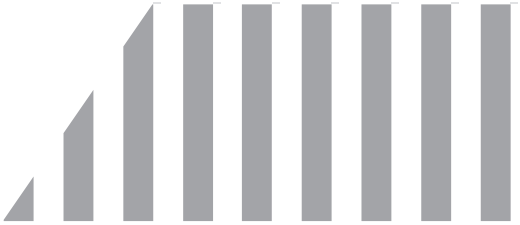


MODERNE STRUKTURIERTE PROGRAMMIERUNG

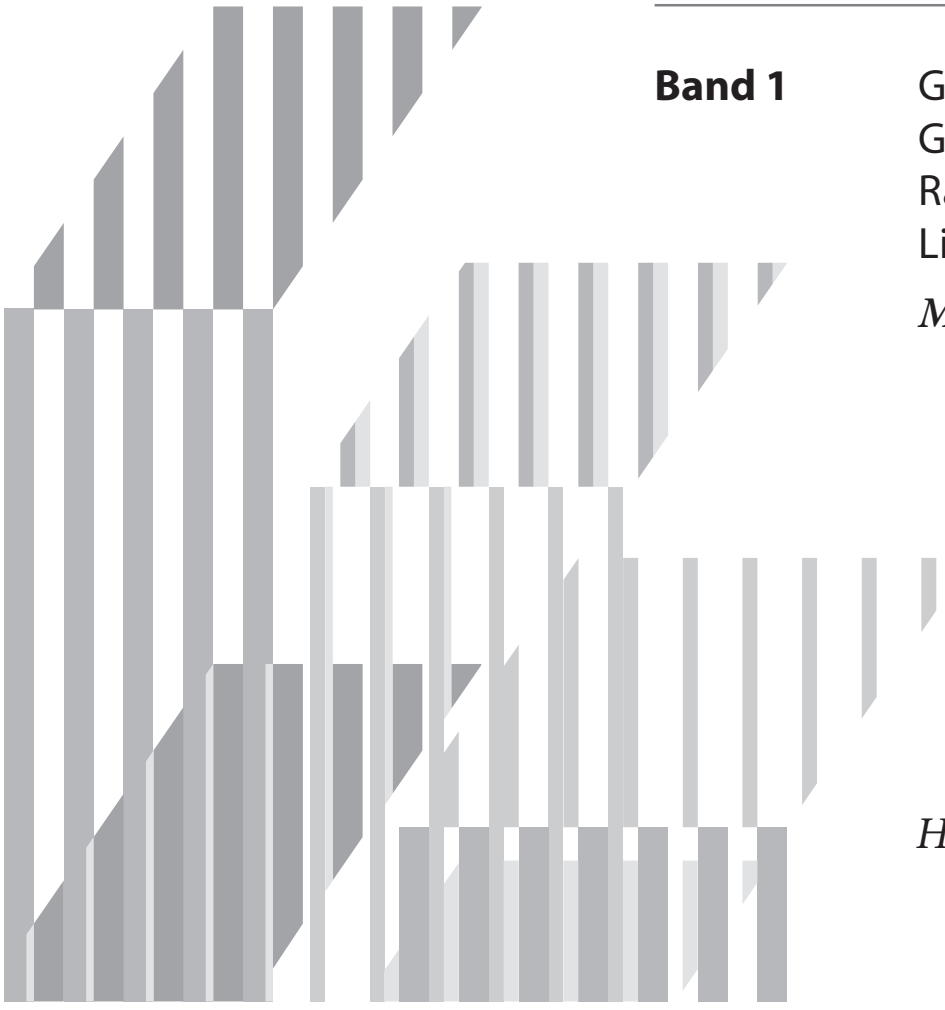
Objektorientiert und algorithmisch
ENTWERFEN | IMPLEMENTIEREN | TESTEN

2. Auflage

Band 1

Grundlagen
Gruppen
Ranggruppen
Link-Listen

Methode



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 „18.1 Literaturverzeichnis“ auf Seite 541 f
Rechtliches	siehe Kapitel „18.2 Rechtliche Hinweise“ auf Seite 543 ff
ISBN des Hardcover-Buchs	978-3-6963-1273-2

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 zur ersten Ausgabe

Die Informationstechnologie (IT) nimmt in den modernen Industriegesellschaften längst eine strategisch wichtige Position ein. Vom Erfolg oder Mißerfolg großer Projekte hängt seither auch ab, ob Firmen überleben und sich weiterentwickeln, oder ob sie scheitern. Regierungen, Firmen, Versorgungseinrichtungen, Verwaltungen, Kultursparten und viele weitere gesellschaftlich relevante Bereiche arbeiten mit Rechnern und sind ohne sie oftmals schon nicht mehr denkbar. Mit diesem Siegeszug der IT geht aber leider auch ein zunehmendes Störgeräusch einher, das durch scheiternde oder unerwartet teure Projekte verursacht wird. Je größer und komplexer die Projekte sind, um so eher geraten sie in diese Gefahr. Vieles, wenn nicht alles, hängt davon ab, welche Qualität die Systementwürfe besitzen und wie sie umgesetzt werden.

Anders als auf vielen anderen Gebieten, in denen Wasserfall-Projektmodelle für die Planung ausreichen, zum Beispiel beim Bauen, gibt es in der IT eine gegenseitige Abhängigkeit von Systementwurf und Realisierung. Auf den ersten Blick trivial, auf den zweiten aber durchaus keineswegs evident ist, dass nur das geplant werden kann (oder sollte), was auch realisierbar ist. Das Spektrum möglicher Implementierungen wirkt auf den Systementwurf zurück, der oftmals mehrfach verändert werden muss, bis die Ideen und die Technologie zusammenpassen. Das kommt daher, dass jede Neuentwicklung ein technologisches Potential besitzt, das öfters nicht sofort voll verstanden und konzeptionell genutzt wird. Dies unterscheidet die IT grundlegend von anderen Industrien, die ihre Planungen auf vorhandenen oder zielgerecht zu entwerfenden Produkten aufbauen.

Natürlich wird auch in der IT das Rad nicht jedes Mal neu erfunden. Eine Fülle von Technologien wurde geschaffen, um Anforderungen abzudecken, die in einer Vielzahl von Anwendungssystemen immer wieder auftreten. Ein Beispiel dafür sind die Datenbanksysteme, vor allem deren relationale Varianten, die es von relativ einfachen bis hin zu hochkomplexen Anwendungssystemen ermöglichen, die Datenhaltung zuverlässig und leistungsfähig auf elegante Weise zu organisieren. Dennoch steht und fällt bei jeder Neuentwicklung alles mit der Qualität des Systementwurfs *und* der Programmierung, denn letztere muss ersteren bestmöglich umsetzen. Leider steht es genau damit allzu häufig nicht zum Besten. Viel zu oft ist die Softwarequalität derart unterirdisch, dass auch der beste Systementwurf nichts nützt. Umgekehrt gilt jedoch für schlechte Entwürfe, dass wiederum die bestmögliche Programmierung die davon betroffenen Projekte nicht retten kann.

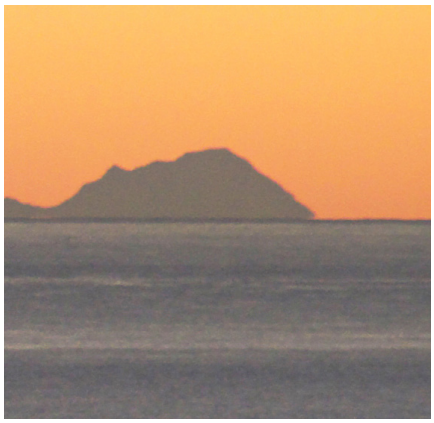
Diese Beobachtung hat seit Jahrzehnten die unterschiedlichsten Ansätze zur Problemlösung hervorgebracht. Davon haben sich etliche als extrem nützlich erwiesen. Vor allem jene Methoden, die sich auf die korrekte Umsetzung des Systementwurfs fokussieren, versprechen zu Recht, die allgemeine Malaise nicht nur irgendwie in den Griff zu bekommen, sondern tatsächlich den Weg in eine bessere Zukunft zu weisen. Insbesondere das **Jackson Structured Programming (JSP)** hat seit seiner Erfindung durch Michael Anthony Jackson am Anfang der siebziger Jahre des 20. Jahrhunderts zahlreiche überzeugte Anhänger gewonnen, die es immer dann einsetzen, wenn es sinnvoll ist – und das ist weit öfter der Fall, als dies auf den ersten Blick scheinen mag.

Viele von uns waren da noch gar nicht geboren, und die IT hat sich seitdem in einem historisch einmaligen – alle Prophezeiungen weit übertreffenden – Siegeslauf in damals unvorstellbarer Weise zu einem zentralen Akteur moderner Gesellschaften entwickelt. Es wäre daher nicht angemessen, das JSP in seiner Ursprungsform zu konservieren und wie den Satz des Pythagoras zur Unsterblichkeit zu erheben. Vielmehr wird es nur dann richtig gewürdigt, wenn man die ihm anhaftende Patina entfernt und es modernisiert in neuem Glanz erstrahlt. Diesem Ziel dient die **Moderne Strukturierte Programmierung (MSP)**, deren Band 1 Sie, geschätzte Leserinnen und Leser, jetzt in Händen halten. Die MSP baut auf dem JSP auf und wahrt getreulich dessen grundlegende, geniale Ideen, aber sie transzendiert es auch, um über fünfzig IT-Jahre hinweg eine Brücke in die Gegenwart zu schlagen. Lassen Sie sich hier nach und nach in JSP/MSP einführen und lernen Sie dabei, Ihr neues Wissen in Ihre Berufspraxis zu transferieren. Dabei wünsche ich, der Autor, Ihnen viel Erfolg!

Lemgo, den 2.5.2023

Übersichtsverzeichnis Band 1

Inhaltsverzeichnis Band 1



01/2012 Blick auf Porto Maurizio (Imperia, Ligurien) und den Golf von Genua vor den Apuanischen Alpen

Einführung

Die Fotografie als Kunst des Sichtbaren und das Programmieren als Kunst des Unsichtbaren haben im digitalen Zeitalter nur auf den ersten Blick nichts gemeinsam. Was aber hat die frühmorgendliche Aufnahme auf der linken Seite mit dem Thema dieser Buchreihe, der Modernen Strukturierten Programmierung (MSP), zu tun – abgesehen von der beliebten Metapher vom neuen Tag, der eine neue Zeit symbolisiert?

- *Die Freude über die Schönheit unseres Planeten ist nicht mehr naiv, seit wir von der Kugelgestalt der Erde wissen und berechnen können, was wir auf der anderen Seite des Golfs von Genua in circa 180 Kilometern Entfernung sehen. Wissenschaft durchdringt unsere Wahrnehmung, also längst auch die anfängliche intuitive Auffassung des Programmierens als kreativer Akt, bei dem es hauptsächlich auf Genialität ankommt.*
- *Ein durchgehend digitaler Prozess führt von diesem Foto dahin, dass Sie, geschätzte Leserinnen und Leser, jetzt diesen Band in den Händen halten. Das hätten wir uns allesamt in früheren Jahren nicht vorstellen können, als noch auf Filmen analog fotografiert wurde und Bücher im Bleisatz per Offsetdruckverfahren entstanden. Dieser epochale Umbruch ist noch ganz nahe, vor allem wenn man bedenkt, dass das analoge Fotografieren in der ersten Hälfte des 19. Jahrhunderts entwickelt wurde und bis in unser Jahrtausend dominierte.*
- *Erst die Entwicklung leistungsfähiger digitaler Kameras, schneller Computer, sowie die digitalisierte Medienproduktion machten diesen Umbruch möglich. Die dafür erforderlichen optischen, elektronischen und mechanischen Techniken wären jedoch ohne die Fortschritte der angewandten Informatik nutzlos, auf denen auch die – für diese Buchreihe verwendeten – komplexen Programme zur Bild- und Textbearbeitung beruhen. Erst das hochdisziplinierte Zusammenwirken der Hardware mit der sie steuernden Software erbringt zuverlässig die gewünschten Ergebnisse.*
- *Die Aufnahme wurde im kameraspezifischen RAW-Format gemacht, das man sich als eine pixelgenaue Aufzeichnung der Grundfarben R(ot) - G(rün) - B(lau) und der jeweiligen Helligkeiten vorstellen kann. Ein Dienstprogramm konvertierte diese Daten in das JPEG-Format, bevor das Bild mit Photoshop® bearbeitet und dann in das Manuskript eingebunden wurde. Alle diese Verarbeitungsschritte erfordern absolute Präzision, die letztlich nur durch die strukturierte Programmierung gewährleistet werden kann.*

Zuverlässigkeit ist das entscheidende Schlüsselwort, auf dem die MSP-Buchreihe aufbaut. Lange genug war der Prozess der Softwareentwicklung ein mühsamer und fehleranfälliger Vorgang, dessen bestmögliche Bewältigung allenfalls genialen Programmierer(inne)n zugetraut wurde. Alle anderen, also auch ich, der Autor, schrieben Programme, die meistens gut funktionierten, aber halt nicht immer. Der Software-Fehlerteufel plagt unsere Branche zwar nach wie vor, aber dank der Entwicklung solider, verständlicher Methoden sind sichere und hochkomplexe Systeme möglich, ohne dass es dafür Genies bräuchte.

Die nachfolgend vorgestellte Strukturierte Programmierung ist eines der Fundamente, auf denen diese Erfolge stehen. Jedefrau und jedermann, die oder der Software herstellt, sollte sie kennen und anwenden, wo immer das sinnvoll ist. Es handelt sich hier um Basiswissen der Informatik, dessen Nutzen auch noch nach Jahrzehnten immer wieder aufs Neue bestätigt wird. Dieses Wissen droht jedoch in Vergessenheit zu geraten, wird es nicht immer mal wieder auf den Stand der Zeit gebracht. Das geschieht in diesem Band und den ihm folgenden. Die MSP-Buchreihe ist für das Selbststudium konzipiert, damit niemand für das Erlernen der strukturierten Methode auf teure Kurse angewiesen ist. Daher wird alles ausführlich erklärt, und das, zusammen mit dem Anspruch, auch komplexe Beispiele zu zeigen, erklärt den erheblichen Umfang, der sich zwangsläufig auch auf die Länge dieser Einführung auswirkt. Die Buchreihe besteht daher aus drei Staffeln mit Bänden, die paarweise zusammengehören, nämlich je ein Methoden-Band und ein Praxis-Band mit den zugehörigen Programmbeispielen.

Wohl jede Autorin und jeder Autor kennt die Selbstzweifel, wenn das erste eigene Buch sich der Fertigstellung nähert und sein künftiges Publikum ins Blickfeld rückt: Wird es überhaupt Leser(innen) finden? Wie werden die Kritiken lauten? War die viele Arbeit dafür sinnvoll aufgewendet? Diese Fragen beschäftigten mich um so mehr, weil ich bis heute über ein Jahrzehnt lang an diesem Buch und den fünf ihm folgenden Bänden arbeitete (und noch arbeite). Meine Freunde und Bekannten, die von dem Buchprojekt wissen, sowie meine Verwandschaft dürften mich wahlweise als verrückt, verbohrt, stur, einseitig, ungesellig, weltfremd und manches mehr angesehen haben. Zugegeben, das stimmt alles. Anders ist aber ein so großes Kaliber nicht zu bewältigen. Was mit einem konservativen Ansatz begann, nämlich die unschätzbar wertvolle Methode von Michael Anthony Jackson[1], vor dem Vergessenwerden zu bewahren, hat sich als eine Reise in Gefilde entpuppt, von denen selbst ich – auch nach Jahrzehnten der Arbeit mit seiner Methode – keinerlei Ahnung hatte.

Um so lange an einem Projekt zu arbeiten, ist ein starker und stetiger Antrieb unerlässlich. Dieser speist sich zum einen aus etwas, das ich nicht anders als Dankbarkeit nennen kann – Dank an M. A. Jackson, dessen Ideen einen großen Teil meines Berufslebens geprägt haben. Zum anderen denke ich an professionelle Programmierer(innen) und Informatiker(innen), sowie an Anfänger oder Umsteiger aus anderen Berufsfeldern, für die seine Methode sich als ähnlich wertvoll erweisen kann, die aber noch nichts davon wissen. Da ich ein technischer Autor bin, also kein Romancier, der sich in andere Personen versetzen kann, habe ich das geschrieben, was ich selbst gern gelesen hätte – in der Hoffnung, damit möglichst Viele zu erreichen.

Warum heisst die Buchreihe „Moderne Strukturierte Programmierung"? Für das erste Wort gibt es ein historisches Vorbild, und das stammt von Edward Yourdon[2] – ebenfalls ein Autor, dem ich viel verdanke. Er schrieb das 1989 erschienene Buch „Modern Structured Analysis" (MSA) [1][3], in dessen Vorwort er übrigens ebenfalls seine Selbstzweifel bekundete. Völlig zu Unrecht, meine ich, denn seine Idee des Event Partitioning, also der Strukturierung von Anwendungssystemen als Ensemble von Prozessen, die auf Ereignisse reagieren, befreite die Systemanalytiker vom Zwang der hierarchischen Zerlegung à la SADT[4], der in den späten 1970er-Jahren meinungsführend war. Yourdons MSA hat sich in der Praxis überzeugend bewährt. (Am Rande: „Modern" oder „Neu" lauten seit der Antike die Schlüsselworte ewiger Jugend, was bei technischen Büchern allerdings als kokett erscheint, weil sie doch früher oder später einer Überarbeitung bedürfen).

Das von M. A. Jackson entwickelte „Jackson Structured Programming" (JSP) erfreute sich ab der Mitte der 1970er Jahre zumindest ein Jahrzehnt lang großer Beliebtheit, und es besitzt auch heute noch eine treue Anhängerschaft. Das zeigt sich unter anderem darin, dass das JSP-Buch von Dr. Klaus Kilberth 2001 in der 8. Auflage erschien [2] und mittlerweile als e-Book erhältlich ist.

Die hier vorgelegte Buchreihe über die „Moderne Strukturierte Programmierung" (MSP) intendiert die Aktualisierung und Weiterentwicklung des JSP, das 1974 durch Jacksons Buch „Principles of Program Design" [3] erstmals einer breiten Fachöffentlichkeit bekannt wurde. Es hatte einen solch durchschlagenden Erfolg, weil es erstmals einen sowohl intuitiv einleuchtenden als auch theoretisch wohlfundierten Weg von der Aufgabenstellung zum fertigen Programmentwurf wies. Jacksons grundlegende Idee der Synchronisierung hierarchisch-grafischer Datenstrukturen anhand sogenannter Korrespondenzen und die daraus folgende Ableitung der – im selben Stil aufgebauten – Programmstrukturen löste auf Anhieb viele Probleme, die auch langerprobte professionelle Anwendungsentwickler immer wieder in Bedrängnis gebracht hatten. Der Vater der strukturierten Programmierung ist aber nicht Jackson, sondern Edsger Wybe Dijkstra[5].

Publikationen über das JSP erschienen auch in Deutschland zunächst nur in englischer Sprache. Das Informatik-Kolleg der Gesellschaft für Mathematik und Datenverarbeitung[6] (GMD) in Darmstadt übersetzte und konsolidierte diese Texte, um das JSP in die dort stattfindende Ausbildung wissenschaftlich-technischer Assistent(inn)en zu integrieren. Es bot diesen Kursus aber auch den wissenschaftlichen Mitarbeitern der GMD an. Auf diesem Weg kam ich[7] als Mitglied der TR 440[8]-Systemgruppe mit dem JSP in Berührung – voller Misstrauen gegen diese angeblich revolutionäre Neuerung, die doch nur ein Aufguss bereits bekannter, untauglicher Ansätze zur strukturierten Entwicklung von Software zu sein schien. Nach dem Kursus waren diese Vorurteile allerdings komplett beseitigt und fortan gehörte das JSP zu meinem beruflichen Grundwissen.

Endnoten: siehe Seite 14

1.1 JSP-Literatur

Die im GMD-Informatik-Kolleg tätigen Autoren R. Legde und K. Truöl erarbeiteten 1977/78 eine umfangreiche JSP-Einführung mit dem Titel „Problem- und datenorientierter Entwurf strukturierter Programme“ [4]. Auf diesem äußerst nützlichen Dokument basiert die Moderne Strukturierte Programmierung in vielerlei Hinsicht. Erst 1979 erschien Jacksons Buch auf Deutsch mit dem Titel „Grundsätze des Programmentwurfs“ [5]. Diese Darstellung seiner Methode erforderte unter anderem, sich mit COBOL auszukennen, war also nicht so leicht wie [4] zu lesen und zu verstehen. Heutige Leser(innen) dürften sich darüber hinaus daran stören, dass es vorwiegend die Stapelverarbeitung thematisiert, zu der schon bald danach die Transaktionsverarbeitung und in neuerer Zeit die Web-Technologien hinzugetreten sind. Auch das Thema Datenbanken sucht man darin vergeblich – kein Wunder, steckten diese doch 1974 noch in den Kinderschuhen. Es wäre daher grob ungerecht, Jacksons bahnbrechende Gedanken aus der heutigen Perspektive zu beurteilen.

Dieselben heutigen Leser(innen) haben aber vermutlich schon eine Menge IT-Literatur konsumiert, die nach dem Prinzip arbeitet, alles in kleinste Lerneinheiten zu zerlegen und so den Weg zum Ziel möglichst eben zu gestalten. Jackson mutete seinen Leser(inne)n dagegen auch steile Anstiege und jähe Kurven zu, wenn ihm das für seine Argumentation dienlich erschien. Man muss sein Buch daher sehr aufmerksam lesen, um seine Gedankengänge zu verstehen und an seinen Einsichten teilzunehmen. Vielleicht liegt es daran, dass seine Absicht, die bis dahin üblichen Programmiertechniken als verkorkst zu entlarven und ein neues Fundament für die Programmentwicklung zu legen, nicht in dem Umfang zum Tragen kam, wie es für eine weite Verbreitung des JSP notwendig und sinnvoll gewesen wäre. Bis auf den heutigen Tag belegen zahllose Code-Fragmente, die man im Internet studieren kann, dass die Grundideen Jacksons leider weithin unbekannt geblieben sind.

Wer die Geduld aufbrachte, Jacksons Beispielfällen und seiner meist zwingenden Argumentation zu folgen, der wurde reich belohnt. Immerhin war – und ist – das JSP die einzige Software-Engineering-Methode, die es ermöglicht, auf einem plausiblen Weg von den Datenstrukturen zur Programmstruktur zu gelangen. Für schnelle Leser(innen) ist das Buch jedoch leider völlig ungeeignet. Insbesondere fehlt eine übersichtliche Zusammenfassung (die für die Mitte des Bands 5 dieser Buchreihe geplant ist). Man muss sich daher die relevanten Aspekte der Methode über viele Seiten verstreut zusammensuchen und kann nicht sicher sein, alles Wesentliche wirklich erfasst zu haben. Positiv ist dagegen, dass Jackson zahlreiche Beispiele anführte, um seine Ideen zu erklären. Das JSP ist ja eine auf Abstraktion basierende Modellierung, was die Gefahr in sich birgt, sich dem Zauber des Abstrakten hinzugeben und darüber die ganz realen Programmierprobleme und -pannen aus dem Blickfeld zu verlieren.

COBOL ist ein Akronym, das am Ende der 1950er Jahre aus der Bezeichnung „Common Business-Oriented Language“ gebildet wurde. Diese krude Programmiersprache war all denjenigen fremd, die – wie ich – aus der Welt von ALGOL, FORTRAN, BCPL und diverser Assemblersprachen kamen. Unglücklicherweise – für unsereinen – war dieses alte COBOL jedoch Jacksons wichtigstes Darstellungswerkzeug. Es lieferte ihm die speicherinternen Datendefinitionen und war die Grundlage für zahlreiche Codebeispiele. Erstaunlicherweise hat Jackson die von ihm entwickelte grafische Sprache, die Struktogramme, nicht so intensiv in den Vordergrund seiner Argumentation gestellt, wie sie es meines Erachtens verdient hätte. Zudem fehlt dem von ihm vorgeschlagenen Pseudocode namens Schematische Logik die erforderliche Eleganz – bei hohem Schreibaufwand. Somit hatte sein Buch aus meiner Sicht einige didaktische Mängel, was der Verbreitung des JSP ausserhalb der kommerziellen Programmierung vermutlich nicht förderlich gewesen ist. Die damaligen objektorientierten Sprachen wie zum Beispiel Simula und Smalltalk spielten in Jacksons Buch keine Rolle. Das JSP wurde vermutlich auch deshalb als eine rein algorithmische Entwurfsmethode aufgefasst, obwohl, genau genommen, das alte COBOL – heute COBOL I genannt – den damaligen algorithmischen Sprachen weit unterlegen war.

1.2 Wo stehen wir heute?

Seit Jacksons englischem Buch [3] sind fünfzig Jahre vergangen, in denen sich die IT weltweit zu einer der führenden Branchen entwickelt hat. Damit einher ging einerseits eine enorme Differenzierung der Plattformen und Anwendungsfelder, andererseits eine damals unvorstellbare Steigerung der Leistungsfähigkeit von Computersystemen. Jede(r) kann heute ein Vielfaches der Speicherkapazität und der Rechenleistung eines TR 440- oder IBM-System/360[9] in ihrem/seinem Smartphone mit sich herumtragen – allerdings ohne damit Hunderte Benutzer gleichzeitig online zu bedienen und parallel dazu aufwendige Stapelverarbeitungen auszuführen. Der größte Teil der Smartphone-Rechenleistung wird für dessen grafische Oberfläche benötigt.

