante von Example1 arbeitet in der Dateidefinition (`fd`) für die Eingabe mit `block 0`, überlässt die Satzblockung also dem System, gibt aber mit `recording f` (fixed) das feste Satzformat vor. Die Satzlänge beträgt 55 Bytes. Ausgabeseitig sind es 77 Bytes. Generell braucht ein(e) COBOL-Programmierer(in) keinen Gedanken an die Speicherungsstruktur in den Dateien zu verschwenden, da die Datenorganisation ausserhalb seines Einflussbereichs liegt. Er kann allenfalls die Blockung beeinflussen, also vorgeben, wieviele

Sätze zu Blöcken– als Zugriffseinheiten – zusammengefasst werden. Meist wird auch darauf verzichtet, um die im Betriebssystem vorhandenen Optimierungsmechanismen wirken zu lassen.

Ein(e) C-Programmierer(in) muss dagegen normalerweise davon ausgehen, dass sowohl die Eingabe als auch die Ausgabe variabel lang erfolgt und jeder Satz mit einer systemabhängigen Zeichensequenz endet. Dem oder den satzbeendenden Zeichen folgt aus der Sicht des Programms das von `fgets()` hinzugefügte C-Null-Byte.

Fatalerweise haben sich auf den Plattformen unterschiedliche Normen etabliert, wie die Zeilen beendet werden. Windows® – und davor DOS – beenden die Zeilen mit CR - LF (Carriage Return - Line Feed = Newline). Unix®-Systeme und das macOS verwenden ausschließlich je ein LF-Zeichen. Vor allem der Datenaustausch zwischen Windows-Systemen und Unix-basierten Systemen (inklusive macOS) leidet unter dieser Inkompatibilität.

Wenn ein Programm auf beiden Plattformen laufen soll, muss es darauf vorbereitet werden, entweder CR - LF oder LF allein zu empfangen, sowie auszugeben. Ist die Eingabedatei gefüllt, reicht es, deren ersten Satz auszuwerten, um die systemseitig verwendete Zeilenende-Form zu erkennen.

Verschärfend kommt hinzu, wie die Lesefunktion `fgets()` im C-Standard arbeitet:

```
The fgets() function reads at most one less than the number of characters specified
by n from the given stream and stores them in the string s. Reading stops when
a newline character is found, at end-of-file or error. The newline, if any, is
retained. If any characters are read and there is no error, a '\0' character is
appended to end the string.
```

Das heisst unter anderem, dass der jeweils letzte Satz einer Datei nicht mit CR-LF respektive „nur LF" enden muss, sondern diese Zeichen fehlen dürfen. Dann liefert `fgets()` lediglich den letzten Satzinhalt, der mit dem C-Null-Byte endet. Wenn eine Eingabedatei nur einen einzigen solchen Satz enthält, kann also nur auf Umwegen – oder gar nicht – entschieden werden, wie das korrekte Ausgabeformat aussehen muss. Dieses Problem liegt im aktuellen Fall jedoch glücklicherweise nicht vor.

Im Programm ist CR-LF voreingestellt. LF wird in C übrigens als `'\n'` dargestellt, und CR als `'\r'`. Falls die Datei von einem Windows-System stammt oder durch Microsoft® Word® unter der Ausgabeeinstellung „Text mit Zeilenvorschub" erstellt wurde (was beim Test vorlag), bleibt es dabei. Andernfalls fehlt CR; dann muss auf „nur LF" umgeschaltet werden. Das erledigt die Abfrage nach dem Kommentarblock. `Example1` ändert die `blank_line` entsprechend. Letztere wird für die Leerzeilenausgabe und als terminierende Sequenz für die Ausgabe gefüllter Sätze mit `fputs()` verwendet, wie die Anweisungen unter `generate header` demonstrieren.

Inversion der Iterations-Kontrollen / Indexzählung ab 0 aufwärts

Ein weiterer Unterschied zwischen COBOL und C wird bei der Iteration I1 deutlich: In COBOL läuft die `perform-until`-Schleife, *bis* die Endebedingung erreicht ist, also genau so, wie es die I1-Kontrolle vorsieht, während eine C-`while`-Schleife läuft, *solange* die angegebene Bedingung erfüllt ist. Das bedeutet, dass für die C-Implementierung mit `while` jede Schleifenkontrolle invertiert werden muss. Das geschieht hier, indem auf das Vorhandensein von Daten anstatt auf das Daten-Ende abgefragt wird.

Der nächste wesentliche Unterschied betrifft die Indexzählung. Die Operation 15 nimmt an, die Indizes würden ab 1 aufwärts gezählt. In C und vielen anderen Sprachen beginnt die Indexzählung dagegen mit 0; manche Sprachen lassen sogar negative Indizes zu. Das wirkt sich auch auf die Inversion der Kontrolle I2 aus: Diese Schleife darf nur so lange laufen, wie der Index `ind` kleiner als die durch `readTextin()` ermittelte Zeilenlänge – ohne schleppende Blanks – ist. Da es nicht üblich ist, ein Programm sowohl in COBOL II als auch in C oder einer ähnlichen Sprache zu implementieren, muss auf diesen Unterschied in der Liste der Operationen

normalerweise nicht hingewiesen werden. Um Zweifel auszuräumen, erscheint ein Kommentar zur jeweiligen Index-Initialisierung als sinnvoll, vor allem wenn vom Standard der jeweiligen Zielsprache abgewichen wird.

Der I2-Schleife folgen die bedingten Trace-Ausgaben der darin ermittelten Anzahlen von Leerzeichen oder anderen Zeichen. Die dafür vorgesehenen `printf()`-Aufrufe sind durch die Präprozessor-Direktiven `#if` und `#endif` geklammert. Nur wenn der per `#define` spezifizierte „Name" `TRACE_DETAIL` den Wert 1 oder größer hat, finden die Trace-Ausgaben statt. Dasselbe gilt weiter unten analog für `TRACE_MODULE`.

Licht und Schatten der Ein- und Ausgabe

Der I2-Schleife folgt die Umsetzung der zeilenbezogenen Ergebnisse. Weit eleganter, als es die geschwätzige COBOL-Arithmetik vermag, lassen sich beispielsweise die Inkrementierungen der „totals" in C formulieren. Dagegen ist die anschliessende String-Bastelei (Op. 23) ein klarer Minuspunkt für C gegenüber COBOL, welches für dasselbe Ergebnis nur vier `move`-Statements benötigt. An solchen Stellen spielt COBOL seine Stärken aus. Der Aufwand für die Zahlkonvertierungen und für das Arrangement der Ausgabe-Datenelemente wird komplett vom COBOL-Compiler und der COBOL-Laufzeitumgebung übernommen. C-Programmierer(inne)n stehen zwar unter anderem die Formatierungsleistungen von `printf()` zur Verfügung, aber er oder sie kann auf deren Erzeugnisse nicht zugreifen, um diese intern zu verwenden. Daher muss vieles selbst programmiert werden, was COBOL-Programmierer(innen) als selbstverständlichen Komfort in Anspruch nehmen. Die programmeigenen Module `itoa()` und `substr()` bezeugen dieses C-Manko. Was für die Op. 23 gilt, trifft analog auf die Op. 25 nach dem Ende der I1-Schleife zu. Die Op. 26 wurde vorgezogen, um für die Summenstrich-Zeile keinen separaten String anlegen zu müssen.

Einen weiteren wichtigen Unterschied zu COBOL stellen die `fgets()`- und `fputs()`-Aufrufe der C-Version dar. In COBOL beschafft eine `read`-Anweisung einen vollständigen Satz fester oder variabler Länge, je nachdem, was für die Datei definiert ist. Mit einer `write`-Anweisung wird stets ein vollständiger Satz ausgegeben, dem nichts nachträglich angefügt werden kann.

In C werden die Ein- und Ausgabedateien dagegen als Datenströme gesehen, deren Informationseinheiten nur durch die LF-Zeichen voneinander getrennt sind. `fgets()` beschafft nach dem Öffnen einer Datei zuerst die „vorderste" Informationseinheit als String vom Dateianfang zum ersten LF, dann von da an, Satz für Satz respektive Zeile für Zeile, die jeweils nächste Informationseinheit bis zum nachfolgenden LF, und schließlich die letzte Informationseinheit bis zum Dateiende mit oder ohne allerletztem LF. `fputs()` schreibt Strings in die Ausgabe, die erst dann als – für `fgets()` – vollständige Informationseinheiten gelten, wenn sie mit LF enden oder am Dateiende stehen. Es ist daher möglich, eine Informationseinheit in mehreren Schritten ausgabeseitig aufzubauen und mit einer separaten (CR-)LF-Ausgabe zu beenden, wie das auch im vorliegenden Fall geschieht.