Währenddessen fand ein beeindruckender Siegeszug der Objektorientierung statt, dem der Großteil zahlreicher Innovationen zu verdanken ist, die für ihre Nutzer mittlerweile als unverzichtbare Errungenschaften gelten. Dieser Aufschwung war nur auf der Basis massiver methodischer Anstrengungen möglich, die unter den OO-Kürzeln OOA (Object-Oriented Analysis), OOD (Object-Oriented Design) und OOP (Object-Oriented Programming) zusammengefasst werden können. Ohne diese theoretische Basis konnten die heutigen komplexen OO-Systeme nicht entwickelt werden.

Die alte algorithmische Programmierung bildet zwar immer noch das Rückgrat der datenbank- und transaktionsbasierten Hochleistungssysteme, aber diese gehören überwiegend zum Gebiet der Mainframes, deren Verbreitung auf Großorganisationen beschränkt ist. In diesem Sektor der IT ist bedauerlicherweise eine erhebliche Verwilderung der guten Sitten bei der Programmentwicklung und –wartung festzustellen, wie ich es in meiner eigenen Praxis oftmals erlebt habe. Aus den Auseinandersetzungen um die „einzig wahre" Softwaretechnologie in den 1970er Jahren ging nämlich keine einzige Methode als Sieger hervor, auch das JSP nicht. Zu divergent waren schon damals die Anforderungen an eine solche „Theorie von Allem", und zu wenig flächendeckend konnte sich die strukturierte Programmierung – in welcher Ausprägung auch immer – etablieren. Die Folgen: Wildwuchs, Zynismus, Methodenaversion, Bastlermentalität, bestenfalls formale – und institutionell erzwungene – Strukturierung mittels Programmgeneratoren.

In diesen Dekaden versanken auch die alten Namen und Abkürzungen: Michael Jackson, das war doch ein berühmt-berüchtigter Pop-Sänger, und JSP bedeutet JavaServer Pages (alt) beziehungsweise Jakarta Server Pages® (neu), nicht wahr? Interessiert sich überhaupt noch jemand ausser einer kleinen Gruppe nostalgischer Alt-Programmierer(innen) für die strukturierte Programmierung? Ist sie nicht rettungslos dem algorithmischen Denken verpflichtet, das angesichts der Erfolge der Objektorientierung als eine verflossene Entwicklungsphase gilt, geradezu als Zeit der Programmier-Dinosaurier? Jede(r) kennt doch diese Tiere, speziell die Pflanzenfresser: massiger Körper, kleiner Kopf, folglich wenig Intelligenz. Diese beleidigende Parallele erscheint zudem als wohlverdient angesichts der Fehlschläge, die wegen der allzu leicht entstehenden Undichtigkeiten in algorithmisch aufgebauten Programmen vielerorts aufgetreten sind. Die Objektorientierung verspricht dagegen, vor allem durch die obligate Kapselung von Daten und den ihnen zugeordneten Methoden in Klassen beziehungsweise Objekten, weit besser strukturierten Code zu ermöglichen. Das ist eine gute Nachricht.

Die schlechte Nachricht lautet: Auch mit den besten objektorientierten Sprachen und Methoden ist es anscheinend immer noch eine Kunst, ein Programm so zu schreiben, dass es gut gestaltet ist und seine Aufgabe einwandfrei löst, also keinen Unfug macht und auch nicht abstürzt. Kunst kann zwar imitiert, aber nicht gelehrt werden. Jemand ist ein – werdender – Könner, oder ist es nicht. Kunst ist vor allem ein Prozess, dessen Ergebnis weder vorhersagbar noch personenunabhängig ist. Dieses Paradigma, dem der Typus des freien Programmierkünstlers entspricht, ist das exakte Gegenteil des Ingenieursgedankens, dass Programme wie andere technische Erzeugnisse konstruiert werden können. Konstruktion ist keine Kunst. Sie ist angewandte Wissenschaft.

1.3 Programme mit MSP konstruieren

Was ist also modern an der Modernen Strukturierten Programmierung? Sie baut auf dem Jackson Structured Programming auf und verfolgt dieselbe Absicht: Programme sollen nicht erfunden, sondern konstruiert werden. Hierfür integriert die MSP-Buchreihe die folgenden Aspekte:

- Durch ein Facelifting des JSP und die intensive Verwendung von Struktogrammen strebt die MSP an, diese Software-Engineering-Methode auch ausserhalb der kommerziellen Programmierung zu etablieren. Anstelle von COBOL oder Jacksons Schematischer Logik dient ein moderner Pseudocode als Beschreibungssprache, der auch eine flexible Syntax für Datendefinitionen umfasst. Neu ist auch die Nutzung der Struktogramme für die Durchlaufanalyse vor der Implementierung, mit der Entwurfsfehler schon frühzeitig aufgedeckt werden können – ein veritabler Schreibtischtest.
- An vielen, teilweise ziemlich komplexen Beispielen zeigt die MSP, dass auf der Basis von M. A. Jacksons Struktogrammen mit Operationen und Kontrollen sowohl – wie bisher – algorithmische als auch – neuerdings – objektorientierte Implementierungen hergeleitet werden können. Für letztere ist ein zusätzlicher OO-Entwurfsschritt erforderlich, dessen Verlauf durch die vorangehende JSP-Modellierung gesteuert und damit weithin vorhersagbar wird. Das eröffnet die Möglichkeit, existierende algorithmische Programme zunächst mit Struktogrammen zu modellieren und dann in objektorientierte Programme zu transformieren – eine strategisch wichtige Basis für die Migration von Altsystemen auf neue Plattformen. Zugleich werden Barrieren abgebaut, die bislang manche Entwickler(innen) davon abhielten, die jeweils beste Lösung für ein Programmierproblem sowohl auf der algorithmischen als auch auf der objektorientierten Seite zu suchen. Noch wichtiger ist mir allerdings, dass werdende professionelle Anwendungsentwickler zahlreiche Anregungen erhalten, wie sie das JSP für ihre eigene Praxis zum Tragen bringen und dabei ein hohes qualitatives Niveau erreichen können.
- Die Entwicklung der modernen, GO-TO-freien Programmiersprachen brachte es mit sich, dass es schwierig oder sogar unmöglich zu werden schien, damit Jacksons Problemlösungsverfahren „Backtracking“ und „Programminversion“ anzuwenden. Die MSP zeigt einen Konstruktionspfad auf, der es ermöglicht, in *jeder* modernen Programmiersprache diese beiden Verfahren anzuwenden.
- Viele Software-Engineering-Methoden scheiterten daran, dass sie nicht miteinander kompatibel waren und zudem mit dogmatischer Härte gelehrt wurden. Zwanghaftes Entwerfen, das starren Mustern verpflichtet ist, schadet jedoch der Qualität der Ergebnisse und schöpft das Potential des JSP nicht aus. Die MSP bettet Jacksons Methode daher in ein Umfeld ein, das von vielfältigen, konkurrierenden Ansätzen durchdrungen ist. Dabei geht es darum, einerseits den Kern der Methode zu verdeutlichen, andererseits das JSP so flexibel wie möglich anzuwenden. Starre Begrifflichkeiten und Regeln, die zu unnötigen Umwegen oder künstlichen Anpassungen an eine verabsolutierte Methodik führen, werden daher aufgegeben. Ein Beispiel dafür ist die Einbindung der Automatentheorie dort, wo eine JSP-konforme Ableitung des Codes – scheinbar oder tatsächlich – als krampfhafte Übung erscheint.
- Die Aktualisierung des JSP und die neuen methodischen Elemente, welche die MSP einbringt, bedeuten auch, neue grafische Elemente vorzuschlagen, die für mehr Übersichtlichkeit sorgen, Irrtümer vermeiden helfen, oder die Qualität der Dokumentation erhöhen. Durchgehend liegt der Akzent darauf, die bei jedem Entwurf anfangs noch weichen Beschreibungselemente zu festen, präzisen Definitionen weiterzuentwickeln, die den Implementierungen zugrunde gelegt werden. Das heißt, sukzessive den Interpretationsspielraum der Programmierer(innen) und damit die deutungsbedingten Irrtumsmöglichkeiten einzuschränken. In diesem Kontext verabschiedet sich die MSP auch von fragwürdigen Thesen aus der JSP-Entstehungsgeschichte, die es manchen Modellierern möglich machten, die Last ihrer Fehlentscheidungen oder ihrer Bequemlichkeit den Programmierer(inne)n aufzubürden.

- Die teilweise extreme Arbeitsteilung in der IT führt vor allem bei Neulingen zu einer verzerrten Wahrnehmung des Entwurfsprozesses. Die MSP-Buchreihe nimmt sich daher die Freiheit, anhand sorgfältig ausgewählter Beispiele den gesamten Weg von der Idee bis hin zu Implementierung und Test darzustellen. Die strukturierte Vorgehensweise wird dabei in mehreren Facetten gezeigt: als Entwicklung aus einem Guss (der traditionelle Weg), als inkrementelle Vorgehensweise, als schrittweise Weiterentwicklung, als Erstellung von Service-Modulen für nicht-JSP-basierten Code, als Revision unbefriedigender Lösungen, als Korrektur eines Irrwegs oder auch als Migration auf eine andere Programmierplattform.
- Die meisten Beispiele wurden in C oder Java® programmiert und getestet, nur eines in COBOL II. In vielen Fällen wird zunächst die algorithmische Realisierung gezeigt, und dann der Übergang zu einer objektorientierten Lösung im Detail vollzogen. Teilweise blieb es bei einer der beiden Alternativen, um den Umfang der Buchreihe nicht unnötig aufzublähen, oder um spezielle Möglichkeiten zu nutzen. Java-Programme dienen zum Beispiel dazu, Mails aufzubauen und zu versenden, oder die Verzeichnisstrukturen des File-Systems auszuwerten. Der Umfang der Beispielprogramme, die zudem nicht jeden interessieren dürften, machte die Auslagerung ihres Codes, der zugehörigen Beschreibungen und des größten Teils der Testdokumentation in den den jeweils zweiten Band der drei Buchstaffeln erforderlich. Diese Aufteilung ermöglicht es, die jeweils zusammengehörenden Bände nebeneinanderzulegen, um die Übertragung des Entwurfs in den Code zu studieren.
- Das JSP beruhte technisch auf sehr einfachen Datenorganisationen, vorwiegend auf sequentiellen Dateien. Die Entwicklung von Datenbanksystemen begann erst damals und es ist bislang kein Ende abzusehen. Dabei entstand eine enorme Vielfalt von technischen Lösungen, die bei großen und mittleren Rechnern vor allem auf die Interaktion mit Transaktionssystemen hin ausgelegt wurden. Dort treten Konkurrenz- und Konsistenzprobleme auf, die für die verlässliche Funktionalität der Anwendungssysteme zwingend gelöst werden mussten. Dagegen spielen konkurrierende Zugriffe auf Desktop-Systemen und erst recht auf portablen Rechnern keine wesentliche Rolle. Auf diesen Maschinen haben sich hauptsächlich relationale Datenbanksysteme durchgesetzt, während auf IBM®-Großrechnern neben dem relationalen Db2® auch das ältere IMS DB und dessen quasi-relationaler Konkurrent ADABAS® im produktiven Einsatz sind.

  Der datenorientierte Entwurf nach Jackson muss natürlich je nach Datenbanksystem adaptiert werden, um dessen Zugriffsformen aufzugreifen, die Aufteilung der bislang kontinuierlichen Verarbeitung in Segmente – Transaktions- oder COMMIT-Intervalle – zu berücksichtigen, und beispielsweise für Restarts die erforderlichen Vorkehrungen zu treffen. Einerseits ist es offensichtlich unmöglich, in dieser Buchreihe das gesamte Spektrum der modernen Datenbanksysteme auf seine JSP-Relevanz hin zu untersuchen und die Konsequenzen für die Modellierung der Daten- und Programmstrukturen zu diskutieren. Andererseits wurde wegen der hohen Bedeutung, die insbesondere die RDBMS – Relational Database Management Systems – für die heutige Anwendungsentwicklung besitzen, diese Thematik in der zweiten und dritten Staffel dieser Buchreihe aufgegriffen.

Ihr Eindruck von alledem als Leser(in) mag sein, dass Sie sich durchaus positiv angesprochen, aber vom Umfang der MSP-Buchreihe abgeschreckt fühlen. In der Tat muss ich zugeben, dass mehrere Beispiele sich als weit raumfordernder erwiesen haben, als es ursprünglich geplant und abzusehen war. Der Spitzenreiter ist sicherlich der internationale Schul-Sportwettbewerb mit dem abschließendem Versand der Ergebnisse per E-Mail. Gerade diese Fallstudie belegt jedoch nicht nur eindrucksvoll die Ausweitung des JSP auf das objektorientierte Gebiet, sondern ermöglicht auch zahlreiche Einsichten in die damit einhergehende Anpassung an die durch Java definierte Basismaschine – ein realitätsnahes Beispiel für eine Migration auf eine neue Plattform.

Alles in allem deckt die MSP-Buchreihe nicht nur den gesamten Umfang des JSP ab, wie ihn Michael A. Jackson definiert hat, sondern sie führt auch vielfältige Modifikationen und Ergänzungen ein, die zusammengenommen das Attribut „modern" rechtfertigen – von den umfangreichen Implementierungs- und Testbeispielen mal völlig abgesehen. Das heißt jedoch auch, dass viele Programmierthemen, welche die IT seit

damals entdeckt hat, nicht berücksichtigt werden konnten. Es wären einfach zu viele, um sie alle zu würdigen. Ausserdem hat es sich in Jahrzehnten meiner JSP-Anwendung auf sehr verschiedenen Gebieten der Informatik gezeigt, dass das JSP deshalb keiner Erweiterung bedurfte – von den Datenbank-Aspekten mal abgesehen.

1.4 ... und die Folgen

Jede Medizin hat Risiken und Nebenwirkungen. Das gilt auch für „Medikamente gegen schlechte Software", also für alle Methoden, die darauf abzielen, den Entwurfsprozess an objektiven Kriterien statt an nebulöser Könnerschaft auszurichten. Das primäre Risiko besteht bei JSP/MSP darin, damit Probleme lösen zu wollen, für die der datenorientierte Ansatz nicht geeignet ist, zum Beispiel in der Robotik – aber das ist im Allgemeinen leicht zu erkennen. Zwar erweitert die MSP den Lösungsraum um die Modellierung von Programmstrukturen, die nicht aus Datenstrukturen abgeleitet werden (können), aber damit geht die Führungswirkung der korrespondenzgesteuerten Verschmelzung von Daten-Struktogrammen verloren. Dennoch sind solche nicht-konservativen Entwürfe der ad-hoc-Programmierung meines Erachtens durchaus vorzuziehen, weil sie es ermöglichen, die so entstandenen Programmstrukturen auf einem guten qualitativen Niveau zu untersuchen – auch und gerade um Fehler aufzudecken. Ob dieser Nutzen es rechtfertigt, den Aufwand der JSP/MSP-Modellierung zu treiben, kann nur von Fall zu Fall entschieden werden.

Damit eng verwandt ist das Risiko des Denkens in starren Kategorien. Ist man erst einmal an eine Entwurfsmethode gewöhnt, liegt es nahe, neue Probleme nur noch durch die Brille dieser Methode zu betrachten. Schlimmstenfalls wird daraus ein Prokrustesbett[10]. In diese Falle sind viele Software-Engineering-Ansätze getreten, die ihre Anhänger auf einen strikten Regelgehorsam verpflichteten und neben sich keine Konkurrenz duldeten.

Wenn JSP/MSP die richtige Methode für Ihr Programmierproblem ist, dann gibt es ein soziales Risiko: Sie könnten zum Aussenseiter werden. Solange Ihr Management bis hinab zum Projektleiter entweder andere Methoden oder gar keine favorisiert, wobei Letzteres im Bereich der algorithmischen Programmierung leider die Regel ist, fallen Sie möglicherweise als Abweichler auf. Anstatt nur die geforderten Dokumente und danach direkt den Programmcode zu produzieren, verschwenden Sie aus der Sicht Anderer, auch Ihrer Kollegen, Ihre Zeit mit Struktogrammen. Der Erfolg wird Ihnen zwar später recht geben, aber bis dahin müssen Sie mit dem Argwohn derer leben, die Ihr Vorgehen nicht verstehen oder Ihre Präferenz nicht teilen – und des Öfteren ausserdem nichts dazulernen wollen, sondern mit dem Erreichten zufrieden sind.

Auch die Nebenwirkungen sind nicht harmlos: Sie verändern sich. Anstatt sich freudig auf die Produktion von Code zu stürzen und Ihr Ego durch großartige Konstruktionen zu bestätigen, die womöglich niemand sonst richtig zu würdigen weiß, gehen Sie einen zunächst als mühsamer erscheinenden Weg. Sie zeichnen Datenstrukturen und ermitteln Korrespondenzen, bauen Programmstrukturen auf, definieren die Auswahl- und Schleifenkontrollen, erstellen die Liste der Operationen und ordnen diese den richtigen Stellen im Programm-Struktogramm zu. Wenn Sie ein OO-Programm entwerfen, schließen Sie den dafür erforderlichen Entwicklungsschritt an und runden Ihre Arbeit mit einem Klassendiagramm ab – möglicherweise, bevor Sie eine einzige Zeile Code geschrieben haben. Es ist auch ein exploratives Vorgehen denkbar, bei dem Sie die Entwurfsschritte iterativ durchlaufen und zwischendrin immer wieder programmieren, gemäß dem Motto von Grady Booch[11] „Design a little, program a little" in seinem Buch „Object-Oriented Design with Applications" [6].

Auf jeden Fall verliert das Programmieren dadurch an Reiz. Der eigentliche Entwurf findet im Vorfeld statt und das Programmieren dient „nur noch" dazu, ihn umzusetzen beziehungsweise die Umsetzbarkeit Ihrer Ideen zu prüfen. Der Suchtfaktor des Erschaffens neuer Welten als Programmierer(in) fällt dadurch – leider – weitgehend weg. Zugleich verändert das disziplinierte, methodische Vorgehen Ihre Arbeitsweise insgesamt. Wo vorher einige Skizzen und möglichst unpräzise Vorgaben Ihnen freie Hand ließen, werden Sie nun bessere Analyseergebnisse erwarten oder gegebenenfalls vorhandene Lücken selbst zu schließen versuchen, damit

Ihr Entwurf überhaupt fertiggestellt werden kann. Sie werden kritischer und sind weniger bereit, analytische Schlampereien zu akzeptieren.

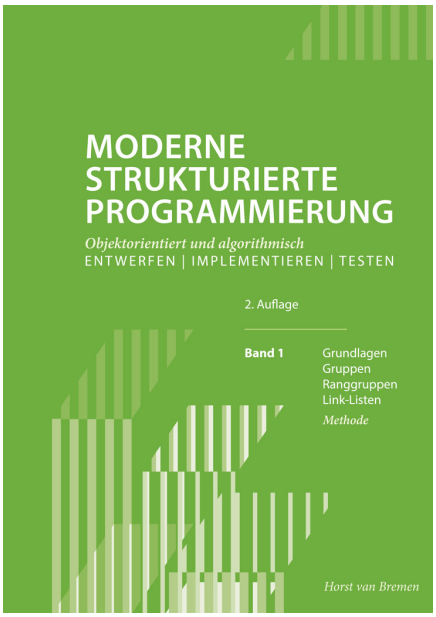
Rundweg positiv ist die Nebenwirkung, dass Sie weniger Angst verspüren werden. Schon während des Entwurfs, der Programmierung und dem Test sind Sie entspannter, weil es nun nicht mehr hauptsächlich auf Ihr Können als Entwickler(in) ankommt, sondern darauf, alles richtig modelliert zu haben. Anstatt sich zu sorgen, ob Sie wirklich alle denkbaren Datenkonstellationen geprüft haben und ob sämtliche Pfade in Ihrem Programm häufig genug korrekt durchlaufen wurden, gibt Ihnen JSP/MSP die Sicherheit, alle relevanten Aspekte beachtet zu haben. Das bislang Undenkbare tritt ein: Sie geben ein Programm in Produktion und es läuft dauerhaft fehlerfrei.

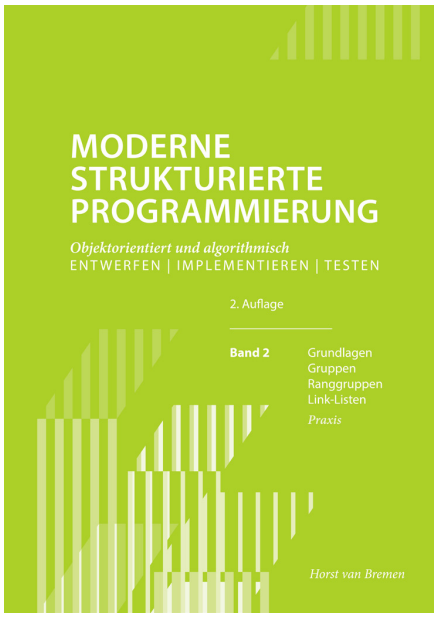
Auch in der adaptiven Wartung, also der Weiterentwicklung, entfalten sich die Vorteile des strukturierten Programmierens: Anstatt im Code nach denjenigen Stellen suchen zu müssen, auf welche die neuen Anforderungen sich auswirken, und dabei auf gar keinen Fall etwas zu übersehen, greifen Sie auf die Dokumentation zurück. Sie machen sich wieder mit den Entwurfsdokumenten vertraut, identifizieren dort die Auswirkungen dieser Anforderungen, erstellen neue Versionen der Struktogramme, sowie der begleitenden Listen von Kontrollen und Operationen und ordnen diese Änderungen den korrespondierenden Code-Partien zu. Nach der Umsetzung finden die Tests auf der Basis der bisherigen beziehungsweise der erweiterten Datenkonstellationen statt. Wenn Sie bei alledem das Richtige getan haben, wird die neue Programmversion ebenso fehlerfrei wie die alte arbeiten.

„Zero defects" ist das eigentliche Ziel aller Software-Entwicklungsmethoden. Ein fehlerfreies Programm enthält nicht nur den korrekten Code, der die Anforderungen umsetzt, sondern es ist auch gut gegliedert und – intern wie extern – dokumentiert. Mit JSP/MSP steht Ihnen nun eine mächtige Zero Defects-Methode zur Verfügung, die sich nicht absolut setzt, sondern sich mit anderen Ansätzen verträgt. Aufgrund meiner jahrzehntelangen positiven Erfahrungen mit dieser Methode, die mir oftmals meine berufliche Existenz gerettet hat, wünsche ich Ihnen viel Freude beim Lernen und noch mehr Erfolg bei der Anwendung in Ihrer Praxis.

1.5 Die Buchreihe

Wie schon erwähnt, besteht die Buchreihe aus drei Staffeln mit paarweisen Bänden[12]. Was ist darin enthalten?





Im Band 1 geht es um die Grundlagen der Methode und um (Rang-)Gruppenverarbeitungen, sowie um Link-Listen. Er beginnt mit einem sehr einfachen Beispiel und steigert sich bis zum automatischen E-Mail-Versand der Ergebnisse des dort entworfenen fiktiven Internationalen Schulsportwettbewerbs. Dabei kommen nahezu ausschließlich „traditionelle" Aspekte der Methode zur Anwendung, die allerdings zum Teil auf ungewohnte Weise dazu dienen, funktionstüchtige Programme sowohl algorithmisch als auch objektorientiert zu entwerfen. Der Übergang zwischen diesen beiden Formen wird ausführlich erklärt und demonstriert. Auch das Thema „Test" spielt darin eine große Rolle, und zwar vor *und* nach der Implementierung. Ergänzend wird eine Java-Standard-Zugriffsschicht für sequentielle Dateien entworfen.

Nur im Ringbuch-Format wurden die Bände 1 und 2 in die Schwierigkeitsniveaus A1 und A2 geteilt. Wer sich noch nie zuvor mit der Strukturierten Programmierung befasst hat, sollte sich mit dem Niveau A2, also den Ranggruppen- und Link-Listen-Verarbeitungen, Zeit lassen. Keine Bange! Es wird alles detailliert erklärt. Auf B1 und B2 (in den Bänden 3 und 4) folgen C1 und C2 (in den Bänden 5 und 6). Die Analogie zu den Sprachniveaus wurde gewählt, um zu verdeutlichen, dass der Weg der Buchreihe vom elementaren Einstieg für Anfänger bis zu einem hohen Maß an Beherrschung von Methode und Praxis für Profis führt.

Der Band 2 enthält alle Beispielprogramme, mit denen die Entwürfe im Band 1 realisiert wurden. Das erste und einfachste Beispiel wurde in COBOL II, C und Java implementiert, während die anderen Beispiele meist in C *und* in Java – als prototypische Exponenten der algorithmischen und der objektorientierten Welt – programmiert wurden. Nur das letzte Beispiel, der E-Mail-Versand, ist allein in Java geschrieben, weil ein C-Programm sehr umständlich wäre und keinen Erkenntnisgewinn böte.

Der Band 3 behandelt das Mischen in vielerlei Konstellationen. Oft stehen die zu mischenden Daten keineswegs passend sortiert zur Verfügung. Dann ist deutlich mehr Kreativität als im Fall synchron sortierter Daten erforderlich, der gleichwohl ebenso detailliert erklärt wird. Mit dem Mischen von relational organisierten Daten einschliesslich Restartfähigkeit wird der Schwierigkeitsgrad B1 erreicht. Er steigt auf B2 mit dem Abgleich von Bilddateien, die einerseits auf Memory Chips und andererseits auf Festplatten oder ähnlichem gespeichert und in Ordnern organisiert sind. Schließlich geht es darum, im Rahmen des Copy Management ganze Ordnerhierarchien gegeneinander abzugleichen, um Differenzen zu erkennen und sie für die zielgerichtete Backup-Aktualisierung zu verwenden. Die strukturierte Hilfsmethode der Programminversion wird erstmals verwendet, aber noch nicht detailliert untersucht.

Der Band 4 zeigt wiederum die zu den Entwürfen gehörenden Implementierungen. Die Programme im Band 4 sind in C und in Java geschrieben, wobei die Java-Programme das relationale Datenbanksystem MySQL nutzen, wo nötig.

Nun verbleiben noch drei Themenkomplexe, nämlich die bereits erwähnten Hilfsmethoden einerseits, und die spezielle Beziehung der Strukturierten Programmierung zur Datenbankprogrammierung andererseits.

Der Band 5-C1 geht die Themen Backtracking und Programminversion systematisch an, und er fasst die JSP/MSP-Methodik zusammen, bevor im Band 5-C2 in Top Down-Manier der Bezug zwischen der Systemanalyse und der relationalen Datenbankprogrammierung aus JSP/MSP-Sicht aufgebaut wird. Der Hardcover-Band 5 vereint C1 und C2. Anhand einer MySQL-Implementierung der Datenbestände wird der Entwurf von Programmen für den Internationalen Schulsportwettbewerb im relationalen Umfeld demonstriert, final sogar anhand einer restartfähigen Verarbeitung. Dieser Band ist definitiv nicht für Anfänger geeignet, wird aber für Fortgeschrittene und Profis – hoffentlich – von großem Nutzen sein.

Der Band 6(-C1/C2) versammelt alle Beispielprogramme, die aus den Entwürfen im Band 5(-C1/C2) entstanden. Sie sind in C *oder* in Java – also nicht in beiden Sprachen – geschrieben, wobei die Datenbankprogramme wie im Band 4-B1 in Java implementiert wurden. Der Grund hierfür ist, dass das C-API (Application Programming Interface) von MySQL nicht sonderlich attraktiv ist und deutlich mehr – langweiligen – Code als das Java-API erfordert. Zudem ist der Konfigurierungsaufwand für die Konnektivität beim C-API erheblich.

1.6 In eigener Sache

Ich möchte nun noch einige – mir wichtige – Anmerkungen machen:

- Alles Schriftliche dient der Kommunikation. Im Fall eines Buches ist es die Kommunikation mit seinen Leser(inne)n, die kein Autor allesamt kennen und deren Vorlieben oder Abneigungen er daher nicht berücksichtigen kann. Diese Kommunikation findet im Allgemeinen indirekt mittels des Textes und der Grafiken statt, die für sich selber sprechen sollen. In etlichen Fällen erschien es mir aber als angebracht, Sie als Leserin oder Leser direkt anzusprechen. Das ist nicht als Kumpelhaftigkeit gemeint, sondern wird von

mir unter anderem dann verwendet, wo es eher um Wahrnehmungen und Meinungen, als um Tatsachen geht (wobei es dazwischen eine beachtliche Grauzone gibt).

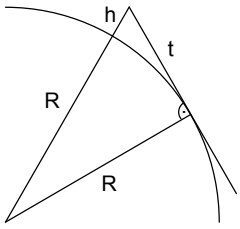
- Wie das Wort „Autor" im vorangehenden Abschnitt zeigt, verwende ich oft nur die männliche Form. Fehler werden von Programmierern gemacht, nicht von Programmiererinnen, die natürlich ebenso wenig unfehlbar sind. Die Autorin Elke Heidenreich soll einen Rundfunkbeitrag einmal folgendermaßen begonnen haben: „Liebe Zuhörer und Zuhörerinnen an den Radioapparaten und Radioapparatinnen", was ihr angeblich einen Shitstorm eintrug. Diese witzige Idee lässt sich leider nicht anhand von Fundstellen im Web belegen, ist ihr aber zuzutrauen. Sie karikierte damit vermutlich die häufig total verkrampften Bemühungen um eine geschlechtsneutrale oder zumindest nicht männlich dominierte Schreibung, was keineswegs gut ankam. Vor dem Hintergrund der weiterhin andauernden Diskussionen um dieses Thema, also um das korrekte Gendern, sind sprachverändernde Bemühungen meiner Ansicht nach zum Scheitern verdammt. Um es Ihnen zu ersparen, sich mit angeblich politisch korrekten, aber unlesbaren Bezeichnungen gequält zu fühlen, habe ich es mir ziemlich einfach gemacht. Bei allen diesbezüglich empfindlichen Seelen möchte ich mich dafür vorab entschuldigen.
- Zudem lässt sich meine kulturelle Prägung als Deutscher nicht verbergen. Warum auch? Jede(r) kann, darf und sollte auf ihre oder seine Kultur stolz sein und sie leben. Diese Prägung manifestiert sich – vielfach unabsichtlich – in Stil und Inhalt. Ebenso wenig wie im vorangehenden Abschnitt ist das als Ausschließung Anderer gemeint. Ganz im Gegenteil habe ich mehrfach internationale Aspekte einbezogen, was ich als erhebliche Bereicherung empfinde. Die Grafiken sind – bis auf eine französische Genealogie – durchgehend in englischer Sprache beschriftet, damit sie nicht bei jeder Übersetzung bearbeitet werden müssen. Das Englische ist die Lingua Franca der IT, und Michael A. Jackson ist Brite.
- Trotz extremer Sorgfalt und mehrfachen, intensiven Überprüfungen des Texts, der Grafiken und der Programme kann ein solches Monumentalwerk nicht fehlerfrei sein. Es ist möglich, dass Sie auf Dinge stoßen, die wirklich falsch und nicht bloß unverständlich sind, was auch schon schlimm genug wäre. Für Hinweise darauf habe ich immer ein offenes Ohr. Erkannte Fehler werden korrigiert und unzulängliche Beschreibungen werden verbessert, sobald ich davon weiß.
- Man kann es nicht allen recht machen. Für manche Anhänger der OO-Glaubensrichtungen (pardon!) grenzt es möglicherweise an ein Sakrileg, dass ausgerechnet auf der Basis einer längst im Treibsand der IT-Geschichte untergegangenen Methode der Versuch unternommen wird, sowohl eine Brücke zwischen der algorithmischen und der OO-Programmierung zu bauen als auch Java-Programme direkt aus JSP-Modellen herzuleiten. Programmierer(innen), die ihre Arbeit zu verlieren drohen, weil die von ihnen jahrzehntelang gewarteten Altsysteme durch neue OO-Software abgelöst werden, können diese Brücke dagegen hoffentlich als einen gangbaren Weg in eine bessere berufliche Zukunft begrüßen.
- Zwangsläufig kommen im Fließtext und in den Grafiken viele englische Worte vor. In letzteren und in den Programmen dominiert die radikale Kleinschreibung, aber im Fließtext kollidiert diese mit der deutschen Groß- und Kleinschreibung. Daher habe ich mich mit dem Englisch-Korrektor und Übersetzer David Roseveare für den Fließtext auf folgende Regeln geeinigt: a) Die englischen Namen von Strukturblöcken werden in Anführungszeichen zitiert. Abkürzungen darin werden gegebenenfalls mit Klammersetzung ausgeschrieben. b) Andere englische Worte werden wie deutsche Worte groß oder klein geschrieben. c) Englische Worte werden nicht „genitiviert" oder anderweitig an deutsche Grammatikregeln angepasst. d) Sie werden untereinander auch nicht mit Bindestrichen verbunden, ausser bei bekannten Ausnahmen wie „object-oriented". Wenn englischen Worten ein deutsches Wort folgt, das nach deutschen Regeln mit einem Bindestrich angeschlossen werden muss, dann geschieht das auch (siehe obiges Beispiel: Top Down-Manier).
- Entspannen Sie sich. Nehmen Sie alles, was Sie hier lesen, mit Humor.

1.7 Last but not least

… bin ich es Ihnen noch schuldig, die kryptische Bemerkung über die Wahrnehmung und die Wissenschaft hinsichtlich des eingangs gezeigten Fotos aufzulösen. Diese Aufnahme ist in mehrfacher Hinsicht aussergewöhnlich:

- Die darauf erkennbaren Berge sind fast immer unsichtbar, weil ein Dunstschleier über dem Golf von Genua sie verbirgt. Nur dadurch, dass kalte Luft kaum Wasserdampf aufnehmen kann, sind solche optischen Überreichweiten ausnahmsweise möglich. Zehn Tage vorher war von Porto Maurizio (Imperia) aus tagsüber auch die ungefähr 140 Kilometer entfernte nördliche Spitze von Korsika, das Cap Corse, sichtbar, wobei sogar Einzelheiten zu erkennen waren.
- Das Original dieser Aufnahme, die mit einem starken Teleobjektiv gemacht wurde, ist nicht ganz so beeindruckend, weil die untere, dunklere Partie darauf nur wenige Details zu erkennen erlaubt. Die von der Kamera gewählte Belichtung hebt zwar sehr schön das – schon ins Orange tendierende – Morgenrot hervor, aber die Tonwertspreizung im Original war zu extrem, um auch den Stadtteil im Vordergrund gut erkennen zu können. Daher wurde in Photoshop mittels des magnetischen Lassos der dunkle Bereich ausgewählt und mit der Funktion „Auto-Kontrast" aufgehellt. Erstaunlich viele Details, die zuvor im Dunkeln versunken waren, traten dadurch ans Licht. Auch dies ist ein schönes Beispiel für die digitalen Techniken, die uns erst seit nicht allzu langer Zeit zur Verfügung stehen.
- Wohl jede(r), die oder der sich einmal am Meer gefragt hat, wie weit denn der Horizont entfernt sei, wird rätseln, „wieviel Berg" es auf diesem Bild eigentlich zu sehen gibt. Es sind ja circa 180 Kilometer vom Aufnahmestandort zu den dort gezeigten Apuanischen Alpen, die erst durch Koordinatenberechnungen identifiziert werden konnten. Deren höchste Erhebung, der Monte Pisanino (1946 m) befindet sich fast ganz links in der Bergkette. Ganz rechts scheint eine Landspitze zu sein, aber das täuscht.

Die geografischen und geometrischen Aspekte sollen nun näher betrachtet werden. Zunächst geht es um die Entfernung zum Horizont. Da 180 Kilometer im Verhältnis zum Erdumfang von circa 40.000 Kilometer viel zu wenig ist, um damit eine brauchbare Grafik zu zeichnen, soll zunächst ein weit kleinerer, kugelförmiger Himmelskörper ohne Atmosphäre betrachtet werden, wobei der Kreisabschnitt zu einem Großkreis um dessen Kern gehört:



$$R^2 + t^2 = (R + h)^2$$
$$= R^2 + 2Rh + h^2$$
$$\Leftrightarrow \quad t^2 = 2Rh + h^2$$
$$\Leftrightarrow \quad t = \sqrt{2Rh + h^2}$$
$$\Leftrightarrow \quad t \approx \sqrt{2Rh}$$

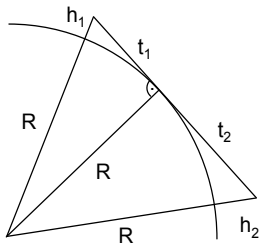
Anmerkung: h^2 kann vernachlässigt werden, wenn $h << R$ gilt, was auf der Erde der Fall ist.

Wenn eine Person von h = 1,60 m Augenhöhe am Rand eines Meeres oder Sees steht, ergibt sich ihre Sichtweite t anhand des mittleren Erdradius R von 6.371 Kilometer zu 4,5 km, wobei der Einfluss der Atmosphäre nicht berücksichtigt ist, die bekanntlich oberflächennahe Strahlen beugt.

Aus der Höhe des Aufnahmestandorts des eingangs gezeigten Fotos, nämlich h = 160 m, errechnet sich somit eine Sichtweite von 45 km bis zu dem Punkt, wo die Tangente t die Wasseroberfläche streift. Diese Höhe ist zwar 100 * 1,6 m, aber wegen der Wurzelfunktion vergrößert sich die Sichtweite nur um das Zehnfache.

Abbildung 1.1: *Ableitung der Tangentenlänge t aus dem Kugelradius R und der Betrachterhöhe h*

Da die Aufnahme auch die Berge jenseits der Sichtweite t zeigt, muss die Grafik erweitert werden:



Weil t_1 = 45 km ist, muss t_2 = 135 km sein, denn $t_1 + t_2$ = 180 km. Um die Sichtweite von 45 km zu verdreifachen, muss h_2 neun mal so groß wie h_1 (160 m) sein, also 1.440 m. Auf einem Himmelskörper ohne Atmosphäre wären also vom Monte Pisanino nur 506 m zu sehen, und das auf eine so große Entfernung!

Abbildung 1.2: *Geometrische Konstellation bei Überreichweiten*

Nun kommen uns zwei Effekte zur Hilfe, der eine wirklich und der andere scheinbar. Der erste ist die terrestrische Refraktion[13]. Darunter versteht man den Effekt, der oberflächennah verlaufende Lichtstrahlen in der Atmosphäre zum Betrachter hin beugt. Genauer gesagt, handelt es sich um den Krümmungsradius der Lichtstrahlen im Verhältnis zur Erdkrümmung, der durchschnittlich circa 13% beträgt. Die Refraktion ist der Grund, warum Sonne und Mond beim Auf- oder Untergang überhaupt zu sehen sind, obwohl sie sich in dieser Zeit, geometrisch gesehen, hinter dem Horizont befinden.

Zitat aus dem genannten Wikipedia-Artikel: »Die Refraktion variiert sehr stark, sie hängt von der aktuellen Dichteschichtung der Atmosphäre ab, genauer vom Gradienten der Feuchtigkeit, der Temperatur und des Druckes der Luft.«

Da die exakten Werte dieser drei Parameter zum Zeitpunkt der Aufnahme nicht bekannt sind, können 13% als Annäherung dienen. Auf der Gesamtstrecke von 180 km beträgt die Krümmung der Erdoberfläche 160 m + 1.440 m = 1.600 m. 13% davon entsprechen 208 m. Die Apuanischen Alpen waren folglich unter dieser Annahme erst ab circa 1.230 m Höhe sichtbar. Alles darunter, also die niedrigeren Berge, sowie die Hügel und das Flachland, bleibt unsichtbar. Das erklärt den Landspitzen-Effekt rechts im Bild.

Circa 710 Meter Rest-Berghöhe des Monte Pisanino sind immer noch nicht sonderlich beeindruckend. Hier kommt der zweite Effekt ins Spiel. Er lässt Sonne und Mond in der Nähe des Horizonts für den Betrachter deutlich größer als am Zenit erscheinen, was am Aufnahmetag auch für die Apuanischen Alpen galt: Die weit entfernte Bergkette wirkte beeindruckend nahe. Hierzu gibt es unter anderem die folgende, ziemlich ernüchternde Fundstelle bei Wikipedia:

> »Die **Mondtäuschung** ist eine optische Täuschung, durch welche der Mond und die Sonne in Horizontnähe größer erscheinen als bei höherem Stand am Firmament. Für diesen insbesondere beim Aufgang des Vollmondes bekannten Effekt gibt es keine physikalische oder astronomische Erklärung. Die Ursache dieses wahrnehmungspsychologischen Phänomens ist nicht endgültig geklärt.«[14]

Die Kamera ließ sich nicht täuschen und machte daher ein Bild, das die tatsächlichen Größenverhältnisse zeigt.

Endnoten

1 **Michael Anthony Jackson** (* 1936) ist ein britischer Informatiker und arbeitet als unabhängiger Berater in London, England und bei AT&T Research, Florham Park, NJ, USA. Er ist Gastprofessor an der Open University in Großbritannien.

Jackson studierte an der Oxford University und war Kommilitone von Tony Hoare. In den 1970er Jahren entwickelte er die Softwareentwicklungsmethode Jackson Structured Programming (JSP), in den 1980er Jahren entwickelte er zusammen mit John Cameron eine Methode zur Systementwicklung (Jackson System Development (JSD)). In den 1990er Jahren entwickelte er die Problem Frames, einen Ansatz um Probleme zu zerlegen und zu strukturieren.

[Seitentitel: Michael Anthony Jackson
Herausgeber: Wikipedia – Die freie Enzyklopädie.
Autor(en): Wikipedia-Autoren, siehe Versionsgeschichte
Datum der letzten Bearbeitung: 23. Oktober 2019, 11:38 UTC
Versions-ID der Seite: 193387996
Permanentlink: https://de.wikipedia.org/w/index.php?title=Michael_Anthony_Jackson&oldid=193387996
Datum des Abrufs: 13. Juni 2022, 15:30 UTC]

2 **Edward Nash Yourdon** (April 30, 1944 – January 20, 2016) was an American software engineer, computer consultant, author and lecturer, and software engineering methodology pioneer. He was one of the lead developers of the structured analysis techniques of the 1970s and a co-developer of both the Yourdon/Whitehead method for object-oriented analysis/design in the late 1980s and the Coad/Yourdon methodology for object-oriented analysis/design in the 1990s.

[Page name: Edward Yourdon
Author: Wikipedia contributors
Publisher: Wikipedia, The Free Encyclopedia.
Date of last revision: 29 July 2022 14:41 UTC
Date retrieved: 16 August 2022 15:23 UTC
Permanent link: https://en.wikipedia.org/w/index.php?title=Edward_Yourdon&oldid=1101143734
Primary contributors: revision history statistics
Page Version ID: 1101143734]

3 Textziffern in eckigen Klammern verweisen auf das Kapitel „9.1 Literaturverzeichnis“ auf Seite 191.

4 Die **Structured Analysis and Design Technique** (**SADT**) ist eine Methode zur Softwareentwicklung, die 1977 von Douglas T. Ross publiziert wurde.

[…]

Basiselement der Modellierung ist die Funktion, dargestellt als Rechteck, die Eingangsgrößen (engl. Input, Pfeile von links) in die Ausgangsgrößen (engl. Output, Pfeile nach rechts) transformiert. Für die Ausführung werden Mechanismen (engl. Mechanisms, Pfeile von unten) benötigt, zum Beispiel Ressourcen. Weitere Einflüsse oder Störgrößen (Control) werden als Pfeile von oben dargestellt.

Mehrere Funktionen können hintereinander ausgeführt werden, in der eine Ausgangsgröße der einen Funktion mit der Eingangsgröße der anderen verbunden wird.

Die schrittweise Modellierung kann top-down erfolgen. Eine Funktion wird hierbei hierarchisch zerlegt, d. h. eine Funktion einer Ebene, wird in der nächsten Ebene als Verschaltung mehrerer Teilfunktion dargestellt.

[Wikipedia -Seitentitel: Structured Analysis and Design Technique
Datum der letzten Bearbeitung: 15. Juli 2019, 15:46 UTC
Versions-ID der Seite: 190456980
Permanentlink: https://de.wikipedia.org/w/index.php?title=Structured_Analysis_and_Design_Technique&oldid=190456980
Datum des Abrufs: 13. Juni 2022, 15:20 UTC]

[Hervorhebung des Autors: Die Quellen von Wikipedia-Zitaten werden ab hier ohne Herausgeber / Autor(en) aufgelistet, weil diese Angaben konstant sind. Dem Wort „Seitentitel" wird daher „Wikipedia-" vorangestellt.]

5 **Edsger Wybe Dijkstra** (* 11. Mai 1930 in Rotterdam; † 6. August 2002 in Nuenen) war ein niederländischer Informatiker. Er war der Wegbereiter der strukturierten Programmierung. 1972 erhielt er den Turing Award für grundlegende Beiträge zur Entwicklung von Programmiersprachen.

[Wikipedia-Seitentitel: Edsger W. Dijkstra
Datum der letzten Bearbeitung: 29. April 2022, 07:04 UTC
Versions-ID der Seite: 222457618
Permanentlink: https://de.wikipedia.org/w/index.php?title=Edsger_W._Dijkstra&oldid=222457618
Datum des Abrufs: 13. Juni 2022, 15:37 UTC]

6 Die **GMD – Forschungszentrum Informationstechnik GmbH** war eine zwischen 1968 und 2001 bestehende deutsche Großforschungseinrichtung für angewandte Mathematik und Informatik.

[…]

Die Gründung erfolgte am 23. April 1968 in Bonn unter dem Namen Gesellschaft für Mathematik und Datenverarbeitung (GMD). Damit sollte das Konzept der Großforschung, das sich im Bereich der Kernenergie bewährt hatte, auf die damals noch Datenverarbeitung genannte Informatik übertragen werden. Hierzu wurde das Institut für Instrumentelle Mathematik (IIM) an der mathematischen Fakultät der Universität Bonn ausgebaut als Großforschungseinrichtung des Bundes mit einer Minderheitsbeteiligung des Landes Nordrhein-Westfalen.

[…]

Im März 1995 erfolgte eine Umbenennung. Auf Initiative des Bundesministeriums für Bildung und Forschung wurde die GMD in den Jahren 2000 bis 2001 gegen den Widerstand der Mitarbeiter, die eine breite Unterstützung durch die regionale Politik erfuhren, mit der Fraunhofer-Gesellschaft fusioniert. […] Die Einrichtung ist im Fraunhofer-Institutszentrum Schloss Birlinghoven aufgegangen.

[Wikipedia-Seitentitel: GMD-Forschungszentrum Informationstechnik
Datum der letzten Bearbeitung: 12. Mai 2021, 10:49 UTC
Versions-ID der Seite: 211872283
Permanentlink: https://de.wikipedia.org/w/index.php?title=GMD-Forschungszentrum_Informationstechnik&oldid=211872283
Datum des Abrufs: 13. Juni 2022, 15:43 UTC]

7 Damals noch unter meinem Geburtsnamen Gomm

8 **TR 440** (gesprochen: T-R-4-40) ist die Bezeichnung des von AEG-Telefunken, Fachbereich Informationstechnik, aus dem „Telefunken-Rechner TR 4" weiterentwickelten Großrechners. AEG-Telefunken lieferte 1969 den ersten TR 440 an das Deutsche Rechenzentrum. Als der TR 440 herauskam, war er der schnellste Rechner, der je in Europa entwickelt worden war. Bis im Jahr 1974 wurden insgesamt 46 Anlagen vom Typ TR 440 gebaut.

Das Gesamtsystem aus Hardware, BS 3 und Programmiersystem wurde auch unter dem Namen TNS 440 (Teilnehmer-System 440) vermarktet.

[…]

Die möglichen Nachfolgesysteme waren weit weniger benutzerfreundlich als das TNS 440; insbesondere gegen die Beschaffung von 7.700-Rechnern mit BS2000® gab es große Widerstände trotz der von Siemens angebotenen Umstellungshilfen. Die Wertschätzung für das TNS 440 bei den Nutzern ging wesentlich auf dessen Systemsoftware zurück: Das Betriebssystem bietet eine Virtuelle Speicherverwaltung mit Speicherschutz und Mehrfachzugriff, das Programmiersystem eine flexible, übersichtliche Kommandosprache, eine gute Ausstattung an Programmiersprachen (einschließlich Sprachverknüpfung) und Programmbibliotheken, sowie innovative Testhilfen für die Programmentwicklung.

[Wikipedia-Seitentitel: TR 440

Datum der letzten Bearbeitung: 20. April 2022, 08:39 UTC
Versions-ID der Seite: 222219658
Permanentlink: https://de.wikipedia.org/w/index.php?title=TR_440&oldid=222219658
Datum des Abrufs: 13. Juni 2022, 15:51 UTC]
[Hervorhebung d. A.: Das Deutsche Rechenzentrum in Darmstadt ging zum 01.01.1973 in der GMD auf.]

9 **System/360** oder kurz **S/360** bezeichnet eine Großrechnerarchitektur der Firma IBM aus dem Jahre 1964. Zuvor wurden von IBM die 700/7000 series gebaut. Dem System/360 folgte das im Jahr 1970 angekündigte System/370.

[…]

Es war eines der erfolgreichsten Computersysteme aller Zeiten. Die von IBM veröffentlichten Spezifikationen erlaubten es außerdem auch anderen Anbietern, Peripheriegeräte für das System/360 anzubieten, häufig kostengünstiger. Umgekehrt begannen andere Firmen wie Amdahl und Univac zum 360-System kompatible Computer zu entwickeln. Ebenso wie die Hardware wurde mit dem System/360 auch die Software vereinheitlicht und kompatibel gemacht.

Die Hauptarchitekten waren Frederick P. Brooks, Gerrit Blaauw, Gene Amdahl. Die Gesamtleitung hatte Bob O. Evans. Beteiligt war auch u. a. Erich Bloch. Brooks, Bloch und Evans erhielten dafür 1985 die National Medal of Technology and Innovation.

[Wikipedia-Seitentitel: System/360
Datum der letzten Bearbeitung: 2. Mai 2022, 16:31 UTC
Versions-ID der Seite: 222558995
Permanentlink: https://de.wikipedia.org/w/index.php?title=System/360&oldid=222558995
Datum des Abrufs: 13. Juni 2022, 15:57 UTC]

10 **Prokrustes** (altgriechisch Προκρούστης (..), deutsch ‚Ausstrecker') war ein Riese aus der griechischen Mythologie, Beiname des Polypemon oder Damastes, eines attischen Räubers in der Umgegend von Eleusis und Sohn des Poseidon.

[…]

In seiner Weltgeschichte berichtet der altgriechische Geschichtsschreiber Diodor (1. Jahrhundert v. Chr.) Folgendes über den Unhold und Wegelagerer Prokrustes:

Prokrustes bot Reisenden ein Bett an, aber in manchen Sagen zwang er auch Wanderer, sich auf ein Bett zu legen. Wenn sie zu groß für das Bett waren, hackte er ihnen die Füße bzw. überschüssige Gliedmaßen ab; waren sie zu klein, hämmerte und reckte er ihnen die Glieder auseinander, indem er sie auf einem Amboss streckte.

Prokrustes wurde von Theseus auf seiner Wanderung nach Athen als letzter der Bösewichte am Kephisos erschlagen.

[…]

Als Prokrustesbett oder Bett des Prokrustes bezeichnet man redensartlich eine Form oder ein Schema, wohinein etwas gezwungen wird, das dort eigentlich nicht hineinpasst.