Daraus ergibt sich, dass eine solche Datei nur Text, niemals Binärinformation enthalten darf, denn diese könnte zufällig ein scheinbares LF-Byte enthalten. Alle Beispiele der MSP-Buchreihe, welche sequentielle Sätze oder Zeilen oder Tupel verwenden, bauen daher auf Textdateien auf, die mit `fgets()` gelesen und mit `fputs()` geschrieben werden können. Die COBOL-typische Verarbeitung von Sätzen, die beispielsweise gepackte Dezimalzahlen enthalten, ist so allerdings überhaupt nicht darstellbar. Für die strukturierte Programmierung ist diese spezielle Option – der Ausgabe in internen Zahlformaten – allerdings unwichtig.

Um die Analogie zum COBOL-Programm Example1 so weit wie sinnvoll aufrechtzuerhalten, konkateniert das C-Programm die meisten auszugebenden Strings aus zwei und mehr Partikeln und setzt nur die (CR-)LF-Ausgabe separat ab. Selbstverständlich könnte `blank_line` stattdessen für die Op. 7, 24, 26 und 27 auch mit `header` oder `tprint` konkateniert werden.

Ein ganz wesentlicher Unterschied zur COBOL-Eingabe aus einer Fixed Block-Datei ist, dass Textdateien in Unix-Systemen und auf vielen weiteren Plattformen keine feste Satzlänge besitzen. Es ist daher möglich, dem C-Programm – und anderen Programmen – Sätze mit mehr als 55 Zeichen zuzuführen. Das sollte nicht

kommentarlos akzeptiert und durch Abschneiden der Überlängen passend gemacht werden, sondern das Programm muss dann abbrechen. Hierfür steht am Anfang der I1-Schleife folgendes:

```
if (IF_New.text_len > 55)
{
   printf("Record lengths > 55 not supported\n");
   return 6;
}
```

1.4.2 readTextin()

Dieses Zugriffsmodul beschafft die Sätze mittels `fgets()` und ermittelt ihre Nettolänge (ohne CR / LF / Null-Byte / schleppende Blanks). Hierfür setzt es eine rückwärtslaufende Schleife auf dem letzten Zeichen des Strings auf. Wenn das Ende der Datei erreicht ist, meldet readTextin den Status `'E'`, die Länge Null und einen String zurück, der nur aus einem LF-Zeichen und dem Null-Byte besteht. Das LF-Zeichen ist lediglich für die Trace-Protokollierung mit `printf()` nützlich.

1.4.3 itoa()

Der Prozedurname `itoa()` lehnt sich an die Benennung der dazu komplementären C-Standardfunktion `atoi` an. Da eine C-Funktion keinen Array rückmelden darf, wurde der Parameter `char result[]` definiert. Dafür muss im rufenden Modul ein ausreichend langer String definiert sein, von dem 1 - 9 Stellen mit dem Ergebnis der Umwandlung des `int value` in einen Ziffernstring mit führenden Leerzeichen belegt werden.

`itoa()` kann sowohl variabel lange Ziffernstrings (`digits == 0`) als auch solche fester Länge (`digits > 0`) erzeugen. Letzteres ist beispielsweise für die Punktsummen- und Zeilennummernausgaben erforderlich. Der Fall `value == 0` wird wegen der Schleifenkontrolle bei der Umwandlung separat behandelt. Die Zahl-in-Ziffer-Umwandlung kellert die Ergebnisse der wiederholten Modulo-10-Division im `stack`. Ist die erzeugte Ziffernkette bei fester Ergebnislänge zu kurz, werden vor dem Umspeichern der gekellerten Ziffern in den Ergebnis-String die fehlenden Blanks vorangestellt. Das Vorzeichen der umzuwandelnden Zahl wird nicht beachtet.

Falsche oder zu kleine Parameterwerte führen zum Abbruch des Programmlaufs mit Fehlermeldung.

1.4.4 substr()

Da es im C99-Standard keine elegante Substring-Funktion gibt (`strncpy()` ist definitiv *nicht* elegant), enthält das Programm hierfür eigenen Code. Das Modul schneidet aus dem String `src` ab der Indexposition `offset` maximal `len` Zeichen heraus. Falls ein Null-Byte, ein LF oder ein CR gefunden wird, endet das Ausschneiden vorzeitig.

Es wird geprüft, ob der Offset hinter das `src`-Ende weist oder die Länge `len` negativ ist.

Das primäre Manko dieses Moduls im speziellen und der C-Standard-Stringbearbeitungs-Funktionen im generellen ist, dass sie keine Strings exportieren und somit nicht kaskadiert werden können. Das ist beispielsweise in Java geändert worden, wo die zu C äquivalenten Methoden String-Objekte exportieren, auf denen weitere Methoden aufsetzen können.

Ausserdem stört hier wie vielerorts das C-Null-Byte, das sicherlich schon vielen Programmierer(inne)n das Genick gebrochen hat. Vordergründig scheint es eine smarte Idee zu sein, variabel lange Strings auf diese Weise zu implementieren, aber erstens sind sehr viele praktisch vorkommende Strings von fester Länge, zweitens wird damit die Datenablage mit einer Aussage über deren Länge vermengt, drittens muss das zusätzliche

Null-Byte bei jeder String-Deklaration beachtet werden, viertens muss die String-Länge – ein häufig benötigtes Attribut – zum Beispiel per `strlen()` mit einem Scan über alle anderen String-Zeichen beschafft werden, fünftens wird Ihnen sicherlich noch weiteres Negatives an dieser Idee aufgefallen sein. Rückblickend erscheint das C-Null-Byte daher genauso als Schnapsidee wie die `fputs()`-/`fgets()`-Kopplung an das LF-Zeichen…

06/2022 Geometria und Astrono(mia) an der Westseite der Ratslaube

Example1 in Java

Die nachfolgend beschriebene Implementierung von Example1 in Java® ist nicht als Musterlösung gedacht und erhebt keinen Anspruch auf Perfektion. Auch sollte daraus keine Präferenz für die eine oder andere Seite in der fortwährenden Debatte über den richtigen Weg in der objektorientierten Programmierung abgeleitet werden. Es geht ausschliesslich darum, eine funktionierende Lösung zu zeigen, ohne deren Vorteile und Schwächen aus einer höheren Perspektive zu betrachten.

2.1 Java-Klassendiagramm von Example1

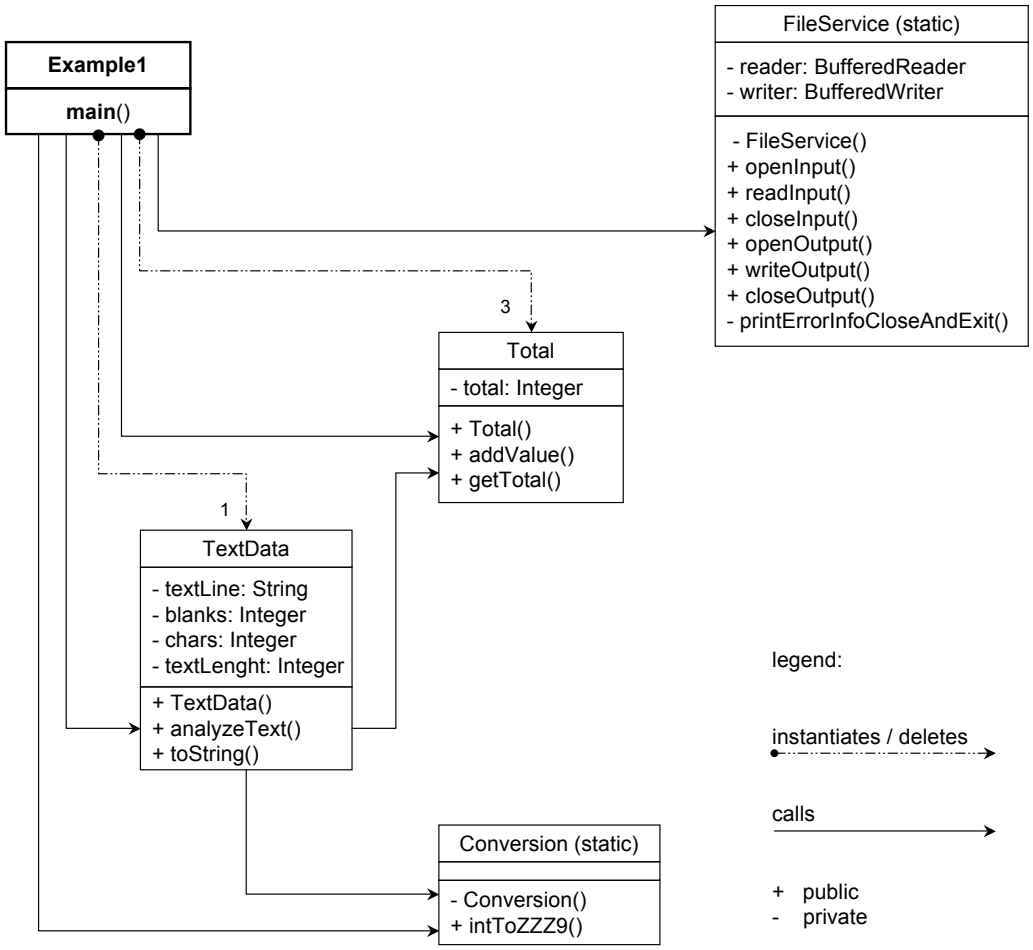


Abbildung 2.1: *Klassendiagramm der Java-Variante von „Example 1“*

Die Unterschiede zur Abbildung „Vereinfachte Relationen ohne ListData“ im Band-1-A1-Kapitel „Objektorientierte Implementierungsvorbereitungen“, welche die Beziehungen zu den Ein- und Ausgabe-Struktogrammen in den Vordergrund stellt, sind erheblich. Die Hinzunahme der `main()`-Klasse ermöglicht es, die Aufruf-Abhängigkeiten zu komplettieren. Nun ist auch sichtbar, dass die `main()` die Objekte der nicht-statischen Klassen anlegt und – nur bei `TextData` – löscht. Statische Klassen sind gekennzeichnet. Zudem bündelt die neue Klasse FileService die elementaren Zugriffsmethoden auf Textdateien.

Die Form des Klassendiagramms lehnt sich an die UML® an, verfolgt aber wegen der expliziten Darstellung der Aufrufbeziehungen auch die Absicht, den Umsteigern aus der algorithmischen Welt eine Brücke zu bauen. In dieser Buchreihe sehen ab hier alle Klassendiagramme so oder ähnlich aus. Zusätzliche semantische Elemente sind beispielsweise für die Darstellung von Vererbungsbeziehungen erforderlich. Sie werden jeweils in der Legende benannt.

Noch vor der Wiedergabe des Codes und dessen Kommentierung soll an dieser Stelle die Aufteilung der Klassen in drei Kategorien hervorgehoben werden, die sich ebenfalls durch die Buchreihe zieht:

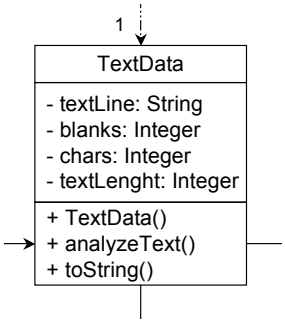
- Es gibt erstens die sogenannten Process Control-Klassen, die keine fachliche Aufgabe haben, sondern die Steuerung des Programmablaufs enthalten. Dazu gehören die Schleifenkonstrukte, innerhalb derer die anderen Klassen respektive die daraus erzeugten Objekte mit konkreten Aktionen beauftragt werden. Aktuell ist dazu nur die Klasse Example1 zu zählen, die als einzige Methode die `main()` beinhaltet. Die `main()`-Klasse ist immer statisch. Andere Process Control-Klassen – in den noch folgenden Programmen – sind dagegen durchgehend nicht-statisch, unter anderem deshalb, um die mit `static`-Deklarationen verbundenen Einschränkungen zu vermeiden.
- Zweitens sind statische Klassen erforderlich, die im vorliegenden Fall für die Ein- und Ausgabe (Klasse `FileService`), sowie für die Konvertierung von Zahlen in Ziffernstrings (Klasse `Conversion`) zuständig sind. Diese werden auch als Utility-Klassen bezeichnet. Ihr Vorteil ist, dass sie von jeder anderen Programmkomponente aus angesprochen werden können, weil sie global adressierbar sind.
- Die dritte Kategorie bilden die fachlichen Klassen. Sie repräsentieren zusammengenommen die wirklich objektorientierten Komponenten des Programms. Das sind hier die Klassen `TextData` und `Total`. Nur von der Klasse `Total` existieren zeitgleich mehrere – das heißt drei – Objekte. Von der Klasse `TextData` existiert zu jeder Zeit entweder ein oder kein Objekt. Sie könnte daher auch als statische Klasse deklariert werden, aber dann wäre das Ganze nur eine Übung, wie man Java für die traditionelle Unterprogrammtechnik nutzen – oder missbrauchen – kann.

Zugegeben, die objektorientierte Implementierung von Example1 ist nicht gerade zwingend oder auch nur irgendwie beeindruckend. Sie ist nur ein erster Schritt dahin, aus dem traditionellen JSP-Entwurf eine überzeugende OO-Implementierung abzuleiten, wie es die kommenden Kapitel zeigen werden.

Übrigens: Namen von Programmen sind im Fließtext in normaler Schrift geschrieben. Bei Java-Programmen heißen die `main()`-Klassen jedoch genauso wie das jeweilige Programm selbst; ihre Klassennamen sind daher zwecks klarer Unterscheidung im Code und in Tabellen von Klassennamen etcetera in der Schriftart Courier wiedergegeben. Im aktuellen Fall besteht das Programm Example1 folglich aus der Klasse `Example1` und weiteren Klassen. Die Überschrift des Kapitels „2.2 Java-Code von Example1“ auf Seite 33 ist aber wiederum im Standard-Format gehalten, weil Courier so groß und fett einfach schlecht aussieht. Sind jetzt alle Klarheiten beseitigt?

2.2 Java-Code von Example1

2.2.1 Klasse TextData



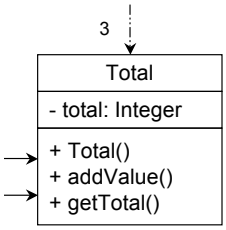

```
package example1;

import commonmethods.Conversion;

/**
 * The class TextData represents the to-be-analyzed text line and it contains
 * the related methods. The input string is read outside of the class.
 */
public class TextData
{
   private String textLine;
   private int    blanks;
   private int    chars;
   private int    textLength;

   /**
    * Constructor for objects of class TextData.
    */
   public TextData(String textLine)
   {
      this.textLine   = textLine;
      this.blanks     = 0;                                    // op. 13
      this.chars      = 0;                                    // op. 12
      this.textLength = textLine.length();                    // op. 14 init
   }

   /**
    * analyzeText() converts the string into a character array and it scans this
    * array backwards for trailing blanks (to be ignored). Then it scans the
    * remaining characters from left to right and counts the blank / non-blank
    * characters. The result values are finally added to the sum parameters.
    */
   public void analyzeText(Total sumBlanks, Total sumChars, Total sumLengths)
```

```
   {
      char[] textChar = this.textLine.toCharArray();

      // skip backward over trailing blanks
      for (int i = this.textLength-1; i >= 0 && textChar[i] == ' '; i--)
         { this.textLength--; };                                    // op. 14 cont.

      // I2: process input string & op. 15 / 16
      for (int i = 0; i < this.textLength; i++)
      {
         // S3: blank found?
         if (textChar[i] == ' ')
            { this.blanks++; }                                      // op. 18
         else
            { this.chars++; }                                       // op. 17
      }

      // op. 19,20,21
      sumChars.addValue(this.chars);
      sumBlanks.addValue(this.blanks);
      sumLengths.addValue(this.textLength);
   }

   /**
    * This method overrides the built-in toString() method in order to produce
    * a result output line.
    */
   @Override
   public String toString()
   {
      String filler55 = "                                "
                      + "                          ";

      // op. 22,23: copy input string / convert string counters
      return this.textLine.substring(0, this.textLength)
                + filler55.substring(0, 55 - this.textLength) + "  "
                + Conversion.convertIntToZZZ9(this.chars)     + "    "
                + Conversion.convertIntToZZZ9(this.blanks)    + "    "
                + Conversion.convertIntToZZZ9(this.textLength);
   }
}
```

Anmerkung

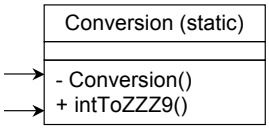
Die Methode `readLine()` der Klasse `BufferedReader` entfernt alle zeilenterminierenden Bytes, also Carriage Return, Line Feed (oder beide), sowie gegebenenfalls vorhandene Terminierungs-Null-Bytes für C-Strings. Deshalb prüft die rückwärtslaufende Schleife nur auf schleppende Blanks.