[Wikipedia-Seitentitel: Prokrustes
Datum der letzten Bearbeitung: 4. Februar 2022, 12:37 UTC
Versions-ID der Seite: 219858623
Permanentlink: https://de.wikipedia.org/w/index.php?title=Prokrustes&oldid=219858623
Datum des Abrufs: 13. Juni 2022, 16:01 UTC]

11 **Grady Booch** (* 27. Februar 1955 in Texas) ist ein amerikanischer Informatiker. Er gilt als Pionier auf dem Gebiet des modularen und objektorientierten Softwareentwurfs und der Klassenbibliotheken (Ada, C++). 1977 schloss er als Bachelor of Science an der United States Air Force Academy ab, 1979 erlangte er den Titel Master of Science an der University of California.

Seit 1980 ist er Chief Scientist der Firma Rational Software in Santa Clara (Kalifornien). Zusammen mit Ivar Jacobson und James Rumbaugh spricht man auch von den Drei Amigos (Begründer der Unified Modeling

Language). Nach der Übernahme von Rational Software durch IBM arbeitet er bei IBM und ist seit 2003 IBM Fellow.

[Wikipedia-Seitentitel: Grady Booch
Datum der letzten Bearbeitung: 25. April 2016, 04:18 UTC
Versions-ID der Seite: 153784355
Permanentlink: https://de.wikipedia.org/w/index.php?title=Grady_Booch&oldid=153784355
Datum des Abrufs: 16. August 2022, 15:58 UTC]

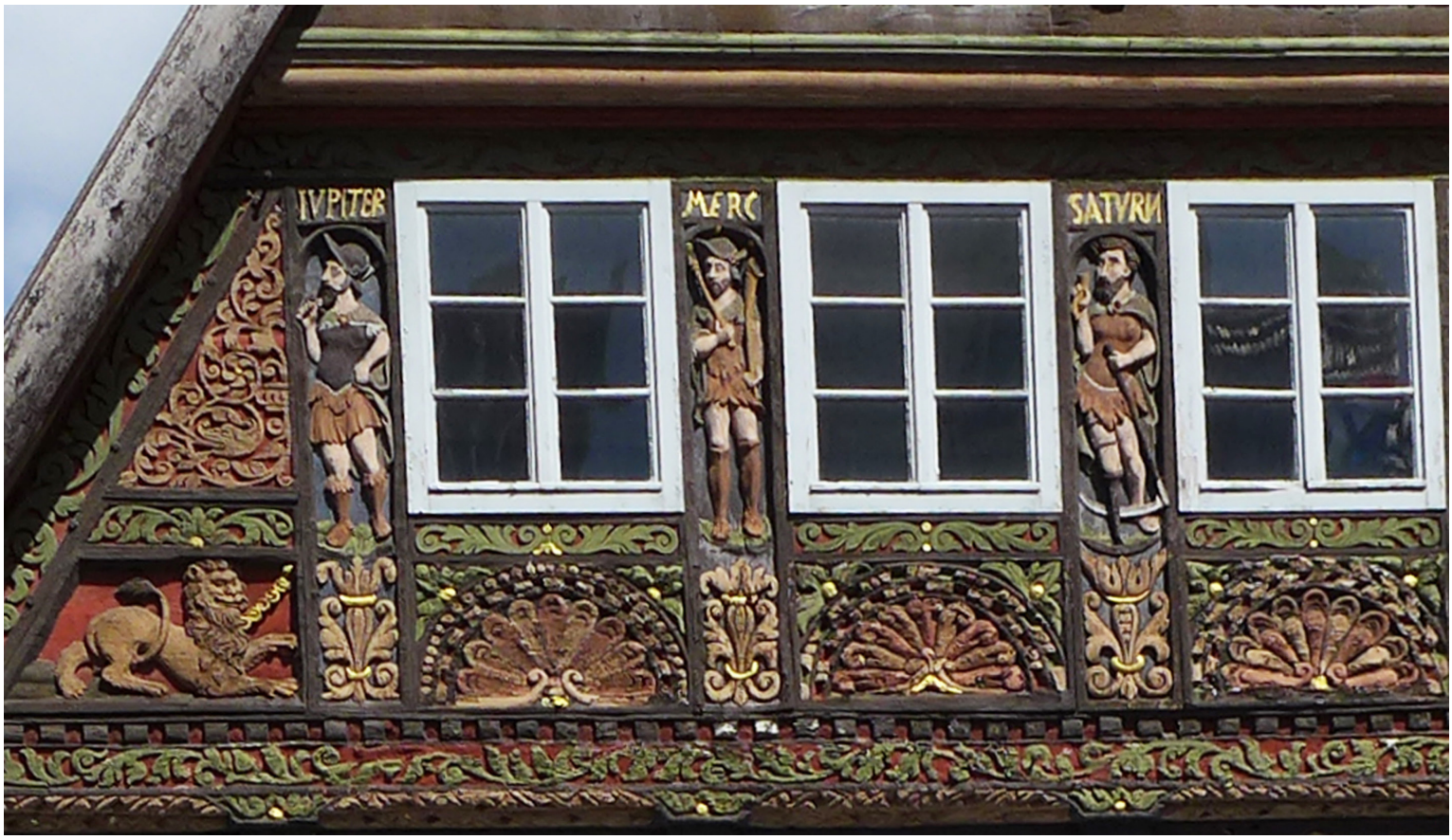
12 Im Oktober 2024 wurden die Paperback-Bände 1 und 2 in die Schwierigkeitsniveaus A1 und A2 aufgeteilt. Die Hauptthemen der A1-Ausgaben sind „Grundlagen" und „Gruppen". In diesem Kontext wurde auch das Titel-Layout der Buchreihe verändert. Die A2-Ausgaben werden mit „Ranggruppen" und „Link-Listen" bezeichnet. Die Hardcover-Bände blieben inhaltlich unverändert. Es werden dort bei der geplanten Neuauflage lediglich alle vier Hauptthemen auf dem Titel aufgeführt und im Band 1-A1 das Unterkapitel zu den Nassi-Shneiderman-Diagrammen ergänzt. Die zweite und die dritte Staffel werden analog zur ersten Staffel aufgebaut sein. Dort soll es jeweils zwei Hardcover-Bände und vier Ringbuch-Bände geben, die den Schwierigkeitsniveaus B1, B2, C1 und C2 entsprechen.

13 Als **terrestrische Refraktion** (auch Strahlenbrechung oder atmosphärische Refraktion genannt) wird die Brechung eines Lichtstrahls in der untersten Erdatmosphäre bezeichnet. Diese entsteht durch die Änderung des Brechungsindexes der Luft entlang des Strahlverlaufs infolge der mit der Höhe abnehmenden Luftdichte und bewirkt eine bogenförmige Krümmung des Strahls, die bei genaueren Vermessungen oder im physikalischen Labor als Korrektion („Reduktion") an jedem gemessenen Vertikalwinkel angebracht werden muss. Diese Strahlkrümmung beträgt durchschnittlich 13 % der Erdkrümmung und erhöht die horizontale Sichtweite geringfügig.

[Wikipedia-Seitentitel: Terrestrische Refraktion
Datum der letzten Bearbeitung: 2. Mai 2022, 17:22 UTC
Versions-ID der Seite: 222560036
Permanentlink: https://de.wikipedia.org/w/index.php?title=Terrestrische_Refraktion&oldid=222560036
Datum des Abrufs: 13. Juni 2022, 16:08 UTC]

14 Erste Hinweise auf das Phänomen der Mondtäuschung finden sich auf Tontafeln aus den königlichen Bibliotheken von Niniveh und Babylon (6. Jahrhundert v. Chr.). Ptolemäus (ca. 150 n. Chr.) vermutete fälschlicherweise vergrößernde Eigenschaften der Atmosphäre. Alhazen (Abu Ali al-Hasan ibn al-Haitham, 965 bis ca. 1040) stellte fest, dass der Mond sowohl am Horizont als auch im Zenit die gleiche Größe hat, und schrieb bereits vom abgeflachten Firmament […] als Ursache der Wahrnehmungstäuschung. Auch Leonardo da Vinci, Johannes Kepler und René Descartes beschäftigten sich mit der Mondtäuschung. Seit über 100 Jahren wird diese optische Täuschung von der wissenschaftlichen Wahrnehmungspsychologie untersucht. Dennoch ist das Phänomen noch immer nicht eindeutig geklärt, es bleiben Widersprüche bei den unterschiedlichen Erklärungsansätzen. Die derzeit anerkanntesten und von vielen Experimenten untermauerten Erklärungen sind die der falsch eingeschätzten Entfernung mit dem abgeflachten Firmament und das Prinzip der Vergleichsobjekte.

[Wikipedia-Seitentitel: Mondtäuschung
Datum der letzten Bearbeitung: 16. Mai 2022, 21:32 UTC
Versions-ID der Seite: 222937687
Permanentlink: https://de.wikipedia.org/w/index.php?title=Mondt%C3%A4uschung&oldid=222937687
Datum des Abrufs: 13. Juni 2022, 16:14 UTC]



06/2022 Jupiter, Merkur und Saturn am Planetenhaus in Lemgo (NRW)

JSP-Basismethode

Jahrtausendelang beobachteten astronomisch interessierte Menschen die Himmelskörper nur mit bloßem Auge, also ohne technische Hilfsmittel. Ihre Erkenntnisse ermöglichten dennoch im dritten Jahrtausend v.Chr. die ägyptische Entwicklung des 365-Tage-Kalenders, den König Ptolemaios III im Jahr 238 v.Chr. wegen der offensichtlichen Kalenderdrift um die Schalttagsregel mit einem zusätzlichen Tag nach jeweils vier Jahren ergänzte[1]. Julius Cäsar setzte diesen Kalender im Jahr 45 v.Chr. für das römische Reich in Kraft.

Im 14. Jahrhundert erkannten Astronomen erneut eine Kalenderdrift, die durch drei überflüssige Schalttage im julianischen Kalender in jeweils vier Jahrhunderten verursacht worden war. Diesen Fehler behob erst die gregorianische Kalenderreform[2] von 1582, bei der im Oktober 10 Tage ausgelassen wurden, was zu erheblicher Verwirrung in den papsttreuen Ländern und langdauernden Widerständen gegen diese Reform in den anderen Ländern führte.

Das Jahr 1576, das auf der Fassade des 1590 - 1595 erbauten „Planetenhauses[3]" in Lemgo genannt wird, lag also nur sechs Jahre vor diesem historischen Einschnitt. Auf dem Halbrelief am Giebelfuß sind die bis dahin bekannten Planeten Jupiter (JVPITER), Merkur (MERC), Saturn (SATVRN), Venus (VENVS) und Mars (MARS), sowie Sonne (SOL, Mitte) und Mond (LVNA, rechts) figürlich abgebildet. Saturn hat ein Holzbein, weil er der langsamste Planet ist. Sonne und Mond galten aufgrund der Lehre des Aristoteles (384 – 322 v. Chr.) ebenfalls als Planeten.

Rund 450 Jahre später ist die moderne Astronomie ohne Computer überhaupt nicht mehr denkbar. Jedes Gebiet der IT ist nicht nur daran, sondern auch an Raumfahrtmissionen, der Satellitennavigation, der Erderkundung und vielem mehr intensiv beteiligt. Die dadurch angestoßenen Forschungen der NASA zur Fehlererkennung und -vermeidung in der Programmierung und im Hardware-Design haben zwar neue Qualitätsstandards gesetzt, aber auch die bittere Erkenntnis zutage gefördert, dass Fehler in komplexen Systemen unausrottbar zu sein scheinen.

Das Jackson Structured Programming erhebt den Anspruch, strukturelle Fehler in Programmen von vornherein vermeiden zu können, wodurch die Fehlermöglichkeiten drastisch reduziert werden. Im folgenden Kapitel geht es darum, das grundsätzliche Vorgehen des JSP an einem einfachen Beispiel zu zeigen.

2.1 Vorbemerkungen

Programmieren lernt man durch Programmieren, so scheint es. Beginnend mit einfachen Beispielen und wenigen Sprachelementen führt ein langer Weg voller Experimente und Irrtümer nach und nach zu steigender Könnerschaft. Die Befriedigung, ein schwieriges Problem nach dem anderen erfolgreich gelöst zu haben, hilft dabei sehr über Durststrecken und die Erkenntnis der eigenen Grenzen hinweg. Solange sich alles axiomatisch zueinander fügt, also einfache, folgerichtig aufeinander aufbauende Überlegungen zum Ziel führen, scheint dieser Weg immer weiter gangbar zu sein. Früher oder später endet er jedoch vor einem undurchdringlichen Dickicht aus zunehmender Komplexität. Nun hilft auch hohe Intelligenz nicht wirklich weiter, vor allem angesichts der Risiken, die sich in großen Programmsystemen verbergen.

Dieses Vorgehen ähnelt stark dem Erwerb einer Fremdsprache auf intuitiver Basis mit ein wenig Unterricht und dem Sammeln von Erfahrungswissen. Auf diesem Weg mag eine passable Sprachbeherrschung erreicht werden, aber: schwierige Texte richtig zu verstehen, anspruchsvolle Diskussionen zu führen oder gar eine eigene literarische Tätigkeit auszuüben ist so nicht möglich. Es fehlt das Strukturwissen, das mit der Grammatik beginnt und weit darüber hinaus die Sprachen und ihre Sprecher prägt. Jede menschliche Sprache erschafft eine eigene Welt, die sich von der einer anderen Sprache durchaus wesentlich unterscheiden kann.

Obwohl Programmiersprachen gegenüber natürlichen Sprachen ausdrucksarm sind und einem begrenzten Zweck dienen, gilt für sie analog dasselbe: Ohne Strukturwissen, ohne Grammatik kann keine Meisterschaft

Endnoten: siehe Seite 49

erworben werden. Mit Grammatik ist hier nicht die mehr oder minder simple Syntax der Programmiersprachen gemeint, sondern der ihnen allen gemeinsame Hintergrund des Konstruierens von Softwareprodukten.

Konstruktion? Produkt? Ja, auch ein (un)gelernter Programmierer oder eine Programmiererin konstruiert Produkte, indem er oder sie Ursache-Wirkungsketten beachtet und somit aufeinander aufbauende Maschinen erschafft, deren Zusammenwirken tatsächlich erfolgreich die jeweilige Aufgabe löst – von der Plage der Programmierfehler mal abgesehen. Sie oder er konstruiert aber kreativ und intuitiv, was ihrem oder seinem Ego durchaus schmeichelt, so dass sich manche dieser Programmierer(innen) als Künstler bezeichnen. (Wenn das Kunst wäre, dann gute Nacht!).

Wer so arbeitet, hat kein ernstzunehmendes Strukturwissen, sondern nur eine Kombination aus Erfahrung und Bauernschläue vorzuweisen – aber das reicht bei weitem nicht aus, wenn es wirklich schwierig wird. Es kommt daher darauf an, die Grammatik der Konstruktion von Software kennenzulernen und mit ihr sinnvolle Programmprodukte zu formen. Wie beim Spracherwerb kann das nicht als Trockenkurs zum Erfolg führen. Wir lernen an Beispielen, obwohl Beispiele immer willkürliche – aber begründet ausgewählte – Ausschnitte aus einem großen Ganzen sind. Wir lernen, indem wir deren Besonderheiten von den allgemeinen Gesetzmäßigkeiten trennen, deren Wirken sich in den Beispielen ausprägt.

2.2 Das erste JSP-Beispiel

So beginnt auch diese Einführung in das JSP mit einem Beispiel. Es sieht sehr einfach aus, enthält aber dennoch die Kernelemente der Methode. Die Aufgabe besteht darin, einen Text einzulesen, darin die Zeichen und Zeilen zu zählen, und den Text mit den Zählergebnissen zusammen auszugeben. Folgende Aspekte sind dabei zu beachten:

- Leerzeichen nach den Textzeichen werden nicht mitgezählt.
- Leerzeilen enthalten 0 Zeichen, also weder Leerzeichen noch Textzeichen.
- Am Beginn der Ausgabe ist ein Titelbereich mit Spaltenüberschriften (Listenkopf) vorzusehen.
- Rechts neben den Textzeilen sind die zeilenbezogenen Zählerwerte (Leerzeichen | andere Zeichen | Zeilenlänge) auszugeben.
- Dieser Ausgabe folgen eine Trennzeile und eine Zeile, welche die Gesamtsummen der Einzelzähler enthält (Listenfuß).
- Bei einer leeren Eingabedatei sind nur Listenkopf und Listenfuß auszugeben.
- In den Eingabezeilen sind nur lesbare Zeichen und Leerzeichen zulässig, keine Tabulatoren etcetera.

Die Eingabe[4] lautet beispielsweise:

```
When, in disgrace with fortune and men's eyes,
I all alone beweep my outcast state
And trouble deaf heaven with my bootless cries
And look upon myself and curse my fate.

Wishing me like to one more rich in hope,
Featur'd like him, like him with friends possess'd,
Desiring this man's art and that man's scope,
With what I most enjoy contented least.

Yet in these thoughts myself almost despising,
Haply I think on thee, and then my state,
```

```
Like to the lark at break of day arising
From sullen earth, sings hymns at heaven's gate.

For thy sweet love remember'd such wealth brings
That then I scorn to change my state with king's.

William Shakespeare   sonnet XXIX
```

Der Eingabetext wird in einer sequentiellen Datei erwartet. Eine solche Datei enthält die Daten in Form von Sätzen, die in der angegebenen Folge abgelegt sind. Jede Zeile des Gedichts, jede Leerzeile und auch die letzte Zeile entsprechen je einem Satz in der Datei.

Die physische Implementierung dieser Datei tut hier nichts zur Sache. Das wird von den Betriebssystemen jeweils verschieden gehandhabt. Entscheidend ist allein, dass die Sätze sequentiell gelesen werden können, ohne sich damit herumschlagen zu müssen, wie sie genau gespeichert sind.

Die Ausgabe sieht zum Beispiel wie folgt aus:

```
Text                                                  chars  blanks  length

When, in disgrace with fortune and men's eyes,           39       7      46
I all alone beweep my outcast state                      29       6      35
And trouble deaf heaven with my bootless cries           39       7      46
And look upon myself and curse my fate.                  32       7      39
                                                          0       0       0
Wishing me like to one more rich in hope,                33       8      41
Featur'd like him, like him with friends possess'd,      44       7      51
Desiring this man's art and that man's scope,            38       7      45
With what I most enjoy contented least.                  33       6      39
                                                          0       0       0
Yet in these thoughts myself almost despising,           40       6      46
Haply I think on thee, and then my state,                33       8      41
Like to the lark at break of day arising                 32       8      40
From sullen earth, sings hymns at heaven's gate.         41       7      48
                                                          0       0       0
For thy sweet love remember'd such wealth brings         41       7      48
That then I scorn to change my state with king's.        40       9      49
                                                          0       0       0
William Shakespeare   sonnet XXIX                        28       5      33
                                                       ====    ====    ====
                                                totals  542     105     647
```

Als Ausgabemedium kann ein Drucker, eine sequentielle Datei, eine sequentiell aufgebaute Bildschirmseite etcetera verwendet werden. Die langfristige Speicherung einer solchen Ausgabe in einer Datei dürfte die Ausnahme sein, da sie ja jederzeit aus der Eingabe erzeugt werden kann.

Operative Berichtsdateien werden aber häufig archiviert, also sequentiell abgelegt und dann auf ein billiges und langfristig zuverlässiges Medium kopiert. Daher wird in diesem Listenausgabebeispiel eine echte Dateiausgabe vorgenommen, obwohl viele Betriebssysteme einfache Ausgaben zum Beispiel per C-printf-Funktion, COBOL-DISPLAY- oder REXX-SAY-Anweisung unterstützen.

2.3 Der intuitive Ansatz

Wie würde ein Programmierer vorgehen, der die strukturierte Programmierung nicht kennt, um diese Aufgabe zu lösen? Er würde wahrscheinlich zunächst nach einem Programm suchen, das für eine ähnliche Aufgabe geschrieben wurde, um es passend abzuändern. Das ist viel bequemer und geht – angeblich – schneller, als alles neu einzugeben. Irgendein Programm mit einer einfachen Einleseschleife wird sich schon finden. Nun müssen nur noch die Details der Ein- und Ausgabe-Satzdefinitionen abgeändert und die fehlenden Anweisungen zum Zählen der Zeichen hinzugefügt, sowie alles nicht mehr Benötigte gelöscht werden, und schon kann der erste Compile stattfinden – falls es sich nicht sowieso um eine Interpretersprache handelt.

Es folgt der erste Testlauf, bei dem das neue Programm leider sofort abbricht. Irgendetwas wurde übersehen oder vergessen, das repariert werden muss, und schon beginnt der zweite Versuch. Diesmal kommt auch eine Ergebnisliste zustande, aber die Summen ganz unten stimmen nicht. Das ist wirklich sonderbar, wo das Programm doch so einfach ist. Nach einigem Nachdenken fällt die Entscheidung für ein ausführliches Ablaufprotokoll, das entweder automatisch angefordert oder mit Hilfe zusätzlicher Anweisungen erzeugt wird. Es zeigt, dass bei Leerzeilen versehentlich die Zählerwerte der ihnen vorhergehenden gefüllten Zeilen aufaddiert werden, weil die Initialisierung der zeilenbezogenen Zähler falsch plaziert ist. Das lässt sich leicht korrigieren – aber jetzt sind alle Summen gleich Null. Dies war also doch nicht die richtige Änderung. Nach erneuter Korrektur stimmen die Summen endlich. Geschafft!

Nein, nicht ganz. Der Test mit der leeren Eingabedatei fehlt noch. Das Programm bricht nun erneut ab. Warum? Ach ja, bei leerer Eingabedatei muss ein kontrollierter Aussprung aus der Hauptschleife stattfinden, damit die Fußzeilen noch gedruckt werden. Das macht einige Mühe, klappt aber dann gut. Nun noch ein letzter Test mit gefüllter Eingabedatei – aber diese Liste ist so kurz wie bei leerer Eingabe. Es ist zum Verzweifeln. Noch ein Start! … und jetzt kommt die Liste richtig heraus. Das ist merkwürdig. Mal klappt es und mal nicht. Ein intensives Studium der veränderten Schleifenende-Abfrage fördert zutage, dass sie sich jetzt auf eine nicht initialisierte Variable bezieht.

Je nachdem, was der vorhergehende Programmlauf in diesem Speicherplatz hinterlassen hat, wird die Schleife bei gefüllter Eingabe normal durchlaufen oder sofort wieder beendet. Nach einer letzten Korrektur funktioniert endlich alles. Unser Programmierer kann sich neuen Herausforderungen zuwenden.

Angenommen, es handle es sich hier nicht um ein einfaches Beispielprogramm, sondern um ein echtes Produktionsprogramm, so wird unser Programmierer bei dessen ersten Einsätzen ein gewisses Kribbeln im Bauch spüren, denn er könnte ja einen Fehler übersehen haben. Aber, so scheint es, alles ist gut. Das Programm läuft störungsfrei. Geschafft!

Monate später kommt allerdings eine E-Mail, dass dieses Programm nachts abgebrochen werden musste, weil es in einer Endlosschleife lief und darin sagenhaft viel CPU-Zeit – und somit Geld – verbrauchte. Nun beginnt eine erneute Fehlersuche und –behebung, nachdem ein Vorgesetzter sich einige unfreundliche Worte über sein unnötig belastetes Budget nicht verkneifen konnte. Die Ursache: Es wurde vergessen, das Programm mit einer Zeile zu testen, die auf der letzten Position rechts (55) ein druckbares Zeichen – kein Blank – enthält. Die Ermittlung der letzten gefüllten Zeichenposition arbeitete mit einem kleinen, aber raffinierten Trick, der leider in diesem Fall nicht funktionierte. Künstlerpech! Ein echter Mann lässt sich davon aber nicht beeindrucken und erschafft auf dieselbe Weise viele schöne Zeitbomben, die auf Dauer seinen Arbeitsplatz sichern – oder irgendwann vernichten.

Das kann Ihnen doch nicht passieren! So dumm sind Sie nicht! Schließlich haben Sie jahrelange Erfahrung und Ihre Programme laufen normalerweise problemlos. Dennoch werden Sie dieses dumpfe Gefühl nicht los, dass jede Nacht in Produktion etwas Schlimmes passieren kann, und Sie sind dann daran schuld. Das kann Sie schlimmstenfalls Ihre berufliche Existenz kosten, Sie aber mindestens zum Gespött Ihrer Kollegen machen – die es auch nicht besser können.

2.4 Konstruieren anstatt Erfinden

In dieser Buchreihe erlernen Sie eine Methode, mit der Sie sich und Anderen viel Kummer ersparen können. Wie bei allem, was man neu lernt, gibt es anfangs eine Phase der Unsicherheit und des Unwissens, die nur schwer ertragen wird. Erfolge müssen her! Nach Jahrzehnten der Anwendung dieser Methode ist jedoch klar, dass auch hier Schritt vor Schritt gesetzt werden muss.

Dieses erste, sehr einfache Beispiel dient dazu, Sie mit den Begriffen, Grafiken und Vorgehensweisen der JSP-Methode vertraut zu machen. Später kommen weitaus schwierigere Themen und Lösungsverfahren, für die Sie das hier dargestellte elementare Wissen benötigen. Lassen Sie sich Zeit und werden Sie nicht ungeduldig, wenn nicht alles hopplahopp geht. Klar, ein erfahrener Programmierer oder eine erfahrene Programmiererin schafft eine so einfache Aufgabe auch ohne JSP-Wissen problemlos, und ein Programm zur Zeichenzählung in Gedichten wird wohl kaum jemals in einen Produktionsablauf eingebunden. Ein wenig Übertreibung war aber nötig, um Ihnen zu zeigen, dass es auch ganz anders laufen kann. Bei wirklich schwierigen Programmen, zumal solchen, von denen viel abhängt (Produktion, Termineinhaltung, Geld, Reputation, Gesundheit, Leben und Tod), ist es von elementarer Bedeutung, dass es wegen vermeidbarer Entwurfs- oder Programmierfehler nicht zu Fehlfunktionen oder Abbrüchen kommt.