2.2.2 Klasse Total




```
package example1;

/**
 * The class Total provides very simple integer counters & functions.
 */
public class Total
{
   private int total;

   public      Total()             { this.total = 0; }

   public void addValue(int value) { this.total += value; }

   public int  getTotal()          { return this.total; }
}
```

2.2.3 Klasse Conversion




```
package commonmethods;

/**
 * The static class Conversion contains methods for inter-type conversions.
 */
public class Conversion
{
   /**
    * Dummy constructor for objects of class Conversion.
    */
   private Conversion() { }

    /**
    * intToZZZ9() converts a small positive integer into a string with
```

```
     * the format ZZZ9, i.e. up to three leading blanks and at least one digit.
     * For negative values, "----" is returned.
     * For values above 9999, ">####" is returned.
     * Note: The previous name of this method was convertIntToZZZ9().
     */
   public static String intToZZZ9(int number)
   {
      if (number < 0)
         { return "----"; };

      if (number > 9999)
         { return ">####"; };

      if (number <= 9)
         { return "   " + number; };

      if (number <= 99)
         { return "  " + number; };

      if (number <= 999)
         { return " " + number; };

      return "" + number;
   }
}
```

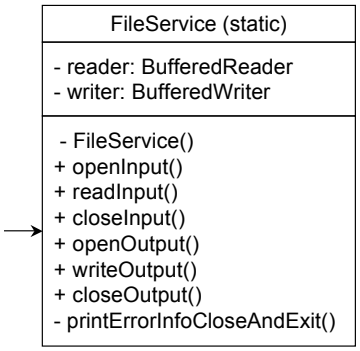
2.2.4 Klasse Example1

```
package example1;

import commonmethods.Conversion;

public class Example1
{
   public static void main(String[] args)
   {
      System.out.println("*** Example1 ***");

      switch(args.length)
      {
         case 0:  System.out.println("\t Path 0 (input) " +
                                     "& path 1 (output) missing!");
                  return;
         case 1:  System.out.println("\t Path 0: " + args[0]);
                  System.out.println("\t Path 1 (output) missing!");
                  return;
         case 2:  /* for (int i = 0; i < args.length; i++)
```

```
                System.out.println("\t Path " + i + ": " + args[i]); */
            break;
   default: for (int i = 0; i < args.length; i++)
            { System.out.println("\t Path " + i + ": " + args[i]); }
            System.out.println("\n Too many paths, or " +
                               "apostrophes missing!");
            return;
}

String inputString;
String filler55 = "                                "
                + "                          ";

// open input file: op. 1
FileService.openInput(args[0]);

// open output file: op. 2
FileService.openOutput(args[1]);

// op. 6: read 1st record
inputString = FileService.readInput();

// op. 7,8: generate header
FileService.writeOutput("Text" + filler55.substring(0, 52) +
                        "chars  blanks  length");
FileService.writeOutput(" ");

// instatiate totals & op. 9,10,11
Total totalChars   = new Total();
Total totalBlanks  = new Total();
Total totalLengths = new Total();

// I1: build extended strings
while (inputString != null)
{
   if (inputString.length() > 55)
   {
      System.out.println("\n Record lengths > 55 not supported! \n");
      return;
   }

   // determine string characteristics
   TextData textData = new TextData(inputString);
   textData.analyzeText(totalBlanks, totalChars, totalLengths);

   // op. 24: output list body line
   System.out.println(textData);                              // test
```

```
            FileService.writeOutput(textData.toString());

            // destroy temporary object
            textData = null;

            // op. 6: read next record
            inputString = FileService.readInput();
         }

         // op. 26,25,27: generate footer
         FileService.writeOutput(filler55 + "  ====    ====    ====");
         FileService.writeOutput(filler55.substring(0, 49)         + "totals  "
                   + Conversion.convertIntToZZZ9(totalChars.getTotal())  + "    "
                   + Conversion.convertIntToZZZ9(totalBlanks.getTotal()) + "    "
                   + Conversion.convertIntToZZZ9(totalLengths.getTotal()));

         // op. 3,4: close files
         FileService.closeInput();
         FileService.closeOutput();

         System.out.println("*** Successful completion ***");

         // stop: op. 5
         // (implicit return)
      }
}
```

2.2.5 Klasse FileService




```
package example1;

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.FileInputStream;
import java.io.FileOutputStream;
```

```
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;

/**
 * The static class FileService contains methods for opening, reading / writing,
 * and closing simple text files.
 */
public final class FileService
{
   private static BufferedReader reader;
   private static BufferedWriter writer;

   /**
    * Dummy constructor for objects of class FileService.
    */
   private FileService() { }

   /**
    * openInput() creates a file input stream for the input file name, i.e.,
    * it opens this file for input. Then an input stream reader for this stream
    * is made, specifying the encoding "ISO-8859-1" for the character set.
    * Finally, a buffered reader object is created and its reference is stored.
    */
   public static void openInput(String inFileName)
   {
      FileInputStream   inputStream  = null;
      InputStreamReader streamReader = null;

      try {
         inputStream = new FileInputStream(inFileName);
      }
      catch(Exception ex) {
         printErrorInfoCloseAndExit(ex, 1,
                                    "Input file OPEN failed");
      }

      try {
         streamReader = new InputStreamReader(inputStream, "ISO-8859-1");
      }
      catch(Exception ex) {
         printErrorInfoCloseAndExit(ex, 2,
                                    "Input stream encoding not supported");
      }

      reader = new BufferedReader(streamReader);
   }
```

```
/**
 * readInput() reads an input record and returns it as a string.
 */
public static String readInput()
{
   String inputString = null;

   try {
      inputString = reader.readLine();
   }
   catch(Exception ex) {
      printErrorInfoCloseAndExit(ex, 3,
                                 "Input READ failed");
   }

   // System.out.println("\t inputString = " + inputString);
   return inputString;
}

/**
 * closeInput() closes the input file.
 */
public static void closeInput()
{
   try {
      reader.close();
   }
   catch(Exception ex) {
      reader = null;
      printErrorInfoCloseAndExit(ex, 4,
                                 "Input file CLOSE failed");
   }
}

/**
 * openOutput() creates a file output stream for the output file name,
 * i.e., it opens this file for output. Then an output stream reader for this
 * stream is made, specifying the encoding "ISO-8859-1" for the character set.
 * Finally, a buffered writer object is created and its reference is stored.
 */
public static void openOutput(String outFileName)
{
   FileOutputStream   outputStream = null;
   OutputStreamWriter streamWriter = null;

   try {
      outputStream = new FileOutputStream(outFileName);
```

```
   }
   catch(Exception ex) {
      printErrorInfoCloseAndExit(ex, 5,
                                 "Output file OPEN failed");
   }

   try {
      streamWriter = new OutputStreamWriter(outputStream, "ISO-8859-1");
   }
   catch(Exception ex) {
      printErrorInfoCloseAndExit(ex, 6,
                                 "Output stream encoding not supported");
   }

   writer = new BufferedWriter(streamWriter);
}

/**
 * writeOutput() writes an output record from a string parameter.
 */
public static void writeOutput(String outputString)
{
   // System.out.println("\t outputString = " + outputString);

   try {
      writer.write(outputString);
   }
   catch(Exception ex) {
      printErrorInfoCloseAndExit(ex, 7,
                                 "Output WRITE failed");
   }

   try {
      writer.newLine();
   }
   catch(Exception ex) {
      printErrorInfoCloseAndExit(ex, 8,
                                 "Output newline failed");
   }
}

/**
 * closeOutput() closes the output file.
 */
public static void closeOutput()
{
   try {
```

```
         writer.close();
      }
      catch(Exception ex) {
         writer = null;
         printErrorInfoCloseAndExit(ex, 9,
                                    "Output file CLOSE failed");
      }
   }

   /**
    * printErrorInfoCloseAndExit() combines the terminating actions.
    */
   private static void printErrorInfoCloseAndExit(Exception excp,
                                                  int       exitCode,
                                                  String    errorMsg)
   {
      System.out.println(errorMsg);
      excp.printStackTrace();

      if (reader != null)
      {
         try {
            reader.close();
         }
         catch(Exception em) {
            System.out.println("Input file emergency CLOSE failed");
         }
      }

      if (writer != null)
      {
         try {
            writer.close();
         }
         catch(Exception em) {
            System.out.println("Output file emergency CLOSE failed");
         }
      }

      System.exit(exitCode);
   }
}
```