Sie stehen am Beginn einer Veränderung, nach der Sie nie wieder ein komplexes Programm Zeile für Zeile erfinden werden. Das heißt nicht, dass Sie ab jetzt sklavisch alles auf JSP-Weise darstellen und jedes banale Programmierproblem so lösen sollten. Diese Übertreibungen waren für eine Zeit seit etwa 1975 typisch, als neue Methoden des Software Engineering rasante Verbreitung erfuhren und die Gurus dieser zu Produkten aufgemotzten Ideen damit extrem viel Geld verdienten. Zweifel an der alleinseligmachenden Wirkung solcher Zwangsjacken wurden als unprofessionell beiseitegeschoben. So sind damals auch zahllose Diagramme gezeichnet worden, die völlig für die Katz waren.

Heute sind wir über eine solche Hybris und Methodenanbetung natürlich weit hinaus… oder etwa nicht? Doch, sind wir, allerdings nicht aus guten Gründen. Ausser in der objektorientierten Programmierung, wo es ohne eine gewisse Disziplin überhaupt nicht funktioniert und wo die Method Wars der vergangenen Jahrzehnte tobten, ist in der als veraltet geschmähten algorithmischen Programmierung ein Rückfall in die Zeiten vor Jackson zu verzeichnen. Junge Programmierer(innen) lernen die JSP-Methode nicht mehr kennen – ebenso wenig COBOL, das als uralte Sprache gilt, die nur noch von pensionsreifen Programmierer(inne)n beherrscht wird und dringend durch Java, Objective C oder irgendeine andere neuere Sprache abgelöst gehört. Diese Sprachen werden allerdings spätestens in zwei, drei Jahrzehnten auch wieder als komplett veraltet gelten…

Es ist daher an der Zeit, die Dinge wieder vom Kopf auf die Füße zu stellen und daran zu erinnern, welch ungeheure Investitionen in Programmen stecken, die in COBOL oder einer anderen Sprache der dritten Generation geschrieben wurden und werden. Im Interesse der Investitionssicherheit und Ihrer ungestörten Nachtruhe ist es daher von entscheidender Bedeutung, die algorithmische Programmierung auf der Basis einer soliden Methode anzuwenden. Auch die objektorientierte Programmierung kann davon profitieren, wie sich in den folgenden Kapiteln immer wieder erweisen wird.

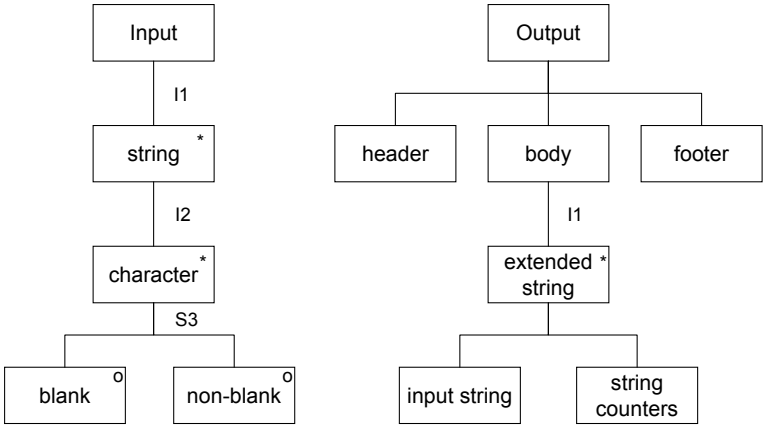
Sie haben doch gewiss schon verblüfft ein bedrucktes und ein nahezu leeres Blatt – nur mit Kopf- und Fußzeile – von der Druckerablage genommen, obwohl die Ausgabe nur eine Seite lang sein sollte. Dann können Sie sicher sein: Die Programmierer(innnen) der zuständigen Ausgabemethode haben noch nie etwas von Programminversion gehört. Sie aber werden nicht nur davon lesen, sondern dieses Vorgehen auch anzuwenden lernen. Aber lassen Sie uns mit weit einfacheren Dingen beginnen.

Die folgende Darstellung führt Sie schrittweise zum Endprodukt; das ist zunächst das funktionierende algorithmische Programm. Sie werden später nicht jeden dieser Schritte durchlaufen, die ersten aber immer, wenn Sie eine schwierige Nuss zu knacken haben.

Danach wird dasselbe Problem objektorientiert gelöst, ausgehend von der letzten JSP-Entwurfsstufe, der Programmstruktur mit Operationen. Hier bestehen erheblich mehr Freiheitsgrade für die Implementierung, die erkannt und sinnvoll genutzt werden müssen.

2.5 Strukturen der Ein- und Ausgabedaten identifizieren

Das folgende Diagramm zeigt die Strukturen der Ein- und Ausgabedaten:



controls:

I1 end of input file
I2 end of string
S3 character is blank

Abbildung 2.1: *Datenstrukturen der Ein- und Ausgabe*

Bevor die Programmstruktur aus den Datenstrukturen abgeleitet werden kann, ist es wichtig, diese Struktogramme zu verstehen.

2.5.1 Vom konkreten Eingabebeispiel zur abstrakten Eingabestruktur

Die Grafik auf der nächsten Seite zeigt die Beziehungen zwischen den Eingabezeilen und ihren abstrakten Entsprechungen im Eingabe-Struktogramm. Es ist leicht zu erkennen, wie die konkreten Dateninhalte des Beispiel-Texts zu Begriffen generalisiert und den Strukturblöcken eindeutig zugeordnet wurden. Diese Strukturblöcke bilden eine Hierarchie, die durch die Relation „besteht aus" zusammengehalten wird, also:

- Die Eingabe (Input) besteht aus aufeinanderfolgenden Eingabezeilen.
- Jede Textzeile ist eine Zeichenkette (String), also einer Wiederholstruktur aufeinanderfolgender Zeichen (Character).
- Jedes dieser Zeichen ist entweder ein Leerzeichen (Blank) oder ein Textzeichen (Non-Blank), also ein Buchstabe, eine Zahl, oder ein Sonderzeichen, zum Beispiel ein Satzzeichen. Die Textzeichen werden nicht voneinander unterschieden, obwohl das möglich wäre. Die Kategorisierung der Eingabeelemente wird hier abgebrochen, weil die Aufgabe keine weitere Einteilung erfordert.

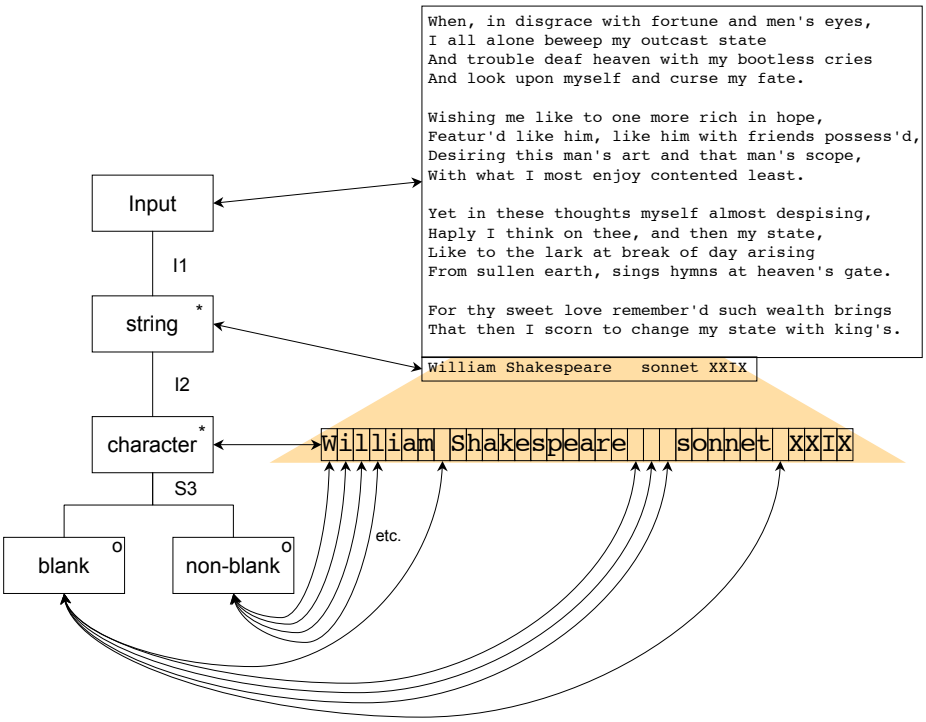


Abbildung 2.2: *Ableitung der Eingabe-Datenstruktur*

An den Verbindungslinien der Hierarchie und in den Strukturblöcken sind Symbole und Ziffern zu sehen, welche die Art der Relation präzisieren. Die an den Linien angebrachten Kürzel müssen sämtlich in der kurzen Liste „Kontrollen" wieder auftauchen, die links unter den Datenstrukturen der Ein- und Ausgabe steht.

2.5.2 I = Iteration

Das „I" bedeutet „Iteration" = Wiederholstruktur. Die 1 ist eine laufende Nummer für die Bedingung, die man wie einen Index tiefgestellt schreiben kann, aber nicht muss.

- „I1 end of input file" bedeutet, dass die Bedingung erfüllt ist, wenn anstatt der Bereitstellung der nächsten Eingabezeile der EOF-Status gesetzt wird.
- Unter einem „I" darf nur ein einziger Strukturblock stehen, der beim Programmieren dem Inneren einer Schleife – als einziges Statement oder als Compound Statement – entspricht.
- Eine Iteration darf auch leer sein, kann also 0, 1 , n >= 2 Elemente enthalten. Ihr entspricht die abweisende Schleife, also diejenige Schleife, die mit der Endeabfrage beginnt.
- In den Iterations-Strukturblöcken steht rechts oben ein Multiplikations-Zeichen (*). Das mag zunächst redundant erscheinen. Tatsächlich erlaubt Jackson aber, Iterationen zunächst ohne Endebedingung anzugeben. Daher ist der Stern auf jeden Fall erforderlich. Ausserdem schützt diese Konvention vor Irrtümern oder Versäumnissen, die beim Zeichnen der Struktogramme passieren können. Die Diagramme werden dadurch auch leichter lesbar.

- Sie sollten es sich aber zur Gewohnheit machen, zumindest das I an die Linie über dem *-Strukturblock zu schreiben und später eine laufende Nummer hinzuzufügen. Irgendwann muss ja doch eine programmierbare Endebedingung angegeben werden, woran das einsame I zusätzlich erinnert.

2.5.3 S = Selection (Auswahl)

Das „S" bedeutet „Selection" = Auswahl beziehungsweise Alternative[5]. Da die Bedingungen durchnumeriert werden, was aber nicht lückenlos geschehen muss, wurde nach I2 das Kürzel S3 vergeben. Die Bedeutung ergibt sich scheinbar von selbst, aber das in der Auswahlbedingung zuerst genannte S-Element muss im Diagramm links unter dem übergeordneten Strukturblock stehen, nach rechts gefolgt vom nächstgenannten S-Element und so weiter, wenn eine Alternative mehr als zwei S-Elemente hat. Bei einer einfachen Alternative ist nur die Bedingung für den linken Zweig anzugeben. Die Bedingung für den S-Strukturblock ganz rechts ergibt sich somit als Rest aus all jenen Bedingungen, für die sich die weiter links stehenden Strukturblöcke qualifizieren..

- Je nach Programmiersprache nennen Programmierer(innen) die multiplen Alternativen zum Beispiel „Case-Struktur" oder „Evaluate-Struktur". Hier geht es aber um Alternativen zwischen Daten(strukturen), nicht um Code-Alternativen.
- Es gibt auch den Fall, dass unter einem „S" nur ein einziger Strukturblock steht. Dann handelt es sich um einen optionalen Teil des Struktogramms. Die Bedingung ist wie bei einer einfachen Alternative anzugeben.
- In jedem Alternativ-Strukturblock steht rechts oben ein kleines „o" für „or" (oder beziehungsweise optional). Jackson erlaubt auch Alternativen ohne Angabe der Auswahlbedingung, etwa in trivialen Fällen. Daher ist das „o" immer einzutragen. Spätestens in seinem Pseudocode namens Schematische Logik verlangt Jackson aber immer, die Auswahlbedingung anzugeben. Daher sollten Sie auch stets das „S" über der Alternative notieren und später numerieren, sowie definieren.

2.5.4 Blöcke ohne S oder I

Nicht gekennzeichnete Strukturblöcke stehen für abstrakte Datenelemente oder Datengruppen, die weder wiederholt noch alternativ beziehungsweise optional auftreten. Das ist im Eingabe-Struktogramm nur beim Strukturblock „Input" gegeben. Im Ausgabe-Struktogramm sind dagegen mehrere solche Elemente enthalten, die aufeinander folgen („sequence" = Folge). Verallgemeinernd werden solche Strukturblöcke selbst dann als Folge-Elemente bezeichnet, wenn sie einzeln auftreten.

2.5.5 Heikle Ratschläge

In der JSP-Literatur findet sich auch der Rat, die Kontrollen anfangs undefiniert zu lassen und sie erst später – nach der Zuordnung der Operationen zu den Programmstrukturen – hinzuzufügen. Das hat aber zur Folge, dass unangenehme Überraschungen erst sehr, sehr spät bemerkt werden und dann irgendwie abgefangen werden müssen.

Michael A. Jackson drückte diesen Rat nicht explizit aus, sondern er verzichtete in [5] auf die Listen der Operationen und der Kontrollen, die Sie in dieser Buchreihe als Standards finden. In seinen Struktogrammen sind den Symbolen für Iteration oder Alternative keine Bedingungsterme zugeordnet; diese ergeben sich erst aus dem jeweiligen Strukturtext, den er „Schematische Logik" nannte, oder aus seinen COBOL-Beispielen. Die Operationen sind – wenn überhaupt – in seinen Grafiken nur angedeutet. Es gibt bei ihm keine Auflistungen einzelner Operationen in Kästchen unterhalb der Strukturblocke-Ebene.

Klaus Kilberth ging in [2] anders vor. Er zeichnete zunächst die Daten-Struktogramme und diskutierte damit bereits die Korrespondenzbedingungen, zu denen wir erst im folgenden Kapitel „2.6 Korrespondenzen zwischen den Datenstrukturen ermitteln" auf Seite 31 ff kommen werden, allerdings ohne sich explizit mit den Abgrenzungen zwischen den Daten-Teilmengen zu befassen, welche ja erst durch die Kontrollen definiert werden. In seinem „Textschritt", den er im Kapitel 2.5 ab Seite 94 explizit beschreibt, besteht der erste Teil darin, dass erst nach (!) der Ableitung der Programmstruktur die Kontrollen spezifiziert und ihr zugeordnet werden. Dann ist der Schaden durch nicht erkannte Strukturkonflikte wegen Ableitung einer deshalb unbrauchbaren Programmstruktur aber schon angerichtet.

In [4] haben dagegen R. Legde / K. Truöl bereits die Listen der Operationen und der Kontrollen eingeführt. Heikel ist dagegen, dass die Autoren auf der Seite 4-15 die folgende Maxime formulierten:

> »Bei der Erstellung der Programmstruktur aus den Datenstrukturen nach diesen formalen Regeln bleibt zunächst unberücksichtigt, dass eventuell die Bedingungen der Kontrollanweisungen für Auswahlen oder Wiederholungen an den entsprechenden Stellen nicht geprüft werden können, da die dafür erforderliche Information erst zu einem späteren Zeitpunkt im Programmablauf vorliegt. Diese Schwierigkeiten werden erst später bei der Codierung berücksichtigt.«

Warum ist das heikel?

Das zeigt der Band 5-C1 anhand von Beispielen, bei denen es zu solchen „losen Enden" kam bzw. kommen kann. Um einen erheblichen Vorgriff an dieser Stelle zu vermeiden, sei nur auf das kommende Kapitel „2.6 Korrespondenzen zwischen den Datenstrukturen ermitteln" auf Seite 31 ff verwiesen. Jedesmal, wenn die dort definierten Voraussetzungen für das Verschmelzen der Datenstrukturen zu einer gemeinsamen Programmstruktur nicht existieren, sind besondere Aufwendungen erforderlich, um das Unpassende passend zu machen, falls das überhaupt möglich ist. Um einen solchen Fall überhaupt erkennen zu können, sind aber die Kontrollen zusammen mit den Hierarchien der beteiligten Strukturbäume von entscheidender Bedeutung.

Hierzu ist zu sagen:

- Jede Methode hat einen begrenzten Radius. Wendet man sie auf ein dafür ungeeignetes Problem an, oder überfordert man sich intellektuell, so kann kein brauchbares Ergebnis entstehen. Das Einführen von Denkverboten oder unnötiger Einschränkungen hat dagegen den Charakter der Sabotage und der Quälerei.
- Ist es im vorliegenden Fall nicht möglich, die Kontrollen wenigstens verbal zutreffend zu beschreiben, so handelt es sich möglicherweise um ein – mit dieser Methode oder im aktuellen Kontext ihrer Verwendung oder generell – unlösbares Problem. Falls die erforderlichen Entscheidungen zwar beschrieben, aber nicht sofort – anhand des dann vorliegenden Datenmaterials – getroffen werden können, bietet die JSP-Methode zwei Lösungswege an, nämlich das Vorauslesen und das Backtracking. Auch diese Techniken haben eine begrenzte Reichweite, aber damit lassen sich schon ziemlich harte Nüsse knacken. Es ist aber auch möglich, dass eine andere Betrachtung der Datenstrukturen oder die Zerlegung des Problems in Teilprobleme zu einem guten Ergebnis führen, das ganz ohne solche Techniken auskommt.
- Mit einiger Erfahrung gewinnen die Kontrollen einen Signalcharakter. Problemlos umsetzbare Kontrollen geben Sicherheit. Schwierige Kontrollen machen auf Komplikationen aufmerksam, die näher zu untersuchen immer lohnt. Das Risiko, etwas Wichtiges zu vergessen, sinkt durch das gewissenhafte Beschreiben der Kontrollen, denn Luftbuchungen werden dadurch rasch deutlich. Diejenigen Operationen, die nur Daten von A nach B bewegen, sind ja relativ unwichtig im Vergleich zu denjenigen, die das Material für die

Steuerung liefern. Gibt es dieses Material nicht, oder hat es nicht die gewünschten Eigenschaften, so sind die Probleme evident.

2.5.6 Sonderfall Leerzeile

Zurück zum Beispiel. In den Vorgaben steht:

- Leerzeilen enthalten 0 Zeichen, also weder Leerzeichen noch Textzeichen.

Das erlaubt es uns, Leerzeilen einfach als String-Spezialfall zu behandeln: Jeder Eingabezeile ist entweder eine Leerzeile oder eine Textzeile. Eine Leerzeile besteht aus 0 Zeichen. I2 ist dann sofort erfüllt. Andernfalls müsste im Diagramm zwischen Leerzeilen und gefüllten Zeilen unterschieden werden, was bei dieser Aufgabenstellung aber nicht nötig ist.

2.5.7 Unbemerkte Implikationen

Öfters ist es ebenso aufschlussreich, zu studieren, was ein Diagramm *nicht* aussagt, obwohl das Nichtgesagte auch wichtig sein könnte. Hier fällt unter anderem folgendes auf:

- Die physische Speicherungsform der Eingabe wird nicht genannt. Das Gedicht könnte in einer Datei, einer Datenbank, einer Tabellenkalkulations-Matrix, oder einer Interprozessnachricht enthalten, Zeichen für Zeichen an einem Eingabegerät eingetippt, oder auf einem Lochstreifen abgelocht sein. Für die zu lösende Programmieraufgabe ist das aber zunächst nebensächlich, denn die Datenstruktur der Eingabe ist davon nicht betroffen. Zu Beginn haben wir uns zwar auf eine Datei als Behälter für den Text festgelegt, aber das ist für die Struktogramme irrelevant.
- Wir haben uns daran gewöhnt, dass Texte aus Zeichen bestehen, die in Zeichensätzen organisiert sind, welche auf Byte-Inhalten aufbauen, zum Beispiel ASCII oder EBCDIC. Ein ASCII-Leerzeichen und ein EBCDIC-Leerzeichen werden nicht durch dieselbe Bitkombination eines Byte repräsentiert, und die anderen Zeichen tun das erst recht nicht.
- Ausserdem gibt es viele byteorientierte Zeichensatztabellen, die sich nur teilweise aufeinander abbilden lassen. Manche Zeichen, vor allem die diakritischen (beispielsweise â, é, ö), existieren nur in bestimmten Zeichensätzen (etwa im ASCII-Zeichensatz 850). Chinesische oder japanische Zeichensätze der Symbolschrift (Kanji) kommen jedoch nicht mit einzelnen Bytes aus, weil es viele tausend Kanji[6] gibt. Ihre Abbildung in die maschinellen Zeichen-Äquivalente ist deshalb aufwendiger als die simple Regel „ein Byte – ein Zeichen". Auch hierzu äußert sich die Eingabedatenstruktur nicht.
- Theoretisch ist die Aufgabe zwar zeichensatzunabhängig, praktisch gehen die Datenstrukturen aber davon aus, dass Zeichen um Zeichen gleich viel Platz belegt – insbesondere bei der Ausgabeformatierung, also eine Äquidistanzschrift vorliegt. Insofern blendet das Struktogramm einen durchaus relevanten Aspekt völlig aus – aber dies ist schließlich nur ein einfaches Beispiel.
- Irgendwie muss bekannt sein, wie lang jede Textzeile ohne schleppende Leerzeichen ist. Wäre das nicht der Fall, so müsste die Zeilenstruktur das widerspiegeln. Bei einer Direkteingabe oder einem Lochstreifen reicht zum Beispiel das CR-Zeichen (CR = Carriage Return = Wagenrücklauf) als Delimiter – aber auf diesen speziellen Aspekt geht das Struktogramm ebenfalls nicht ein.
- Es ist nicht klar, ob die Zeichenketten von links nach rechts oder von rechts nach links durchsucht werden sollen. Kulturell tendieren viele Menschen zur ersten Folge, aber das ist nicht selbstverständlich. Wenn schleppende Leerzeichen vorhanden sind, die bei der Zählung ignoriert werden sollen, liegt es nahe, von rechts nach links zu arbeiten – zumindest für die Suche nach dem rechten Ende der Zeichenkette, falls die

String-Länge nicht beim Lesen mitgeliefert wird. Eine sequentielle Eingabe Zeichen für Zeichen von links nach rechts erlaubt solch eine Rückwärtssuche natürlich nicht, es sei denn, die Zeichenketten würden intern zwischengespeichert.

2.5.8 Vom konkreten Ausgabebeispiel zur abstrakten Ausgabestruktur

Die „Abbildung 2.3: Ableitung der Ausgabe-Datenstruktur“ auf Seite 30 zeigt die Beziehungen zwischen den Ausgabezeilen und ihren abstrakten Entsprechungen im Ausgabe-Struktogramm. Auch hier ist leicht zu erkennen, wie die konkreten Dateninhalte der Beispiel-Ausgabe generalisiert und den Strukturblöcken eindeutig zugeordnet wurden. Die Ähnlichkeit der Eingabe- und Ausgabe-Struktogramme ist keineswegs selbstverständlich, sondern entspringt den gemeinsamen Formatvorgaben, sowie der aufgabenspezifischen Sicht, die beiden Modellen zugrundeliegt. Wäre die Eingabe ein Bitstrom, der in Paketen übermittelt wird, die Ausgabe aber eine Microsoft® Excel®[7]-Matrix im herstellerspezifischen Format, so wären beide Diagramme extrem verschieden und eine direkte Aufgabenlösung nicht möglich.

Die Sicht auf die Daten ist also nur in engen Grenzen frei wählbar, die durch die physische Beschaffenheit dieser Daten bestimmt sind. „Physisch“ bedeutet: Hardware, Betriebs-System-Software, sowie die sprachspezifischen Compiler- und Laufzeitroutinen definieren die bearbeitbaren Konstrukte. Diese maschinellen Komponenten werden zusammen als Basismaschine bezeichnet. Es gibt also viele verschiedene Basismaschinen, an deren Eigenschaften sich die Programmierung jeweils zwingend orientieren muss.

Auch das Ausgabe-Struktogramm thematisiert bestimmte Punkte nicht. So ist weder ausgesagt, ob zu Beginn ein Seitenvorschub stattfinden soll noch ist erkennbar, wie die Ausrichtung der Zahlenkolonnen bewirkt wird. Eingeschobene Leerzeichen zwischen dem Text und den Zahlen sind nicht modelliert. „input string“ reicht bis zum Beginn der Zahlenkolonnen, obwohl eingabeseitig eine variable Länge möglich ist.

Je nach Programmiersprache ist das kein relevantes Thema, weil die Ausgabeformatierung – zum Beispiel in COBOL – automatisch stattfindet, oder es muss tatsächlich jedes Distanz-Leerzeichen im Programm erzeugt werden. Im zweiten Fall ist das Ausgabe-Struktogramm entsprechend zu erweitern, was Sie als kleine Übungsaufgabe selbst machen sollten.

Die Abfolge nicht gekennzeichneter Strukturblöcke von links nach rechts ist als Sequentialität zu lesen. Was ist damit gemeint?

- „header“, „body“ und „footer“ folgen in der Ausgabe nacheinander: Zuerst wird der „header“ ausgegeben. Ihm folgen die Sätze, die den „body“ ausmachen. Danach kommt der „footer“. Nach ihm kommt in dieser Datei aus Anwendungssicht nichts mehr.
- „input string“ steht links von „string counters“. Hier spiegelt das Diagramm zunächst lediglich die optische Anordnung der Ausgabeelemente innerhalb der Body-Zeilen wieder. Das erscheint als eine kulturell bedingte grafische Konvention.