2.3 Java-Code-Durchgang

Der Programmtext einschliesslich aller Deklarationen und Definitionen ist deutlich länger als der COBOL-II-Text. Dies liegt natürlich an dem für die Klassen betriebenen Aufwand. Dieser hat sich jedoch beim Testen rasch bezahlt gemacht. Die Implementierung verlief trotz der Verwendung einer neuen Plattform erstaunlich problemlos. Fehler konnten dank der aussagekräftigen Fehlermeldungen des Compilers rasch gefunden und behoben werden. Der Editor unterstützt die Programmierung nicht nur durch das Einfärben des Codes, sondern auch durch feine Codeblocklinien am linken Rand. Codeblöcke können „zugeklappt" und „aufgeklappt" werden. NetBeans® unterstützt die Entwicklung ausserdem durch Fenster, die zusätzlich zur üblichen Projekt- und Navigator-Ansicht für weitere Analysen geöffnet werden können.

Die Plattformunabhängigkeit der Sprache Java ermöglicht ihr eine weit größere Verbreitung, als sie beispielsweise C++ beschieden war. Dennoch sollte nicht unterschätzt werden, wie verbreitet C++, C#, Objective C und andere, ähnliche Sprachen – noch immer – sind. Gewiss beherrschen auch nicht alle Leser Java. Daher ist es erforderlich, das einfache Java-Programm Example1 hier näher zu betrachten. Vorab ist es jedoch anzuraten, sich mit dieser Sprache zumindest oberflächlich bekanntzumachen. Je nach Vorbildung sollte der Einstieg ein anderer sein. So bringt es beispielsweise das Buch „Java ist auch eine Insel" von Christian Ullenboom [15] auf stolze 1258 Seiten, von denen die Seiten 97 bis 223 als Einführung in die Grundlagen der Sprache verstanden werden können.

Der didaktische Ansatz Ullenbooms richtet sich an Programmierer(innen), die bereits Java-Grundkenntnisse besitzen und ihr Wissen nun erheblich intensivieren möchten. Wer dagegen mit Java ganz von vorn beginnt, sollte sich ein Buch aussuchen, das gezielt als Einführung konzipiert wurde. Anfänger im Programmieren sind vermutlich auch damit überfordert, obwohl die objektorientierte Programmierung angeblich die natürlichere Sichtweise auf ein zu lösendes Problem eröffnet. Diese kühne Behauptung wagt der Autor aufgrund jahrzehntelanger Erfahrung allerdings zu bezweifeln.

Zurück zum Thema: In diesem Durchgang sollen die wesentlichen Aspekte der Implementierung dargestellt werden, wobei Java-Grundkenntnisse vorausgesetzt werden. Der Durchgang beginnt mit der neuen Ein- und Ausgabeklasse, widmet sich dann der Exception-Behandlung und wendet sich schließlich den aufgabenspezifischen Klassen zu. Die nachfolgenden Programme werden übrigens nicht so aufwendig wie Example1 beschrieben, weil der Umfang der Praxis-Bände in der Buchreihe sonst unvertretbar groß würde.

Übrigens: Example1 ist eine einfache Java-Application, besitzt also kein grafisches Interface, sondern verwendet das in NetBeans integrierte Ausgabefenster für ihre Meldungen.

2.3.1 Utility-Klasse FileService

Diese Klasse besteht aus einer Kollektion von Methoden zum Öffnen, zum Lesen oder Schreiben und zum Schließen von Textdateien, die den `ISO-8859-1`-Code verwenden. Mit dieser Codetabelle wird eine breite Abdeckung der Sonderzeichen westeuropäischer Sprachen erzielt. Die Methoden und die Exception-Behandlung sind so einfach wie möglich gehalten; der Code wird hier daher nicht näher beschrieben.

Es sind nur zwei Klassenvariable vorhanden, nämlich `reader` und `writer`, die zudem nach aussen unsichtbar sind. Für das Öffnen und Schließen gilt jeweils eine dreistufige Klassenhierarchie: Nach dem Erzeugen eines temporären `FileInputStream`- oder `FileOutputStream`-Objekts anhand des jeweiligen Dateinamens erfolgt die Codetafel-Zuordnung durch das Anlegen eines ebenfalls temporären `StreamReader`- oder `StreamWriter`-Objekts. Daraus wird wiederum je ein `BufferedReader`- oder `BufferedWriter-Objekt` abgeleitet und dessen Handle in `reader` oder `writer` gespeichert. Daher bleiben letztere Objekte resident.

Die Lese- und Schreib-Methoden sind nicht ganz symmetrisch, weil nach dem Schreiben eines Ergebnissatzes respektive einer Ergebniszeile jeweils ein Newline-Zeichen (LF = Line Feed) angehängt wird. Das ist der Zeilen- oder Satzbegrenzer in Textdateien unter Unix. Der ausgebende Code muss somit nicht berücksichtigen, auf

welchem Unix-Derivat er läuft. Für andere Plattformen müsste gegebenenfalls ein CR-Zeichen (CR = Carriage Return) oder sogar CR-LF angehängt werden. Diese Feinheiten werden hier aber nicht berücksichtigt.

Die Methoden zum Öffnen, Lesen, Schreiben und Schließen besitzen keine Zustandssteuerung, könnten also vom Programm in beliebiger Reihenfolge aufgerufen werden. Dabei wäre allerdings mit Exceptions zu rechnen, die von den Methoden abgefangen werden und die letztlich zum Lauf-Abbruch führen. Es sind aber meist banale Ursachen, wie beispielsweise eine nicht vorhandene Eingabedatei, die Exceptions auslösen. Nach der Ausgabe eines fehlerspezifischen Texts und des Stack-Trace – für die Fehlersuche – werden die (noch) offenen Dateien geschlossen und der Programmlauf unter Ausgabe des `exitCode` beendet. Tritt bei einem Close-Aufruf eine Exception ein, so wird die zugehörige Referenz auf `reader` oder `writer` gelöscht, um in `printErrorInfoCloseAndExit()` dasselbe nicht noch einmal zu versuchen. Exceptions in letzterer Methode werden nur protokolliert.

Diese Exception-Behandlungen konnten nur teilweise, nämlich mit falschen Pfad- und Dateinamen, getestet werden. Beim Lesen, Schreiben und Schließen ließ sich keine Exception provozieren, ohne absichtlich Fehler im Code einzubauen.

Es liegt auf der Hand, dass das eine extrem simple Lösung ist. Sie erlaubt nur eine Eingabedatei und eine Ausgabedatei. Daher wird ab dem Unterkapitel „Zweiter Vorschlag" des Band-1-A2-Kapitels „File Services" eine weitaus leistungsfähigere, aber zugleich aufwendigere Utility-Klasse für Zugriffe auf Textdateien konzipiert und im A2-Kapitel „File Services in Java" implementiert. Immerhin ermöglicht die Klasse `FileService` es den aufgabenspezifischen Klassen, alle zugriffsspezifischen Aspekte auszulagern. Dadurch wird der Code der letzteren Klassen deutlich übersichtlicher.

2.3.2 Hinweis zur Exception-Behandlung

Es gibt Java-ähnliche Sprachen, welche die Exception-Behandlung in das Belieben des Programmierers oder der Programmiererin stellen. Im Gegensatz dazu müssen Java-Methodenaufrufe, die Exceptions auslösen können, in `try`-Blöcken ausgeführt werden, denen `catch`-Blöcke im Allgemeinen unmittelbar folgen (zumindest erzwingt beispielsweise NetBeans das). Da die Ein- und Ausgabeoperationen über die Programmstruktur verstreut sind, müsste bei unmittelbarer Realisierung dieser Operationen in den Methoden dort jedesmal die `try`- und `catch`-Logik codiert werden. Das würde sehr unübersichtlich und redundant sein, sowie eine generalisierte Fehlerbehandlung deutlich erschweren. Auch deshalb erscheint die gemeinsam genutzte Klasse `FileService` als sinnvoll, weil sie alle erforderlichen Anweisungen bündelt und dafür die passenden Methoden anbietet.