  Programmiertechnisch besteht bei Sprachen wie COBOL tatsächlich kein Zwang, zuerst den eingelesenen String links in der Ausgabezeile zu plazieren und dann die Zählerwerte rechts davon abzulegen. Das Programm könnte genausogut zuerst die Zählerwerte für die Ausgabe formatieren, ohne dass sich am Aussehen der Ausgabe dadurch etwas ändern würde.

  Bei anderen Programmiersprachen entsteht jede Ausgabezeile sequentiell durch Aneinanderreihung der auszugebenden Zeichen oder Zeichenketten. Will man bei der Modellierung sprachneutral bleiben, dann müssen die ersten Ausgabe-Zeichen(ketten) links von den ihnen folgenden gezeichnet werden.
- Zwingende Sequentialität liegt also vor, wenn erst alle Vorgänger eines bestimmten Datenelements passiert werden müssen, bevor es verarbeitet werden kann, oder wenn beim Aufbauen einer Ausgabe eine bestimmte Reihenfolge einzuhalten ist. Kann man dagegen wahlfrei vorgehen, dann spiegelt sich in einer Sequenz lediglich eine sinnvolle Aufreihung wieder.

Im Rechner gibt es zudem kein Rechts oder Links, sondern nur auf- oder absteigende Adressfolgen. So wird die Datendeklaration in einem Programm üblicherweise die rechts stehenden Datenelemente in einer von oben nach unten verlaufenden Listendarstellung mit steigenden Adressen aneinanderreihen, während hierarchisch im Struktogramm weiter unten stehende Bestandteile als Schachtelungen von Einschüben innerhalb dieser Listendarstellung erscheinen. Das gilt für zusammenhängende Gebilde, die Sätze, Zeichenketten, Tabellen oder Matrizen etcetera genannt werden. Die grafisch plausible Zwei- oder Mehrdimensionalität dieser Datenstrukturen wird so auf die Eindimensionalität der Speicherzellenfolge abgebildet.

Die Grafik verknüpft ein konkretes Ausgabebeispiel mit der daraus abgeleiteten Datenstruktur der Ausgabe:

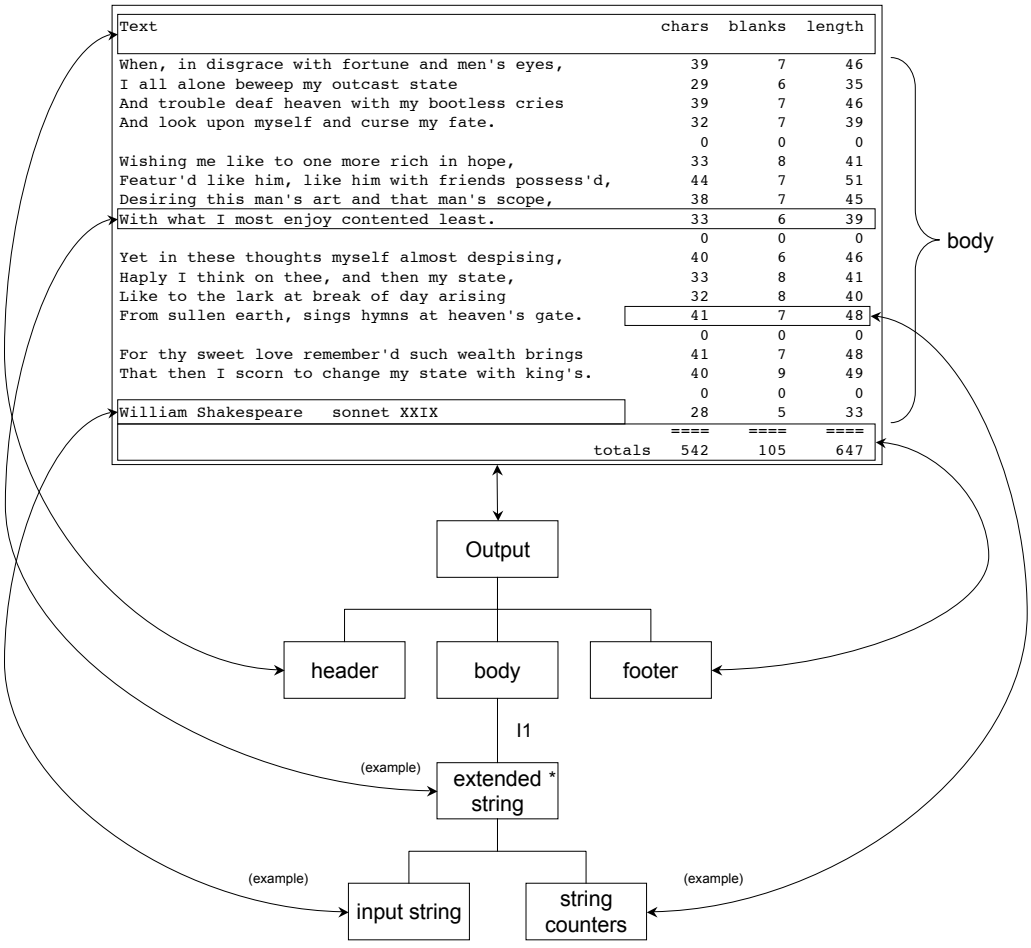


Abbildung 2.3: *Ableitung der Ausgabe-Datenstruktur*

Hinweis: Die Kontrolle I1 wurde vom Eingabe-Struktogramm übernommen, weil der „body" endet, wenn alle Eingabesätze verarbeitet wurden.

2.5.9 Inkompatible Strukturblöcke

Im Ausgabe-Struktogramm sind Sequenz-Elemente enthalten. Diese Strukturblöcke tragen – wie bereits beschrieben – keine Kennzeichnung. „*" und „o" und dürfen nicht gemeinsam mit Sequenz-Elementen und auch nicht beide zusammen im einem gegebenen Unterbaum auftreten. Diagramme wie das folgende sind deshalb nicht erlaubt:

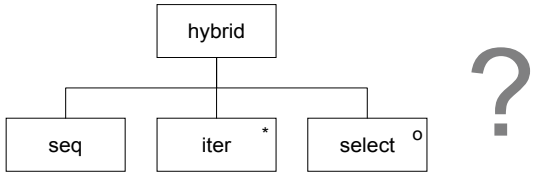


Abbildung 2.4: *Unzulässige Kombinationen von Strukturblöcken*

2.6 Korrespondenzen zwischen den Datenstrukturen ermitteln

Die Gemeinsamkeiten zwischen den Daten-Struktogrammen sind dafür entscheidend, ob daraus direkt die Programmstruktur abgeleitet werden kann. Ebenso interessant sind aber auch die Unterschiede, das heißt die nicht zusammenpassenden Elemente. Diese müssen an geeigneter Stelle in der künftigen Programmstruktur berücksichtigt werden.

Der zweite Schritt nach dem Zeichnen der Ein- und Ausgabestruktogramme besteht darin, diese einander gegenüberzustellen und nach Gemeinsamkeiten zu suchen, die Jackson „Korrespondenzen" nannte. Diese werden durch Verbindungslinien[8] markiert, wie die folgende Grafik zeigt.

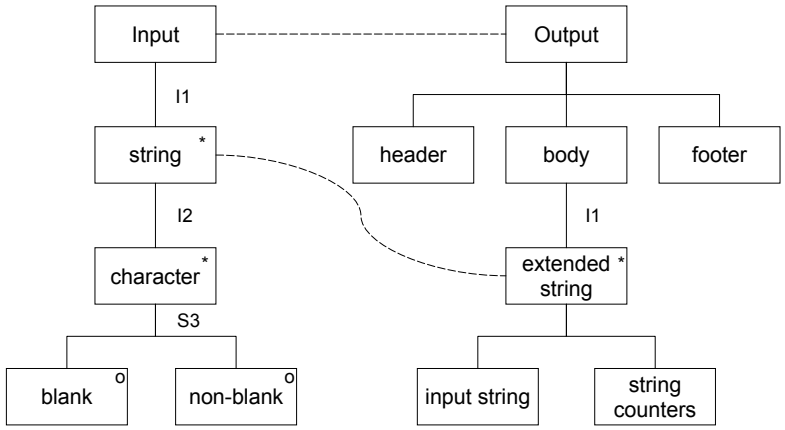


Abbildung 2.5: *Aufsuchen und Markieren der Korrespondenzen*

Hier sind nur sehr wenige Korrespondenzen zu finden, aber sie reichen aus.

Die gestrichelten Verbindungslinien zwischen den korrespondierenden Strukturblöcken sind deshalb so wichtig, weil die Programmstruktur im dritten Schritt aus den Datenstrukturen anhand der Korrespondenzen abgeleitet wird. Was ist eine Korrespondenz denn genau?

Zwei Strukturblöcke korrespondieren, wenn sie in Anzahl und Reihenfolge übereinstimmen.

Das sehen wir hier bestätigt:

- Die Eingabedatei wird in die Ausgabedatei transformiert.
- Jede Eingabezeile wird in eine Ausgabezeile umgesetzt.
- Die Reihenfolge der Eingabezeilen ist mit der Ausgabe-Reihenfolge identisch.

Da die Differenzierung der Eingabezeichen in Leerzeichen und Textzeichen sich nicht in der Ausgabestruktur widerspiegelt, kann dafür auch keine Korrespondenz angegeben werden. Es ist aber unschwer möglich, sich vorzustellen, wie aus der Einteilung der Eingabezeichen in die beiden Klassen die ausgabeseitigen Zählerwerte entstehen. Nichts fällt hier vom Himmel.

Allgemeiner ausgedrückt, handelt es sich bei den Korrespondenzen um eine Form der Synchronisation. Nur wenn Ein- und Ausgabe überall da vollständig synchronisiert sind, wo es auf die gleichzeitige Verarbeitbarkeit ankommt, kann Code geschrieben werden, der die Eingabe korrekt in die Ausgabe transformiert. Bei mehreren Eingabe- oder Ausgabe-Datenstrukturen gilt das analog.

2.7 Programmstruktur ohne Operationen ableiten

Daher können die beiden ***Daten***strukturen zur ***Programm***struktur verschmolzen werden. Das zeigt die Grafik auf der nächsten Seite.

Die korrespondierenden Strukturblöcke wurden dabei zusammengefasst. So entstanden die beiden Strukturblöcke der Programmstruktur, auf die von links und von rechts je ein Pfeil weist. Die anderen Strukturblöcke wurden in die Programmstruktur hierarchisch unverändert übernommen, wobei „Eingabe vor Ausgabe“ gilt. Der formale Strukturblock „process input string“ kam hinzu, um die Einheitlichkeits-Regel für Sequenzen einzuhalten. Ausserdem wurden die Kontrollen dort plaziert, wo sie entsprechend ihren Herkunftsstrukturen hingehören. Nur bei Strukturblöcken, die durch Verschmelzung entstehen, müssen die Beschreibungen der Kontrollen gegebenenfalls angepasst werden, ohne ihren Sinn zu ändern. Das war hier aber nicht nötig.

Es fällt auf, dass die Bezeichnungen der – abstrakten – Datenelemente beziehungsweise Datengruppen in Aktivitäten umgeformt wurden, die auf diese Daten wirken. Dabei kann man es sich sehr einfach machen und zum Beispiel bei Eingabedaten überall „verarbeiten“ oder „process“ oder „verarb.“ oder – noch kürzer – einfach „v.“ oder „pr.“ ergänzen, aber es lohnt sich, die Aktivitäten knapp und zutreffend mit anderen Worten zu kennzeichnen, weil das bei der Zuordnung der Operationen hilft.

Nun liegt bereits der Aufbau des Programms vor, der aber noch keine Operationen mit den Daten enthält. Er heißt daher „Programmstruktur ohne Operationen“ und ist nur ein Zwischenprodukt, auf das man bei einiger Übung auch öfters verzichten kann – ein Skelett ohne Muskeln.

Dennoch darf dieser Schritt keinesfalls unterschätzt werden. Er stellt neben den Struktogrammen die zweite zentrale Idee der Methode dar: Anstatt ein Programm zu erfinden, wird sein Aufbau aus einleuchtend aufgebauten Datenstrukturen abgeleitet, also *konstruiert*.

Gelingt es nicht, hinreichend viele echte Korrespondenzen zu finden, oder widersprechen sich zwischendrin die Ordnungskriterien, so kann die Programmstruktur ohne Operationen nicht oder nicht sofort ermittelt werden. Die JSP-Methode bietet zwar einige Erweiterungen für hartnäckige Probleme an, aber sie kann auch keine Wunder vollbringen. Falsche Vorgaben oder fehlende Voraussetzungen sind durch noch so schöne Struktogramme nicht reparierbar.

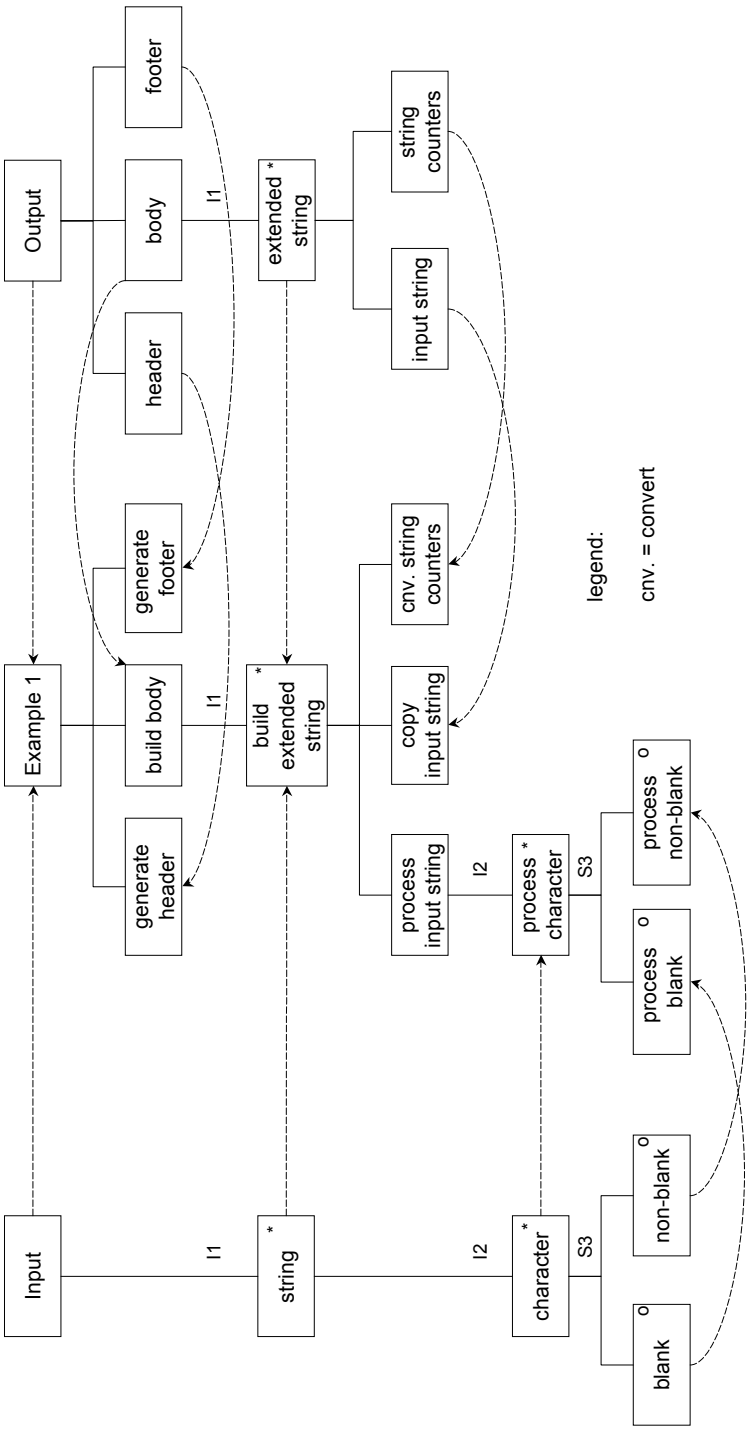


Abbildung 2.6: *Verschmelzung der korrespondierenden Datenstrukturen zur Programmstruktur*

2.8 Programmstruktur mit Operationen ergänzen

Die „Muskeln" werden hinzugefügt, indem die nötigen Operationen – also die Verrichtungen an den Daten – an den richtigen Stellen in zusätzlichen Strukturblöcken unter oder neben den vorhandenen Strukturblöcken hinzugefügt werden. Dabei müssen den Iterationen und Alternativen teilweise Sequenz-Elemente vor- und nachgestellt werden, wofür der Strukturbaum nach festen Regeln zu erweitern ist. Die neuen Strukturblöcke stehen natürlich ebenfalls in der Relation „besteht aus", das heißt, sie konkretisieren, was die Aktivitätsbezeichnungen in den abstrakten Strukturblöcken darüber eigentlich meinen.

Nach dem Einfügen der Operationskästchen und dem Einschieben des aus formalen Gründen erforderlichen zusätzlichen Strukturblocks „build extended strings" sieht die Programmstruktur zunächst so aus:

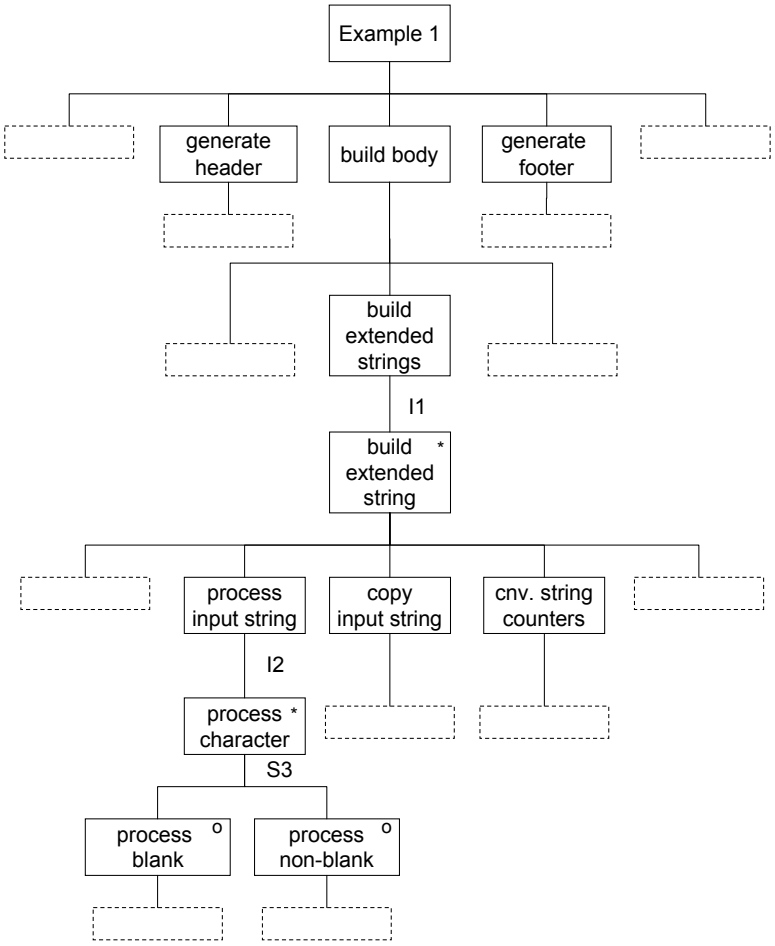


Abbildung 2.7: *Formale Ergänzungen der Programmstruktur*

Solche zusätzlichen Ebenen können zum Beispiel in den folgenden Fällen erforderlich werden:

- Vor einer Iteration ist im Allgemeinen eine Initialisierung erforderlich, die zum Beispiel einem Zähler einen Startwert zuweist. Das ist hier bei „build extended string" der Fall. Der neue Strukturblock darüber trägt die Bezeichnung „build extended strings", weil er die gesamte Schleife repräsentiert.

- Nach einer Iteration muss öfters das Ergebnis einer Summation oder ein erreichter Indexwert abgespeichert werden. Das ist hier zwar erforderlich, aber „process input string" ist bereits ein formaler Strukturblock, neben dem – rechts – solche Operationen plaziert werden können.
- Vor Alternativen ist es häufig nötig, vorhandene Speicherinhalte umzuformen, damit die Abfragen als boole'sche Ausdrücke ohne eingeschachtelte Operationen formuliert werden können. Es gibt zwar Programmiersprachen, die es erlauben, innerhalb von Abfragen Funktionen aufzurufen und Werte zuzuweisen, aber dies erfordert hohe Programmierdisziplin und stellt eine potente Fehlerquelle dar. Also sind die nötigen Umformungs-Operationen der jeweiligen Alternative sequentiell voranzustellen.
- Nach einer Alternative sind ab und zu gemeinsame Operationen anzuordnen, die sonst in den Operationskästchen zu den einzelnen Alternativfällen mehrfach rechts auftreten würden. Das nennt man Herausfaktorisieren. Auch dann ist ein weiterer Sequenz-Strukturblock nötig. Darauf wurde hier aber verzichtet.

Zusätzliche abstrakte Strukturblöcke in der Baumhierarchie sollten nahezu ebenso heißen, wie diejenigen unter ihnen, die aus der Programmstruktur ohne Operationen stammen. Bei Iterationen wird einfach aus dem Singular der Plural gemacht, während bei Alternativen eine passende Aktivitätsbezeichnung die Erweiterung zur Sequenz ausdrückt.

Nun ist die Liste der Operationen zu erstellen. Diese sieht wie folgt aus:

Op.	Bedeutung
1	open input file
2	open output file
3	close input file
4	close output file
5	stop
6	read input record = in_string
7	print „Text ..." line
8	print empty line
9	total_chars := 0
10	total_blanks := 0
11	total_len := 0
12	chars := 0
13	blanks := 0
14	len := LENGTH(in_string)
15	ind := 1
16	ind := ind + 1
17	chars := chars + 1
18	blanks := blanks + 1
19	total_chars := total_chars + chars
20	total_blanks := total_blanks + blanks
21	total_len := total_len + len
22	copy in_string to output area
23	convert string counters to counter columns
24	print copied in_string & counter columns
25	convert total counters to counter columns
26	print === line
27	print „totals" & counter columns

Die Zuweisungen sind in ALGOL-Syntax geschrieben: Links vom Zuweisungszeichen „:=“ steht die Empfänger-Variable, rechts der zuzuweisende Wert. ALGOL = ALGOrithmic Language ist eine Sprache der dritten Generation, die zum Vorläufer vieler ähnlich strukturierter Sprachen wie zum Beispiel C wurde und hauptsächlich zur Lösung numerischer Aufgaben diente. Zeiger-, Zeichen- und String-Operationen fehlten dagegen in der ersten Version dieser Sprache. Dagegen waren und sind feldorientierte Anweisungen die Domäne von COBOL, dessen Sprachmittel die Numerik nur ziemlich umständlich unterstützen, und das kein so stringentes Unterprogrammkonzept wie ALGOL besitzt. Diese Einseitigkeiten haben sich bis heute in vielen Nachfolgesprachen gehalten. Auch mit der Verschiebung zu den objektorientierten Sprachen hin hat sich daran nichts gebessert. Nach wie vor ist in den modernen Programmiersprachen die Eleganz von COBOL beim Umgang mit Feldern und Formaten unerreicht.

Zurück zur Liste der Operationen. Meist ist sie nicht so ordentlich wie hier aufgebaut, weil des Öfteren einige Operationen vergessen und daher später hinzugefügt, andere als überflüssig erkannt und wieder gestrichen, oder Operationsnummern mit neuen Bedeutungen belegt werden (Vorsicht!).

Die Operationen in der Liste sind durchnumeriert, was nicht unbedingt lückenlos, aber immer eindeutig geschehen muss. In den folgenden Diagrammen sind diese Nummern in den Operationskästchen plaziert, wobei die mit Kommata aufgebauten Listen als Sequenzen zu lesen sind: Zuerst findet die Operation ganz links statt, dann die nach dem Komma rechts folgende, dann die übernächste nach rechts und so weiter. Bei einfachen Operationen ohne Abhängigkeit von vorhergehenden Operationen im selben Kästchen ist die Reihenfolge natürlich beliebig. Als Ordnungskriterium könnte dann beispielsweise die Aufeinanderfolge der Strukturelemente der Ein- oder Ausgabe dienen, weil das die Richtigkeitskontrolle erleichtert.

Beim Plazieren der Operationen kann es geschehen, dass Zuordnungen unklar oder widersprüchlich erscheinen und sich scheinbar nur durch Tricks – wie etwa das Hinzufügen von Schaltern und Schalter-Abfragen – Lösungen finden lassen. Ebenso ist es möglich, dass für unbedingt erforderliche Operationen kein geeigneter Platz in der Programmstruktur zu existieren scheint. Das sind sichere Zeichen dafür, dass im bisherigen Entwurfsprozess etwas schiefgelaufen ist.

Ein typischer Fall ist der, dass dem ersten Auftreten eines Strukturelements keine besondere Bedeutung beigemessen wurde und nun der Unterschied zur Behandlung weiterer solcher Strukturelemente unangenehm auffällt. Im vorliegenden Beispiel ist die Komplexität aber so gering, dass die Plazierung der Operationen ohne Schwierigkeiten stattfindet.