Da es sich hier nicht um ein Produktionsprogramm handelt, erscheint diese Form der Fehlerbehandlung als völlig ausreichend. Für Programme, die in Fehlerfällen nicht abbrechen dürfen, muss erheblich mehr Aufwand als hier getrieben werden.

2.3.3 main()

Am Anfang der Java `main()` sehen wir die folgenden Zeilen:

```
package example1;

import commonmethods.Conversion;

public class Example1
{
   public static void main(String[] args)
```

```
{
   System.out.println("*** Example1 ***");
```


In Java werden die Codepakete zu „Projects" zusammengefasst. Package-Namen werden von Eclipse stets klein geschrieben; andere IDEs – Integrated Development Environments – wie zum Beispiel BlueJ und NetBeans sind da nicht so wählerisch. Alle Java-Klassen von Example1 gehören zum `package example1`. Jede Java-Klasse, die sich auf Code ausserhalb ihres Packages bezieht, muss diesen mit der Anweisung `import` beiziehen, wobei NetBeans tatkräftige Unterstützung leistet. Importierte Klassen gehören entweder zu den Java-Standard-Bibliotheken, von denen es inzwischen extrem viele gibt, oder sie sind gekaufte respektive selbst erstellte Klassen. Die `main()`-Klasse und die Klasse `TextData` benötigen dieselbe simple Konvertierungsmethode aus dem Package `commonmethods` und der darin enthaltenen Klasse `Conversion`. Um das Package `commonmethods` einzubinden, wird es in NetBeans als JAR-Bibliothek – in den Projekteigenschaften – hinzugefügt. Dieses Package wurde angelegt, um Konvertierungsmethoden aufzunehmen, die von mehreren Programmen benötigt werden.

Die Deklaration der Java `main()`-Methode beginnt, wie es sich in einer Java-Utility-Klasse gehört, mit `public static`. Abweichend von anderen Sprachen, beispielsweise C++, gibt die `main()`-Methode keinen Ergebniswert zurück. Sie erhält ihre Parameter als `String`-Array mit dem Namen `args`: `String[] args`.

Am Anfang des ausführbaren Codes wurde eine `println()`-Anweisung aufgenommen, um die Banner Line des Programms auszugeben. Komplementär dazu steht am Ende der `main()` eine Ausgabe, welche die erfolgreiche Ausführung des Programms meldet. Übrigens: Ein Java-Programm, das eine Main Class besitzt, listet unter NetBeans alle Text-Laufausgaben in einem separaten Output-Fenster auf. Das erleichtert es auch erheblich, Protokoll-Kopien anzufertigen.

Erst auf den zweiten Blick fällt auf, dass die Klasse `System` aus dem Package `java.lang` nicht importiert werden muss. Andere Kernklassen von Java, mit denen beispielsweise Arrays angelegt und bearbeitet werden, erfordern dagegen den Import. Die Angabe `out` ist eine Klassenvariable, die in `System` mit `static PrintStream` deklariert ist und den Standard Output Stream repräsentiert. `println()` ist folglich eine – mehrfach überladene – Methode von `PrintStream`, das etliche weitere Ausgabefunktionen und Ausgabe-Hilfsfunktionen enthält. Ein interessanter Aspekt von `println()` ist, dass es bei der Übergabe eines nicht schon als Parameter vorgesehenen Objekts dessen `toString()`-Methode aufruft. Davon wird in Example1 ebenfalls Gebrauch gemacht.

Die anschliessende Prüfung der Parameter-Anzahl sorgt dafür, dass zumindest alle erwarteten Parameter – unabhängig von ihrer Korrektheit – vorhanden sein müssen, bevor die eigentliche Verarbeitung beginnt:

```
switch(args.length)
{
   case 0:  System.out.println("\t Path 0 (input) " +
                               "& path 1 (output) missing!");
            return;
   case 1:  System.out.println("\t Path 0: " + args[0]);
            System.out.println("\t Path 1 (output) missing!");
            return;
   case 2:  /* for (int i = 0; i < args.length; i++)
                System.out.println("\t Path " + i + ": " + args[i]); */
            break;
   default: for (int i = 0; i < args.length; i++)
            { System.out.println("\t Path " + i + ": " + args[i]); }
            System.out.println("\n Too many paths, or " +
```

```
                                        "apostrophes missing!");
                return;
        }
```

Wenn es nicht exakt zwei Parameter sind, welche die vollständigen Pfade zu den Ein- und Ausgabedateien benennen, bricht der Lauf mit Ausgabe einer passenden Fehlermeldung ab. Sind es zwei, dann werden die Parameter nicht aufgelistet, es sei denn, die Kommentarklammern würden entfernt. Bei mehr als zwei Parametern ist es naheliegend, dass versäumt wurde, die Dateinamen in Apostrophe einzuschließen, sofern sie Leerzeichen enthalten, denn die Laufzeitumgebung deutet ungeschützte Leerzeichen als Trenner zwischen Parametern.

NetBeans ermöglicht die Vorgabe von Laufparametern, indem zuerst der Menupunkt „Als Hauptprojekt wählen" in der Drop Down-Liste unter „Ausführen" gewählt wird, woraufhin rechts davon die Liste der vorhandenen Projekte erscheint. Darin wählt man das Projekt aus, das nun getestet respektive produktiv gestartet werden soll. Dieser Schritt ist nur einmal erforderlich, bis man auf dieselbe Weise zu einem anderen Projekt wechselt.

Anschließend klickt man im selben Menu auf „Projektkonfiguration festlegen" und dann rechts davon auf „Anpassen...". Dies öffnet das Panel „Projekteigenschaften festlegen", in dem zahlreiche Voreinstellungen vorgenommen werden können. Ganz links muss unter „Kategorie" der Punkt „Ausführen" gewählt werden, falls noch nicht geschehen. (Der Punkt „Bibliotheken" ermöglicht es, unter dem Reiter „Übersetzen" externe Quellen anzugeben, im aktuellen Fall also den JAR „CommonMethods" hinzuzufügen). Rechts davon können mehrere Konfigurationen hinterlegt werden, was das komfortable Umschalten zwischen verschiedenen Lauf-Vorgaben ermöglicht. Gewöhnlich wird dort die „<Standardkonfiguration>" angezeigt.

„Runtime Platform" und „Hauptklasse" werden von NetBeans vorbelegt. Die Zeile „Argumente" nimmt die Laufparameter auf. Falls man sich geirrt hat, kann man das Panel per „Abbrechen" folgenlos verlassen; andernfalls werden die neuen Vorgaben durch den Klick auf „OK" oder durch Betätigen der Return-Taste gespeichert. Erst dann kann das Programm mit den gespeicherten – neuen oder alten – Vorgaben a) durch Klick auf „Hauptprojekt ausführen" in der nun bekannten Drop Down-Liste, b) durch Klick auf das grüne Dreieck in der Werkzeugleiste oder c) mit der Funktionstaste 6 gestartet werden.

Zurück zum Programm: Nach der Prüfung der Parameter-Anzahl – ohne Prüfung der Parameterinhalte – folgen `String`-Deklarationen und -Zuweisungen. Das Öffnen der Dateien, das Vorlesen und die Ausgabe der beiden Header-Zeilen erfolgen mit Hilfe der Utility-Klasse `FileService`. Beim Öffnen wird die Ausgabedatei entweder neu angelegt oder geleert, falls schon vorhanden. Anschließend werden die drei `Total`-Objekte instantiiert und initialisiert. Die Klassen `Total`, `TextData` – bis auf `analyzeText()` / `toString()` – und `Conversion` sind im übrigen derart simpel, dass sie im Code-Durchgang nicht erklärt werden müssen.