Zu Anfang empfiehlt es sich, die Operationen zum Öffnen, Schließen, Lesen und Stoppen einzutragen:

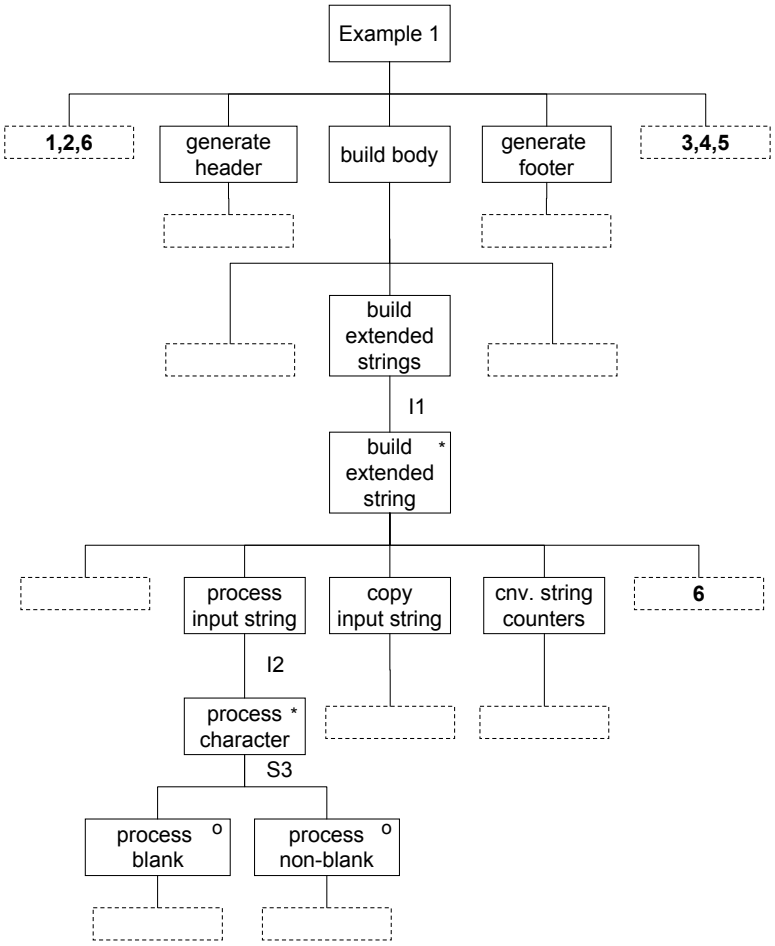


Abbildung 2.8: *Plazierung der grundlegenden Operationen*

Bei den Operationen 1, 2, 3, 4 und 5 erfordert die Plazierung kein großes Nachdenken. Dagegen verwundert zunächst die Anordnung der Operation 6 an zwei Orten.

Die Vorlese- und Nachlese-Operationen sind nämlich Jackson-typisch angeordnet:

- Das Vorlesen nach dem Öffnen der Eingabedatei stellt den ersten Input String bereit, oder es setzt den EOF-Indikator (siehe weiter unten zu I1). Des öfteren muss eine leere Eingabe mit dem Abbruch des Laufs quittiert werden. Daher ist es wichtig, dass das Vorlesen ausserhalb derjenigen Schleife erfolgt, welche die Eingabedaten verarbeitet. Nicht selten sind jedoch Schleifen anzutreffen, deren Endebedingung vor dem ersten Durchlauf – selbstverständlich – nicht erfüllt sein darf, und in denen sämtliche Leseoperationen stattfinden.

  Der Nachteil solcher Konstruktionen ist, das sie bei leerer Eingabe das Übergehen der Verarbeitung für die normalerweise vorhandenen Daten und die Beendigung der Schleife erzwingen müssen, was häufig mehr oder minder unglücklich mit geschachtelten Abfragen, Flags, Break-Anweisungen oder gar expliziten Sprüngen geschieht.
- Das Nachlesen ist dort zu plazieren, wo die zuvor gelesenen Daten verbraucht sind und neue Daten benötigt werden. Das ist im vorliegenden Beispiel spätestens dann nötig, wenn die Ausgabe zu einer

Eingabezeile vollständig erzeugt wurde. Die früheste Nachlese-Position wäre unmittelbar nach „process input string".

Es passiert durchaus mal, dass man das Nachlesen vergisst oder an falscher Stelle anordnet. Dann kann eine unendliche Schleife entstehen. Zumindest wird das Programm nicht das erwartete Verhalten zeigen.

Nun können die – bezüglich des eingelesenen Strings lokalen – Operationen 12 bis 18 zugeordnet werden, wobei 12 bis 15 nur Initialisierungen beziehungsweise Zuweisungen sind, die vor „process input string" stattfinden müssen und damit auf derselben Hierarchieebene stehen.

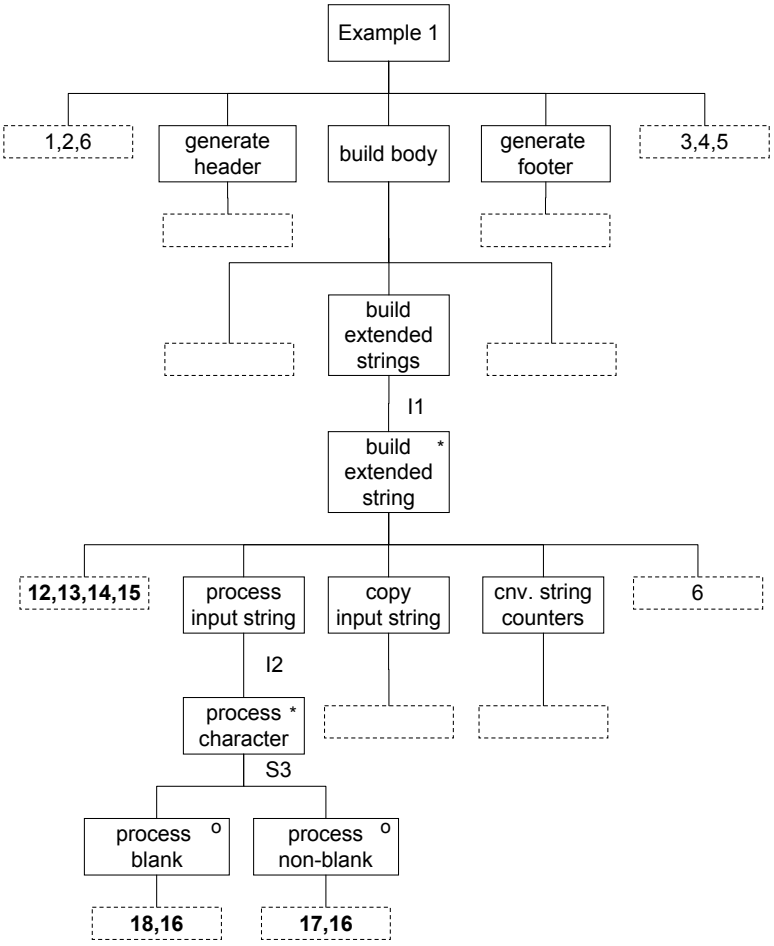


Abbildung 2.9: *Plazierung elementarer Initialisierungs- und Inkrementierungsoperationen*

Mit den Operationen 18 und 17 werden – abhängig vom Ausgang der Abfrage S3 – die Zeichen (Blank | kein Blank) gezählt. Die Operation 16 schaltet den Zeichenindex weiter. Das ist ein Zeiger auf das jeweils nächste auszuwertende Zeichen, der mit 1 initialisiert wird. Zur Realisierung von S3 folgt weiter unten ein Abschnitt.

Auch hier wird wieder das Vorlesen und Nachlesen verwendet, allerdings auf Zeichenebene. Der Sinn ist derselbe: Bereitstellen eines auswertbaren Datenelements beziehungsweise einer Datengruppe oder Setzen einer Endebedingung, sowie Fortschalten auf das nächste Datenelement oder auf die nächste Datengruppe nach der Auswertung. Die Endebedingung I2 muss erfüllt sein, wenn der Zeichenindex auf die erste Position rechts vom Eingabestring zeigt.

Nun können die Operationen 9, 10, 11, 19, 20 und 21 plaziert werden. 9, 10 und 11 sind wieder Initialisierungen, die vor dem Eintritt in die Hauptschleife „build extended strings" stattfinden müssen. Mit 19, 20 und 21 werden die lokalen Ergebnisse aufkumuliert. Sie müssen daher nach „process input string" auf derselben Ebene stehen. Dafür ist ein Einschub erforderlich, weil diese Operationen nicht „copy input string" zugeordnet werden können:

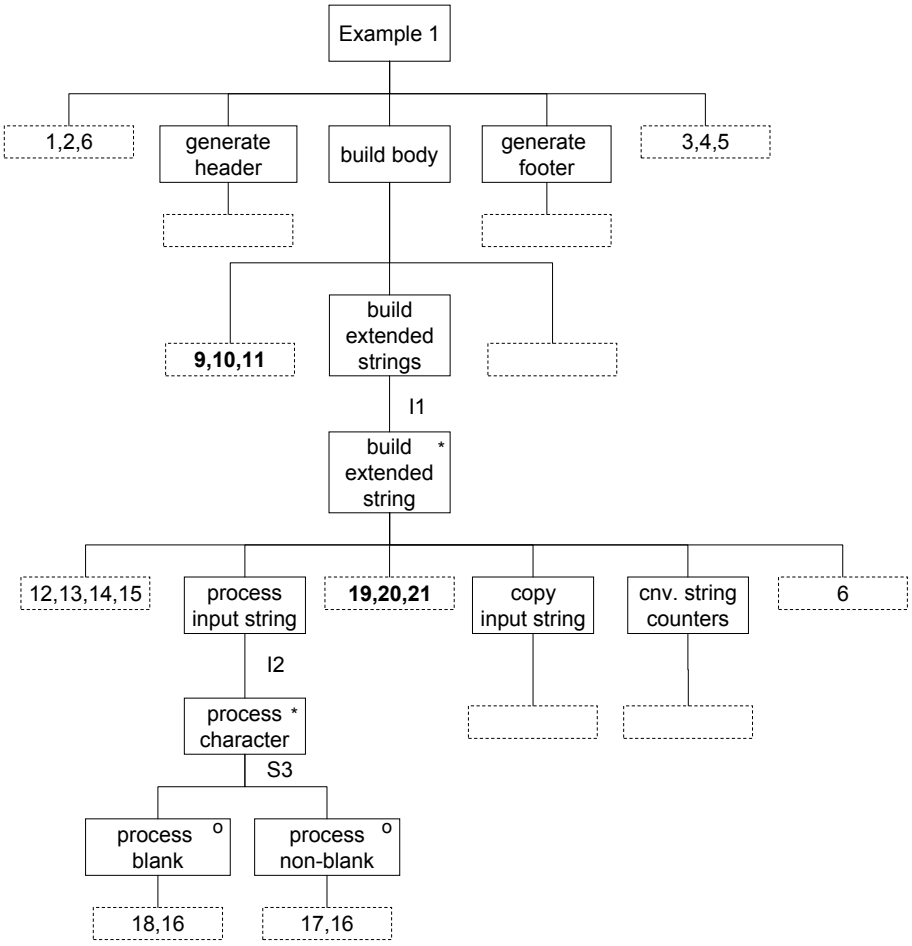


Abbildung 2.10: *Plazierung von Initialisierungs- und Kumulierungsoperationen*

Hier fällt auf, dass die Operationen 9, 10 und 11 ebensogut an den Anfang der Verarbeitung gestellt werden könnten, da sie diesem sequentiell folgen. In späteren Beispielen werden solche Initialisierungen im Allgemeinen solchen Operationen wie 1,2 und 6 vorangestellt, weil sie häufig zusammen mit dem Anlegen der programmweit definierten Datenelemente stattfinden. Variablen mit kleinerem Gültigkeitsbereich, also lokale Variablen, werden dagegen im Moment des Anlegens fast immer zugleich initialisiert. Das Zusammentreffen von Deklaration und Initialisierung muss kommentiert werden, damit die Umsetzung der Programmstruktur mit Operationen in den jeweiligen Programmcode vollständig überprüfbar ist.

De facto wird ein Compiler solche Operationen wie 9, 10 und 11 entweder in Vorbelegungen der beim Modul-Start – von einer Systemkomponente – zu ladenden Speichersegmente oder in initiale Zuweisungen vor dem Aktivieren des eigentlichen Programmcodes umsetzen, denn der Inhalt des Variablenbereichs eines (Unter-)Programms ist ohne explizite Startwertvorgaben in vielen Sprachen undefiniert.

Die Ausgabeoperationen 7 und 8, sowie 22 bis 27 fehlen noch. Sie werden jetzt ergänzt:

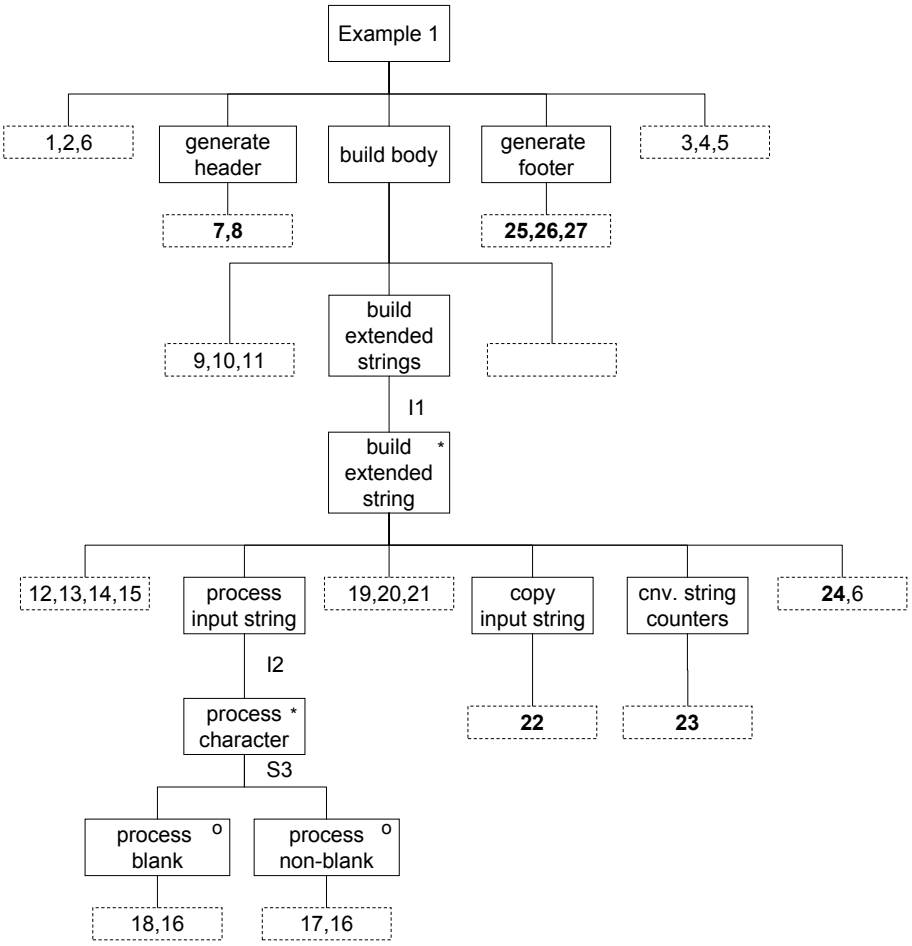


Abbildung 2.11: *Plazierung der Ausgabeoperationen*

Nur die Operationen 22 bis 24 gehören zur Einzel-String-Ebene. Sie werden so oft durchlaufen, wie es solche Strings gibt. Die anderen Operationen sind dagegen einleitend (7 und 8) oder abschließend (25, 26 und 27) anzuordnen.

Hier lohnt es sich, ganz genau hinzusehen. Sind wirklich alle Voraussetzungen für diese Operationen vorhanden? Nur das kann ausgegeben werden, was auch eingelesen beziehungsweise speicherintern aufgebaut wurde. Es ist durchaus möglich, dass an dieser Stelle noch Versäumnisse erkannt werden. Im einfachsten Fall wurden lediglich Operationen vergessen, die nachträglich definiert und an den richtigen Stellen plaziert werden müssen. Es ist aber auch möglich, dass Fehler im Entwurf erst jetzt offenbar werden.

In solchen Fällen ist von improvisierten Lösungen dringend abzuraten, obwohl vielleicht die Zeit drängt und der Aufwand zur Überprüfung der bisherigen Entwurfsschritte als unnötig erscheint. Warum nicht – so kurz vor dem Ziel! – rasch einige Schalter definieren und mit einigen genial plazierten Zuweisungen oder Abfragen die Probleme aus der Welt schaffen? Damit wird jedoch meist alles nur noch schlimmer. Der klare Weg des Entwurfs auf der Basis der Datenstrukturen und der erkannten Korrespondenzen wird verlassen. Stattdessen gelangen „Methoden" der Spaghettiprogrammierung zur Anwendung, die das bisherige Vorgehen unterlaufen, neue und undurchsichtige Abhängigkeiten schaffen, sowie spätere Überprüfungen und Änderungen erschweren.

Schlimmer noch: Echte Entwurfsfehler werden verschleiert und schein-beseitigt, anstatt sie konsequent aufzudecken und zu beheben. Der immer anzuratende Weg der Überprüfung von Anfang an führt womöglich zur Entdeckung zentraler Irrtümer, die den „Quick and Dirty"-Scheinlösungen immer anhaften und sich irgendwann rächen werden.

Das leere Kästchen rechts neben „build extended strings" stört noch. Wurde etwas vergessen? Dort müssten Operationen nach dem Schleifenende eingetragen werden, die etwa Schleifenergebnisse auswerten. Da diese Aufgabe bereits von den Ausgabe-Operationen übernommen wird, gibt es hier nichts zu tun. Das Kästchen kann gelöscht werden:

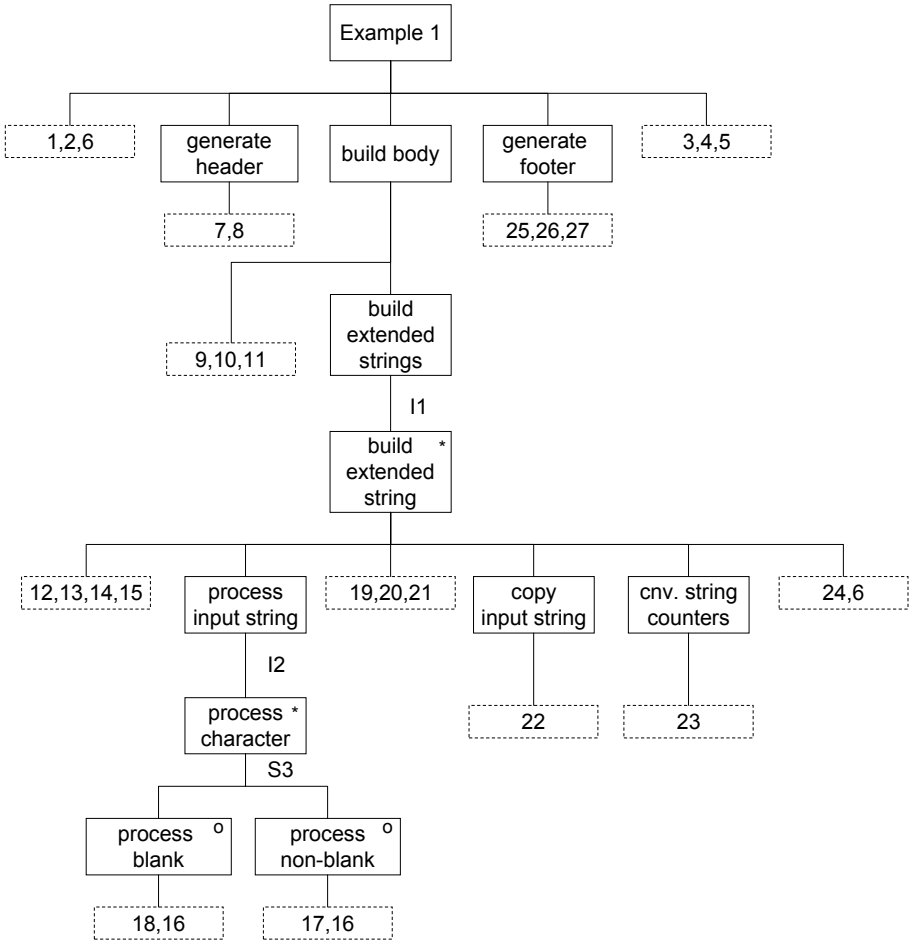


Abbildung 2.12: *Fertige Programmstruktur mit Operationen*

Damit kommt man der algorithmischen Implementierung schon ganz nahe.

Ein Hinweis zur Zeichentechnik: Die Operationen sind in gestrichelt umrandeten Kästchen aufgezählt, was hier nur aus optischen Gründen geschah. Beim Zeichnen von Hand werden die Kästchenränder durchgezogen oder es wird sogar nur die obere Linie gezeichnet. Da häufig Operationen ergänzt werden müssen, weil etwas vergessen wurde, nervt das ständige Wegradieren und erneute Zeichnen der Konturen ziemlich. Ausserdem sind die Kästchen meistens verschieden groß, was die Zeichnung unruhig macht. Daher ist es am besten, ein-

fach zunächst jeweils die obere Linie zu ziehen und die Verbindung zum Baum herzustellen. Ob die Kästchen später geschlossen werden, ist Geschmackssache.

Das kann dann so aussehen:

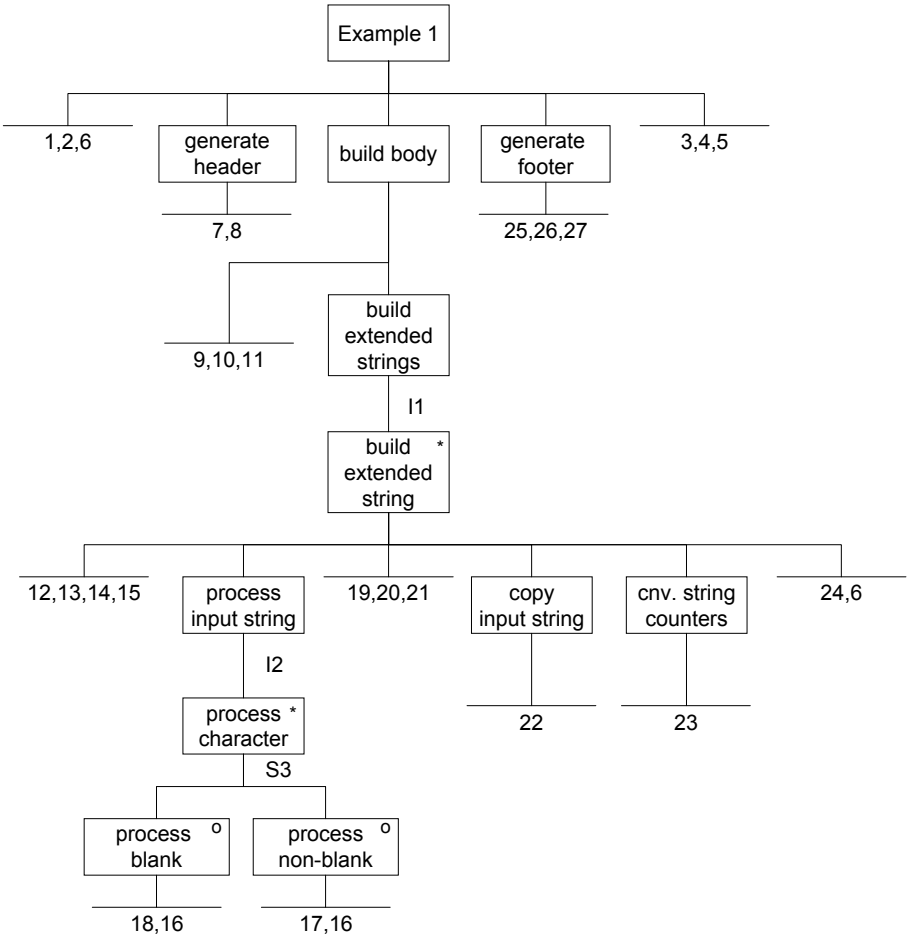


Abbildung 2.13: *Alternative Darstellung für Handzeichnungen*

Einige abschließende Hinweise zur Plazierung der Operationskästchen und zur Kontrolle:

- Operationskästchen können problemlos wie gezeigt angehängt oder eingeschoben werden, solange dies in einer Sequenzstruktur geschieht. Wenn sie unten an einen vorhandenen Strukturblock angehängt werden, sollten darin nur diejenigen Operationen aufgeführt sein, die zu dem betreffenden abstrakten Datenelement beziehungsweise der abstrakten Datengruppe gehören.
- Alle anderen Operationen sollten der Klarheit wegen in separaten Kästchen angeordnet werden. Der etwas höhere Zeichenaufwand wird dadurch belohnt, dass Fehlplazierungen solcher Operationen schneller und sicherer erkannt werden können. Die Versuchung, hier frei erfundene Dinge hinzuzufügen, ist ab und zu groß. Das muss nicht schlecht sein, aber es sollte verdeutlicht werden. Problematisch wird es natürlich, wenn so versucht wird, das methodische Vorgehen zu unterlaufen und letztlich eine Spaghettiprogrammierung hübsch einzukleiden.

Die Stellvertretung der z.T. relativ langen Operationsbeschreibungen durch die Nummern spart zwar viel Platz in den Kästchen, ist aber fehleranfällig. Daher empfiehlt es sich, diese Operationslisten anhand der Programmstruktur mit Operationen sorgfältig zu kontrollieren:

- Sind alle in der Liste definierten Operationen auch im Struktogramm vorhanden?
- Wurden Operationen vergessen? Wird etwas ohne vorherige Zuweisung verwendet?
- Sind die Voraussetzungen für jede dieser Operationen durch vorhergehende Operationen erfüllt beziehungsweise am jeweiligen Platz in Struktogramm zugesichert? Stimmt die Folge?
- Wurde jede Operation an der Stelle plaziert, wo das zugehörige abstrakte Element der Ein- oder Ausgabe sie benötigt? Jede abstrakte Datengruppe und jedes Datenelement im Struktogramm benötigen ja nicht nur Entsprechungen in der realen Datenwelt, sondern auch Operationen, welche diese Daten(gruppen) handzuhaben ermöglichen.