Die anschließende Kontrolle I1 verwendet unmittelbar die Rückmeldung von `readInput()` vom Vorlesen oder Nachlesen. Der Schleifenkopf lautet daher wie folgt:

```
// I1: build extended strings
 while (inputString != null)
 {
```

Wie das C-Programm muss die Java-Variante damit rechnen, dass längere Sätze als zulässig angeboten werden. Dieser Fall wird wie folgt abgefangen:

```
    if (inputString.length() > 55)
    {
       System.out.println("\n Record lengths > 55 not supported! \n");
```

```
            return;
        }
```

Hier wird die Methode `length()` des `String`-Objekts `inputString` erneut genutzt.

Der zuvor gezeigte Code bis auf I1 ist nicht Bestandteil des methodischen Entwurfs, sondern ergibt sich aus den speziellen Eigenschaften der für die Programmausführung gewählten Plattform und der Entscheidung, wie auf welchen Fehler reagiert werden soll. Hier stellt sich natürlich die Frage nach der Abgrenzung zwischen den beim Entwurf berücksichtigten Datenstrukturen samt den Eigenheiten der Operationen, und den als implementierungsabhängig bezeichneten Zusätzen.

Diese Abgrenzung kann bei unbekannter Zielsprache oder bei einer solch großen Bandbreite wie hier – COBOL II kontra C / Java – nur sehr allgemein formuliert werden. Ist dagegen klar, womit das Programm implementiert werden soll, müssen alle Details beim Entwurf berücksichtigt werden, die das Verhalten zur Laufzeit signifikant beeinflussen. Bevor Sie jetzt anfangen, über das Wort „signifikant" zu philosophieren, versetzen Sie sich bitte in die Situation des Analytikers, der für irgendwelche – ihm unbekannten – Programmierer(innen) die Vorgaben erstellt. Dann werden Sie eher geneigt sein, etwas klar zu modellieren, als wenn Sie das Programm selbst zu implementieren vorhaben.

Nach der Satzlängenprüfung finden die ersten Verarbeitungsschritte statt. Mit

```
        // determine string characteristics
        TextData textData = new TextData(inputString);
        textData.analyzeText(totalBlanks, totalChars, totalLengths);
```

baut das Programm ein `TextData`-Objekt `textData` aus dem jeweiligen `inputString` auf und unterzieht es in der `TextData`-Methode `analyzeText()` der Untersuchung auf Leerzeichen beziehungsweise druckbare Zeichen. Diese Methode erhält die Handles der `Total`-Objekte als Parameter, um intern mit `addValue()` die ermittelten satzbezogenen Ergebnisse aufzukumulieren. Durch

```
        // op. 24: output list body line
        System.out.println(textData);                              // test
        FileService.writeOutput(textData.toString());
```

erfolgt die Ausgabe der Ergebniszeile in das Laufprotokoll und in die Ausgabedatei. In beiden Fällen wird `textData` ausgewertet, aber `println()` ruft bei ihm unbekannten Objekten intern die Methode `toString()` auf, was `writeOutput()` nicht kann. Daher muss dort `toString()` explizit aufgerufen werden, wofür ein „Override" der von `Object` geerbten Methode `toString()` in der Klasse `TextData` erforderlich ist:

```
@Override
public String toString()
{
   String filler55 = "                                "
                   + "                          ";

   // op. 22,23: copy input string / convert string counters
   return this.textLine.substring(0, this.textLength)
             + filler55.substring(0, 55 - this.textLength) + "  "
             + Conversion.convertIntToZZZ9(this.chars)     + "    "
             + Conversion.convertIntToZZZ9(this.blanks)    + "    "
```

```
                + Conversion.convertIntToZZZ9(this.textLength);
    }
```

Die String-Konkatenationen mit dem Zeichen `'+'` erlauben eine sehr kompakte Schreibung. Das erste Return-Argument

```
            this.textLine.substring(0, this.textLength)
```

wirkt anfangs befremdlich. Warum muss der Netto-String mit der Länge `textLength` aus `textLine` ausgeschnitten werden? Das kann hier noch nicht erklärt werden, sondern muss bis zum nächsten Unterkapitel warten. Die danach folgende Zeile

```
                + filler55.substring(0, 55 – this.textLength) + "  "
```

sorgt dafür, dass Zeilen mit weniger als 55 Zeichen aufgefüllt werden, indem variabel viele Leerzeichen aus dem `String filler55` ausgeschnitten und mit dem Netto-String konkateniert werden. Die drei anschliessenden Zeilen nutzen die Utility-Klasse `Conversion`, deren Methode `intToZZZ9()` die übergebene Integer-Zahl in einen vierstelligen String umsetzt. Führende Nullen bei Zahlen unterhalb 1000 werden durch Leerzeichen ersetzt. Das COBOL-Format ZZZ9 hat exakt diese Bedeutung und ist daher Namenspate. Die Zahlenspalten werden durch zusätzliche Leerzeichen voneinander getrennt.

Die `main()`-Anweisungen

```
        // destroy temporary object
        textData = null;

        // op. 6: read next record
        inputString = FileService.readInput();
    }
```

schließen die I1-Schleife ab. Die Zuweisung von `null` zu `textData` führt nicht unmittelbar zur Zerstörung des `TextData`-Objekts `textData`, sondern zeigt nur der Garbage Collection in der Laufzeitumgebung an, dass dieses Objekt nicht mehr referenziert wird und somit gelöscht werden kann. Danach wird nachgelesen, wobei ein neues `textData`-Objekt entsteht. Final erfolgt die Ausgabe der Fußzeilen:

```
    // op. 26,25,27: generate footer
    FileService.writeOutput(filler55 + "  ====    ====    ====");
    FileService.writeOutput(filler55.substring(0, 49) + "totals  "
              + Conversion.convertIntToZZZ9(totalChars.getTotal())  + "    "
              + Conversion.convertIntToZZZ9(totalBlanks.getTotal()) + "    "
              + Conversion.convertIntToZZZ9(totalLengths.getTotal()));
```

Auch hier kommt die Methode `intToZZZ9()` zur Anwendung. Sie holt sich ihr Argument mit der trivialen Methode `getTotal()` aus der jeweiligen `Total`-Instanz. Da alle String-Längen bekannt und konstant sind, ist der Code trivial. Die `main()` besitzt hierfür eine eigene Instanz von `filler55`.

Danach folgen nur noch die Schlussoperationen

```
    // op. 3,4: close files
```

```
FileService.closeInput();
FileService.closeOutput();

System.out.println("*** Successful completion ***");

// stop: op. 5
// (implicit return)
```

Das Programm muss nicht explizit stoppen, sondern der Rücksprung aus der `main()` in die Laufzeitumgebung signalisiert letzterer das Ende des Laufs.

2.3.4 analyzeText()

Auf dem `TextData`-Objekt `textData` operiert die Methode `analyzeText()`, die vor der Zeichenzählung die Netto-Stringlänge feststellt. Hierzu wandelt sie zuerst den `String` `textLine` in einen `char`-Array `textChar` um, der in der danach folgenden Schleife von hinten nach vorn durchsucht wird, um schleppende Leerzeichen zu übergehen. Wenn eine Zeile nur aus Leerzeichen besteht, wird ihre Länge `textLength` schließlich auf Null herabgezählt. Ist die Zeile leer, enthält also überhaupt keine Zeichen, dann ist auch der Array leer und somit die Schleifenendeabfrage sofort erfüllt. Der Original-String `textLine` wird durch dieses Vorgehen nicht tangiert. Daher darf in dem `toString()`-Override in der Klasse `TextData` nicht die Original-Stringlänge verwendet werden.

Der Kommentar neben dem Schleifenkern bezieht sich auf die Operation 14: len := LENGTH(in_string). Die Op. 14 wurde durch die Beschaffung der Brutto-Stringlänge beim Einlesen begonnen und wird hier folglich zu Ende geführt.

```
char[] textChar = this.textLine.toCharArray();

// skip backward over trailing blanks
for (int i = this.textLength-1; i >= 0 && textChar[i] == ' '; i--)
   { this.textLength--; };                                    // op. 14 cont.
```

Die Zeichenzählung basiert auf der Nettolänge ohne schleppende Blanks:

```
// I2: process input string & op. 15 / 16
for (int i = 0; i < this.textLength; i++)
{
   // S3: blank found?
   if (textChar[i] == ' ')
      { this.blanks++; }                                      // op. 18
   else
      { this.chars++; }                                       // op. 17
}
```

Die Operationen 15 und 16 zum Setzen und Inkrementieren des Index sind im Schleifenkopf zusammengefasst; abweichend von der Spezifikation startet der Index mit Null. Die Kontrolle S3 prüft jedes einzelne Zeichen darauf, ob es ein Leerzeichen ist, wobei im Gegensatz zur COBOL- oder C-Implementierung zu beachten ist, dass je nach Zeichensatz die Zeichen im `char`-Array `textChar` mehr als ein Byte pro Element belegen können.

Hier, im Kern der Verarbeitung, lohnt es sich, einen Moment lang innezuhalten und sich zu fragen: Geht es nicht auch anders? Immerhin wird hier ein zusätzlicher Array benötigt, anstatt den `String textLine` direkt auszuwerten, beispielsweise mittels `substr()`. Bei einem Laufzeitvergleich müsste vermutlich eine große Zeichenmenge durchsucht werden, um überhaupt eine signifikante Differenz zwischen den alternativen Implementierungen feststellen zu können. Optimierungen, die keinen evidenten Laufzeitgewinn erbringen, sind doch fragwürdig, oder nicht?

Tatsächlich ist jedoch oftmals eine Vielzahl solcher Mikroentscheidungen dafür verantwortlich, ob ein Programm performant läuft. Wer weiss denn genau, welcher Aufwand bei jedem `substr()`-Aufruf getrieben werden muss? Immerhin müsste jedes einzelne Zeichen per `substr()` isoliert werden, um es zu bewerten. Das kann bei einer großen Zeichenmenge durchaus insgesamt erheblich länger dauern, als die hier gezeigte einmalige Umsetzung in einen Array pro String mit nachfolgender Indexlogik. Es geht in diesem Beispiel ja schließlich darum, eine Übung zu absolvieren, die Hinweise für die Gestaltung realer Produktionsprogramme zu geben vermag. Jedenfalls spricht nichts gegen die Array-Lösung.

Abschließend aktualisiert `analyzeText()` die drei `Total`-Objekte, indem es die neuen Zählergebnisse mit Hilfe der klasseninternen Methode `addValue()` aufkumuliert, wie oben schon beschrieben:

```
// op. 19,20,21
sumChars.addValue(this.chars);
sumBlanks.addValue(this.blanks);
sumLengths.addValue(this.textLength);
```

2.4 Verarbeitung von diakritischen Zeichen

Um zu zeigen, dass mit der ISO-8859-1-Formatierung tatsächlich solche Zeichen korrekt wiedergegeben werden, folgt hier eine Laufprotokoll-Ausgabe der letzten 4 Codereihen (xC0 - xFF):

```
Text                                                chars  blanks  length

Unusual characters                                     17       1      18
À Á Â Ã Ä Å Æ Ç È É Ê Ë Ì Í Î Ï                        16      15      31
Ð Ñ Ò Ó Ô Õ Ö × Ø Ù Ú Û Ü Ý Þ ß                        16      15      31
à á â ã ä å æ ç è é ê ë ì í î ï                        16      15      31
ð ñ ò ó ô õ ö ÷ ø ù ú û ü ý þ ÿ                        16      15      31
*** Successful completion ***
```

Die von Example1 erzeugte Textdatei musste allerdings erst in Microsoft Word als Unicode-Text eingeschleust und dann mit der Codierung „Westeuropäisch (Windows Latin 1)“ geöffnet werden. Das Ergebnis ist

```
Text                                                chars  blanks  length

Unusual characters                                     17       1      18
À Á Â Ã Ä Å Æ Ç È É Ê Ë Ì Í Î Ï                        16      15      31
Ð Ñ Ò Ó Ô Õ Ö × Ø Ù Ú Û Ü Ý Þ ß                        16      15      31
à á â ã ä å æ ç è é ê ë ì í î ï                        16      15      31
ð ñ ò ó ô õ ö ÷ ø ù ú û ü ý þ ÿ                        16      15      31
                                                     ====    ====    ====
                                              totals    81      61     142
```

2.5 Bewertung des Java-Codes von Example1

Mal ganz abgesehen von Schönheitsfragen oder irgendwelchen Orthodoxie-Aspekten erscheint es als sinnvoll, den Java-Code von Example1 abschließend im Licht von JSP/MSP zu bewerten:

- Die ersten und wichtigsten Fragen lauten: Ist es gelungen, die auf JSP/MSP basierende Modellierung tatsächlich in eine objektorientierte Lösung umzusetzen? Ist somit die Forderung vollständig erfüllt, dass der dynamische Programmablauf exakt der Programmstruktur folgt?

  Die Antwort auf diese Fragen lautet natürlich „Ja", denn sonst wären alle weiteren Bemühungen in dieser Richtung sinnlos und JSP/MSP blieben weiterhin nur dem algorithmischen Denken verhaftet. Das Erstaunliche beim Schreiben dieses allerersten objektorientierten Beispiels war, dass sich alles folgerichtig „von selbst" ergab. Es war nicht nötig, irgendwelche Tricks anzuwenden oder krampfhaft Strukturen „passend" zu machen. Die Aufteilung des Codes in Klassen und Methoden, die Zuordnung der Klassen- und Instanzvariablen, sowie die Plazierung der Operationen und Kontrollen verliefen harmonisch und rasch.

  Die zweite Frage erfordert, den aus der Programmstruktur stammenden – geplanten – Kontrollfluss und den realen Kontrollfluss der Java-Implementierung Schritt für Schritt miteinander zu vergleichen. Das ist sowieso vor dem allerersten Test nötig, um sicherzustellen, dass der Entwurf getreulich umgesetzt wurde. Das Ergebnis lautet: Beide sind deckungsgleich, auch wenn die Realisierung sich nun über mehrere Klassen erstreckt. Ist der statische Kontrollfluß identisch, dann wird es auch der dynamische sein.
- Wieviel Code stammt denn nun tatsächlich aus dem Entwurf, und wieviel kam durch die Implementierung auf dieser speziellen Plattform dazu?

  Um das besser nachvollziehbar zu machen, und um die abschließende Qualitätskontrolle zu erleichtern, wurden die Operationen und Kontrollen wie bei COBOL II und C kommentiert. Daran ist auch deren Anteil am Code sofort erkennbar – und er ist relativ niedrig. Ziemlich viel Code war aus formalen Gründen erforderlich, oder musste hinzugefügt werden, um die Gegebenheiten der Java-Laufzeitumgebung zu berücksichtigen. Andererseits war der Aufwand zum Formatieren der Ausgabeliste bei C höher – und weniger elegant – als bei Java; hier punktet COBOL mit seinen unschlagbar praktischen Formatierungsfähigkeiten.
- War es schwieriger als bei COBOL II und C, die Operationen und Kontrollen zu implementieren?

  Das traf zu. Die JSP-Methode macht es ziemlich einfach, einen gegebenen Entwurf in ein algorithmisches Programm umzusetzen. Eine objektorientierte Implementierung erfordert, die Kapselung der Instanzvariablen in ihren Objekten jederzeit im Blick zu behalten und gegebenenfalls zusätzliche Methoden für einfachste Operationen zu implementieren, weil diese Variablen nicht global adressiert werden können. Für algorithmisch trainierte Programmierer(innen) ist es auch nicht immer evident, ob eine Methode auf einem Objekt operieren, oder ob man ihr stattdessen ein Objekt-Handle als Parameter übergeben sollte. Das potentielle Kaskadieren von objektbezogenen Methodenaufrufen, zum Beispiel bei String-Operationen, ist zwar extrem praktisch, gegenüber C aber auch ungewohnt. COBOL-Programmierer(innen) sind in diesem Punkt sowieso leidgeprüft, weil COBOL II kein wirklich konkurrenzfähiges Unterprogrammkonzept besitzt.
- Wie ist das Ergebnis aus der Sicht der Wartungsprogrammierung unter der Annahme zu beurteilen, dass die Programmstruktur nachträglich geändert werden muss?

  Ohne Disziplin wird das nichts. Die durch die Implementierung geschaffene Codestruktur und die Programmstruktur sind nicht besonders ähnlich, außer in den Process Control-Klassen. Ein solch tiefgehender Eingriff erfordert daher, dass Wartungsprogrammierer(innen) sich zunächst mit dem Aufbau des Programms befassen, bis klar ist, wie der bisherige Entwurf in Code transformiert wurde. Erst dann kann mit Zuversicht damit begonnen werden, die Konsequenzen der Programmstruktur-Änderungen zu lokalisieren und diese umzusetzen. Das setzt eine gute Dokumentation vorher und nachher voraus.