2.9 Nachbemerkungen zu den Operationen

In der JSP-Literatur findet sich auch der Begriff „elementare Operationen", verbunden mit der Forderung, die Modellierung solle keine Implementierung vorwegnehmen. Bei der Definition der Operationen müsse demnach höchstmögliche Allgemeinheit walten, ohne Rücksicht auf die – meist doch bereits bekannte – Zielsprache. Was ist davon zu halten?

Nun ja, Autoren sollten ihre Meinungen den Leser(inne)n nicht aufdrängen, aber auch nicht so subtil verstekken, dass diese die Manipulation nicht bemerken. Ab und zu muss aber – so wie hier – ganz deutlich gemacht werden, was für ein Krampf solche Ideen sind. Hier stehen sich doch Anforderungen an die Modellierung diametral gegenüber:

- Was letztlich als „elementar" zu bezeichnen ist, lässt sich nicht eindeutig definieren. Dem einen gilt nur ein Assemblerbefehl als elementar, auch wenn dieser eine Matrixmultiplikation per Mikrocode implementiert, dem anderen reicht ein so komplexes Konstrukt wie ein Datenbankzugriff. Bei großer Ratlosigkeit darf es auch gern ein „intelligentes Zugriffsmodul" sein, in dem das Wunder geschieht, das die Leserin / der Leser der Operationsliste gern genau verstanden hätte.
- Es ist zwar wünschenswert, sich bei der Definition der Operationen nicht allzu eng an die Zielsprache anzulehnen, aber diese bildet nun einmal das Substrat, aus dem alles Höhere erschaffen wird. Entfernt man sich zu sehr von der Zielsprache – nach „unten" durch Zerlegung ins Atomare hinein oder nach „oben" durch großzügiges Postulieren darin nicht implementierter Funktionen -, so wird die Modellierung willkürlich. Es ist also ein guter Kompromiss gefragt, der ein hinreichendes Abstraktionsniveau mit der erforderlichen Deutlichkeit verbindet. Gerade die Formulierung der Operationen in einer neutralen Art und Weise, wie sie hier verwendet wird, garantiert sowohl die Verständlichkeit als auch die Portabilität beim Übergang auf andere Zielsprachen.
- Wenn die Zielsprache sehr atomar ist, also zum Beispiel eine Assemblersprache, oder wenn sie komplexe Konstrukte auf einfache Weise zu nutzen erlaubt, also möglicherweise eine Sprache der sogenannten vierten Generation, kann es sinnvoll sein, die Operationen wie im Beispiel gezeigt zu formulieren, solange der Bezug zur geplanten Implementierung gewahrt bleibt. Eine Operationsliste aus Assemblerbefehlen wird enorm lang und nur für Experten lesbar sein. Umgekehrt kann bei 4GL-Sprachen die Liste auch aus Operationen bestehen, die in einem Komplexitätsgrad-Missverhältnis zueinander stehen.
- Worte wie „elementar" sind gefährlich. Sie lenken die Aufmerksamkeit bei der Modellierung auf Nebengleise der politischen Korrektheit. Ähnlich schlimme Wirkungen hat die willkürliche Unterscheidung in technische und fachliche Sachverhalte bei der Systemanalyse. Was technisch und was fachlich ist, lässt sich nicht sauber voneinander abgrenzen. Hier wie dort wirkt das Verdikt, die spätere Implementierung nicht bei der

Modellierung vorwegzunehmen. Im Ergebnis entstehen oft genug Luftschlösser, die zwar prima aussehen, sich aber nicht umsetzen lassen.

- Fazit: Lösen Sie sich vom Denken in geistigen Zwangsjacken, die von irgendwelchen selbsternannten Gurus erfunden wurden. Was methodisch sauber strukturiert und technisch korrekt implementiert ist, funktioniert – und allein darauf kommt es an.

2.10 Kontrollen überprüfen und präzisieren

Nun ist es an der Zeit, die Kontrollen anhand der Operationsliste noch einmal genau zu betrachten. Bislang lautet die kurze Liste der Kontrollen wie folgt:

I1 end of input file
I2 end of string
S3 character is blank

Da jetzt die Operationen bekannt sind, ist zu fragen, ob die Operationsergebnisse ausreichen, um die Kontrollen zu programmieren. Wie sieht das hier aus?

i1	Die Operation 6 muss zusätzlich einen EOF-Indikator setzen, wenn die Basismaschine signalisiert, dass kein weiterer Eingabesatz mehr zur Verfügung steht. I1 muss also diesen Indikator abfragen. Der Indikator soll zum Beispiel das Zeichen `'D'` (= „Data“) enthalten, wenn ein Satz eingelesen wurde, sonst das Zeichen `'E'` (=„End“). Somit lautet I1 nun: EOF_indicator = `'E'`
I2	Die fiktive Funktion LENGTH soll die Länge „len“ der Zeichenkette im Satz ohne nachfolgende Blanks zurückliefern, wie auch immer das funktionsintern programmiert wird. Eine Leerzeile soll das Ergebnis 0 (Null) liefern. Da der Zeichen-Zeiger „ind“ am Anfang der Verarbeitung eines Strings auf 1 gesetzt wird und somit auf das erste Zeichen ganz links zeigt, muss die Schleife mit dem Endekriterium I2 bei Leerzeilen sofort enden, bei gefüllten Zeilen aber erst nach Abarbeitung des letzten Zeichens ganz rechts. Nach Prüfung eines Zeichens wird „ind“ um 1 erhöht. Also ist I2 wie folgt zu programmieren: ind > len
S3	Je nach Programmiersprache kann ein String als ein Feld (= Array) aus aufeinanderfolgenden Zeichen aufgefasst oder mit Hilfe von Zeichenfunktionen inspiziert werden. In COBOL gibt es hierfür ausserdem den Reference Modifier. Daher werden hier drei algorithmische Lösungen angeboten, um die Zeichenkette namens „in_string“ zu untersuchen. Der Unterstrich ist in COBOL übrigens unzulässig; stattdessen ist der Bindestrich zu verwenden. a) Feldelement-Abfrage in_string_byte[ind] = `' '` mit „in_string_byte“ als Element des Arrays „in_string“ und `' '` = Leerzeichen. Ist diese Bedingung erfüllt, wird der linke Zweig gewählt, sonst der rechte. b) Zeichenfunktion SUBSTR(in_string, ind, 1) = `' '` mit SUBSTR als Funktion mit den Argumenten (str, pos, len), die eine Zeichenkette aus dem String „str“ ab der Position „pos“ in der Länge „len“ ausschneidet und als Funktionsergebnis abliefert. c) Reference Modifier in-string(ind:1) = `' '` mit (pos:len) analog zu SUBSTR

Die konkretisierten Kontrollen lauten daher wie folgt:

I1 EOF_indicator = 'E'
I2 ind > len

und eine der folgenden Varianten (oder etwas Ähnliches je nach Zielsprache):

S3 in_string_byte[ind] = ' '
S3 SUBSTR(in_string, ind, 1) = ' '
S3 in-string(ind:1) = ' '

2.11 Ausgabe-Lösungsalternative

Die in den vorangehenden Unterkapiteln gezeigte Lösung für das erste JSP-Beispiel stammt aus der Mitte der 70er Jahre und wurde vom Autor erstmals in einem Seminar zur Programmentwicklung auf dem TR 440 vorgestellt. Danach gingen etliche Jahre ins Land, in denen kein Anlass bestand, das JSP-Beispiel an neue Möglichkeiten anzupassen – bis die Spreadsheet-Programme aufkamen und die Nutzung eines persönlichen Rechners zur Normalität wurde. Nun war es möglich, mit Sprachen wie Visual Basic® auch Datenmengen zu bearbeiten und damit die traditionelle – auch die JSP-basierte – Programmierung beiseite zu schieben. Das kann natürlich nicht die Absicht dieses Kapitels sein. Vielmehr geht es darum, eine Alternative für die Formatierung der Ausgabe vorzustellen, die erstens das Programm davon entlastet, die Ausrichtung der Ergebnisspalten selbst vornehmen zu müssen, und zweitens den Fließtext in einer anderen Schriftart als Courier – als Beispiel für einen Zeichensatz mit fester Zeichenbreite – darstellen zu können. Dafür eignen sich die CSV-Dateien[9] (csv = comma separated values) als Importformat, zum Beispiel für die Funktion Calc in Apache OpenOffice®[10]. Im konkreten Fall sähe die Ausgabe mit Semikolon als Trenner wie auf der nächsten Seite gezeigt aus.

```
Text;;chars;blanks;length
;;;;
When, in disgrace with fortune and men's eyes,;;39;7;46
I all alone beweep my outcast state;;29;6;35
And trouble deaf heaven with my bootless cries;;39;7;46
And look upon myself and curse my fate.;;32;7;39
;;0;0;0
Wishing me like to one more rich in hope,;;33;8;41
Featur'd like him, like him with friends possess'd,;;44;7;51
Desiring this man's art and that man's scope,;;38;7;45
With what I most enjoy contented least.;;33;6;39
;;0;0;0
Yet in these thoughts myself almost despising,;;40;6;46
Haply I think on thee, and then my state,;;33;8;41
Like to the lark at break of day arising;;32;8;40
From sullen earth, sings hymns at heaven's gate.;;41;7;48
;;0;0;0
For thy sweet love remember'd such wealth brings;;41;7;48
That then I scorn to change my state with king's.;;40;9;49
;;0;0;0
William Shakespeare   sonnet XXIX;;28;5;33
;;====;====;====
;totals;542;105;647
```

Das erscheint auf den ersten Blick als ziemlich fremdartig. Nun, wenn das Programm anstelle einer vollständig aufbereiteten Textdatei mit dem Suffix `'.txt'` eine Ausgabe wie die obige mit dem Suffix `'.csv'` erzeugen könnte, dann würde ihr Inhalt beispielsweise nach einem Doppelklick auf das Dateisymbol vom Spreadsheet-Programm Excel – und nach etwas Formatierungsarbeit – auf dem Bildschirm wie folgt angezeigt:

	A	B	C	D	E
1	Text		chars	blanks	length
2					
3	When, in disgrace with fortune and men's eyes,		39	7	46
4	I all alone beweep my outcast state		29	6	35
5	And trouble deaf heaven with my bootless cries		39	7	46
6	And look upon myself and curse my fate.		32	7	39
7			0	0	0
8	Wishing me like to one more rich in hope,		33	8	41
9	Featur'd like him, like him with friends possess'd,		44	7	51
10	Desiring this man's art and that man's scope,		38	7	45
11	With what I most enjoy contented least.		33	6	39
12			0	0	0
13	Yet in these thoughts myself almost despising,		40	6	46
14	Haply I think on thee, and then my state,		33	8	41
15	Like to the lark at break of day arising		32	8	40
16	From sullen earth, sings hymns at heaven's gate.		41	7	48
17			0	0	0
18	For thy sweet love remember'd such wealth brings		41	7	48
19	That then I scorn to change my state with king's.		40	9	49
20			0	0	0
21	William Shakespeare sonnet XXIX		28	5	33
22			====	====	====
23		totals	542	105	647

Abbildung 2.14: *Formatierte Excel-Darstellung des Laufergebnisses*

Die Spalten A und B, sowie die erste Zeile sind mit der Schriftart Times und der Schriftgröße 12 Pt formatiert, der Rest der Tabelle mit der Schriftart Verdana® und 10 Pt. Ohne die Zeilen- und Spaltennamen und ohne Raster erhält man beim Drucken beziehungsweise nach der Umwandlung in ein PDF®[11]-Dokument folgendes Ergebnis:

Text	chars	blanks	length
When, in disgrace with fortune and men's eyes,	39	7	46
I all alone beweep my outcast state	29	6	35
And trouble deaf heaven with my bootless cries	39	7	46
And look upon myself and curse my fate.	32	7	39
	0	0	0
Wishing me like to one more rich in hope,	33	8	41
Featur'd like him, like him with friends possess'd,	44	7	51
Desiring this man's art and that man's scope,	38	7	45
With what I most enjoy contented least.	33	6	39
	0	0	0
Yet in these thoughts myself almost despising,	40	6	46
Haply I think on thee, and then my state,	33	8	41
Like to the lark at break of day arising	32	8	40
From sullen earth, sings hymns at heaven's gate.	41	7	48
	0	0	0
For thy sweet love remember'd such wealth brings	41	7	48
That then I scorn to change my state with king's.	40	9	49
	0	0	0
William Shakespeare sonnet XXIX	28	5	33
	====	====	====
totals	542	105	647

Abbildung 2.15: *Ergebnis im PDF-Format ohne störende Grafikelemente*

Das sieht doch schon weit besser aus…

Die Semikolon-Zeichen werden anstelle der im Text enthaltenen Kommata als Spalten-Separatoren verwendet. Es gibt auch einen Kopfdatensatz, der bis auf die Spalte B die Spaltennamen definiert. Für fünf Spalten sind vier Separatoren erforderlich und in jedem Ausgabesatz vorhanden. Somit entspricht die Datei der Norm RFC 4180.

Anpassung des bisherigen Entwurfs

Was muss an dem bisherigen Entwurfsergebnis verändert werden, um die gewünschte CSV-Datei anstelle der bisherigen Textdatei zu erzeugen? Wie wir uns rasch überzeugen können, sind die Strukturen der Ein- und Ausgabe und somit die Korrespondenzen unverändert geblieben. Der einzige relevante Unterschied liegt darin, dass das Programm die Druckaufbereitung nicht mehr selbst vornimmt, sondern für die zu erzeugenden Weissräume jeweils Separatoren einsetzt. Das lässt sich leicht durch Ergänzung beziehungsweise durch geringfügige Änderung der Liste der Operationen bewerkstelligen. Dies und die anschliessende Erstellung einer neuen Programmversion anhand der vorhandenen Lösungen im Band 2-A1 – siehe auch die folgenden Implementierungs-Vorbereitungskapitel im Band 1-A1 – könnte für Sie eine gute Übung sein.

Endnoten

1 **Kalenderberechnung**

[…]

Die Berechnung von Kalendern (arithmetische Kalender) setzt umfangreiche astronomische und mathematische Kenntnisse voraus. Bei der Entwicklung des frühen ägyptischen astralen Sothiskalenders waren diese Kenntnisse vorhanden. Die Einführung eines ägyptischen Verwaltungskalenders auf 365-Tage-Basis folgte spätestens im dritten Jahrtausend v. Chr. Dieser konnte jedoch das Durchwandern der Jahreszeiten nicht verhindern. Die ägyptischen Könige bemängelten die Jahreszeitenverschiebung, doch erst Ptolemaios III. unternahm 238 v. Chr. einen Versuch zur Einführung eines Schalttages. Nach seinem Tod wurde neben dem neuen Schalttageskalender jedoch wieder der alte ägyptische Verwaltungskalender benutzt. Der julianische Kalender, der 45 v. Chr. von Julius Cäsar eingeführt wurde, stützte sich gleichwohl auf die Kalenderform des Ptolemaios III.

[Wikipedia-Seitentitel: Kalender
Datum der letzten Bearbeitung: 25. April 2022, 17:07 UTC
Versions-ID der Seite: 222362914
Permanentlink: https://de.wikipedia.org/w/index.php?title=Kalender&oldid=222362914
Datum des Abrufs: 13. Juni 2022, 16:20 UTC]

2 Der **gregorianische Kalender**, auch **bürgerlicher Kalender**, ist der weltweit meistgebrauchte Kalender. Er entstand gegen Ende des 16. Jahrhunderts durch eine Reform des julianischen Kalenders. Benannt ist er nach Papst Gregor XIII., der ihn 1582 mit der päpstlichen Bulle Inter gravissimas verordnete. Der gregorianische ist wie schon der julianische Kalender ein Sonnenkalender (Solarkalender) mit einer gegenüber dem julianischen Kalender verbesserten Schaltjahresregelung durch Interkalation. Damit liegt ihm eine durchschnittliche Jahreslänge von 365,2425 Tagen zugrunde, die den etwa 365,2422 Tagen des Sonnenjahres (Tropisches Jahr) näherkommt als noch die 365,25 Tage des julianischen Kalenders. Der gregorianische Kalender löste im Laufe der Zeit sowohl den julianischen als auch zahlreiche andere Kalender ab. Auf dem gregorianischen Kalender beruht auch die Datumsdarstellung nach ISO 8601.

Der Zweck der gregorianischen Kalenderreform bestand darin, ein weiteres Auseinanderdriften von Kalender- und Sonnenjahr zu verhindern und beide besser zu synchronisieren.

[Wikipedia-Seitentitel: Gregorianischer Kalender
Datum der letzten Bearbeitung: 6. Juni 2022, 21:16 UTC
Versions-ID der Seite: 223492325
Permanentlink: https://de.wikipedia.org/w/index.php?title=Gregorianischer_Kalender&oldid=223492325
Datum des Abrufs: 13. Juni 2022, 16:22 UTC]

3 Der Erbauer des Planetenhauses war Georg Crossmann (* um 1550; † 1612 in Lemgo), ein deutscher Bildhauer und Baumeister der Weserrenaissance aus Lemgo: siehe Wikipedia-Artikel zu ihm. Der Sonnenkönig in der Mitte des Giebels ist von den personifizierten Planeten umgeben. Links und rechts sind zwei angekettete Löwen zu sehen. Diese höchst originelle Darstellung ohne jeglichen christlichen Bezug lässt das Altertum wiederauferstehen. „Renaissance" bedeutet ja „Wiedergeburt".

Der Bildband „Lemgo" von Heinrich Gräfenstein erschien 1990 im Gieseking-Verlag, Bielefeld. Es beschreibt das Planetenhaus auf den Seiten 51 bis 53.

4 Dieses Sonett liegt in verschiedenen Schreibungen vor. Die hier gezeigte Variante verwendet der besseren Verständlichkeit halber nur wenige Worte, die seit langem ausser Gebrauch sind (zum Beispiel „thy").

5 Die 1:1-Übersetzung „Selektion" erinnert an die Nazi-Vernichtungslager und wird daher in dieser Buchreihe nicht verwendet. Das Wort „Auswahl" ist auch schwierig; so wirkt beispielsweise die Formulierung „eine Auswahl mit mehr als 2 S-Elementen" ziemlich merkwürdig. Daher wird stattdessen das Wort „Alternative" verwendet, das nicht mit „selection" rückübersetzt werden kann.

6 **Kanji** […] sind die in der japanischen Schrifttradition verwendeten Schriftzeichen chinesischen Ursprungs.

Gesamtzahl der Kanji

Die tatsächliche Anzahl der Kanji ist eine Auslegungssache. Das Daikanwa Jiten ist mit seinen rund 50.000 Schriftzeichen nahezu umfassend. Es gibt allerdings historische wie auch neuere chinesische Wörterbücher, die über 80.000 Schriftzeichen enthalten. Von vielen Schriftzeichen sind dabei seltene historische Varianten aufgeführt und einzeln gezählt. Für den Alltag ausreichende Schriftzeichenwörterbücher enthalten zwischen 4.400 (NTC's New Japanese-English Character Dictionary) und 13.000 (Super Daijirin) Schriftzeichen.

[Wikipedia-Seitentitel: Kanji
Datum der letzten Bearbeitung: 27. Mai 2022, 21:34 UTC
Versions-ID der Seite: 223217938
Permanentlink: https://de.wikipedia.org/w/index.php?title=Kanji&oldid=223217938
Datum des Abrufs: 13. Juni 2022, 16:32 UTC

7 **Microsoft Excel** (abgekürzt MS Excel) ([…]) ist das am weitesten verbreitete Tabellenkalkulationsprogramm.

[…]

Wie die meisten Tabellenkalkulationen ermöglicht Excel umfangreiche Berechnungen mit Formeln und Funktionen, unter anderem mit kaufmännischen, statistischen und Datumsfunktionen. Excel besitzt auch zahlreiche mathematische Funktionen, so dass viele Probleme der Wirtschaftsmathematik berechnet werden können. Es können Texte verkettet oder logische Berechnungen (wenn…dann) durchgeführt werden. Abhängig von Inhalten und Werten in der Tabelle kann auf Inhalte an anderer Stelle der Tabellen zugegriffen werden. Die Ergebnisse können mit Hilfe von Sortier-, Gruppier- und Filterfunktionen sowie Pivot-Tabellen ausgewertet und in Diagrammen grafisch dargestellt werden. Tabellen oder Teile davon können gegen Layout- oder Inhaltsänderungen geschützt werden.

[Wikipedia-Seitentitel: Microsoft Excel
Datum der letzten Bearbeitung: 4. April 2022, 08:21 UTC
Versions-ID der Seite: 221769502
Permanentlink: https://de.wikipedia.org/w/index.php?title=Microsoft_Excel&oldid=221769502
Datum des Abrufs: 13. Juni 2022, 17:05 UTC]

8 Alternativ können die Korrespondenzen auch durch parallele Nummerierungen der korrespondierenden Elemente markiert werden. Diese Form wird später vorgestellt.

9 Das Dateiformat **CSV** steht für englisch Comma-separated values (seltener Character-separated values) und beschreibt den Aufbau einer Textdatei zur Speicherung oder zum Austausch einfach strukturierter Daten. Die Dateinamenserweiterung lautet .csv.

Gelegentlich wird für die Trennung der Datenfelder das Tabulatorzeichen verwendet. Diese Variante nennt man TSV (englisch Tab-separated values).

Ein allgemeiner Standard für das Dateiformat CSV existiert nicht, jedoch wird es im RFC 4180 grundlegend beschrieben. Die zu verwendende Zeichenkodierung ist ebenso wenig festgelegt; 7-Bit-ASCII-Code gilt weithin als der kleinste gemeinsame Nenner.

In CSV-Dateien können Tabellen oder eine Liste unterschiedlich langer Listen abgebildet werden.

[…]

Dateiaufbau

Innerhalb der Textdatei haben einige Zeichen eine Sonderfunktion zur Strukturierung der Daten.

• Ein Zeichen wird zur **Trennung** von Datensätzen benutzt. Dies ist in der Regel der Zeilenumbruch des dateierzeugenden Betriebssystems – bei dem Betriebssystem Windows® sind es in der Praxis oft tatsächlich zwei Zeichen.

• Ein Zeichen wird zur **Trennung** von Datenfeldern (**Spalten**) innerhalb der Datensätze benutzt. Allgemein wird dafür das Komma eingesetzt. Abhängig von beteiligter Software und Benutzereinstellungen sind auch Semikolon, Doppelpunkt, Tabulatorzeichen, Leerzeichen oder andere Zeichen üblich.

• Um Sonderzeichen innerhalb der Daten nutzen zu können (z. B. Komma in Dezimalzahlwerten), wird ein **Feldbegrenzerzeichen** (auch: **Textbegrenzungszeichen**) benutzt. Normalerweise ist dieser Feldbegrenzer das Anführungszeichen ". Wenn der Feldbegrenzer selbst in den Daten enthalten ist, wird dieser im Datenfeld verdoppelt […].

Der erste Datensatz kann ein Kopfdatensatz sein, der die Spaltennamen definiert.

Jeder Datensatz sollte laut RFC 4180, Absatz 2, Punkt 4 die gleiche Anzahl Spalten enthalten – dies wird aber nicht immer eingehalten.

[Wikipedia-Seitentitel: CSV (Dateiformat)
Datum der letzten Bearbeitung: 9. Januar 2022, 12:53 UTC
Versions-ID der Seite: 218964806
Permanentlink: https://de.wikipedia.org/w/index.php?title=CSV_(Dateiformat)&oldid=218964806
Datum des Abrufs: 13. Juni 2022, 17:13 UTC]

10 **Apache OpenOffice** (vormals **OpenOffice.org**) ist ein freies Office-Paket, das aus einer Kombination verschiedener Programme zur Textverarbeitung (Writer), Tabellenkalkulation (Calc), Präsentation (Impress) und zum Zeichnen (Draw) besteht. Ein Datenbankprogramm (Base) und ein Formeleditor (Math) sind ebenfalls enthalten. Es ist für alle wichtigen Betriebssysteme verfügbar. Das quelloffene Projekt war bis zur Abspaltung von LibreOffice eines der international führenden Office-Pakete, inzwischen hat LibreOffice diese Rolle aufgrund der größeren Entwicklerzahl und häufigeren Sicherheitsupdates übernommen.

[Wikipedia-Seitentitel: Apache OpenOffice
Datum der letzten Bearbeitung: 11. Juni 2022, 10:40 UTC
Versions-ID der Seite: 223613684
Permanentlink: https://de.wikipedia.org/w/index.php?title=Apache_OpenOffice&oldid=223613684
Datum des Abrufs: 13. Juni 2022, 17:19 UTC]

11 Das **Portable Document Format** (englisch; kurz **PDF**; deutsch: (trans)portables Dokumentenformat) ist ein plattformunabhängiges Dateiformat, das 1992 vom Unternehmen Adobe Inc.® entwickelt und veröffentlicht wurde und aktuell von der PDF Association® weiterentwickelt wird.

PDF ging aus dem 1991 von Adobe-Mitbegründer John Warnock initiierten "Project Camelot" hervor. Ziel war, ein Dateiformat für elektronische Schriftstücke zu schaffen, sodass diese unabhängig vom ursprünglichen Anwendungsprogramm, vom Betriebssystem oder von der Hardwareplattform originalgetreu wiedergegeben werden können. Das Ziel wurde erreicht und findet seinen Niederschlag in der ISO-Normenserie 32000 (ISO 15930 für PDF/X). Hierzu griff man wesentlich auf die Funktionsweise des PostScript-Formats zurück. Ein Leser einer PDF-Datei soll das Schriftstück immer in der Form betrachten und ausdrucken können, die der Autor festgelegt hat. Die typischen Konvertierungsprobleme (wie veränderter Seitenumbruch oder falsche Schriftarten) beim Austausch eines Schriftstückes zwischen verschiedenen Programmen entfallen dadurch.

Neben Text, Bildern und Grafik kann eine PDF-Datei auch Hilfen enthalten, die die Navigation innerhalb des Schriftstückes erleichtern. Dazu gehören zum Beispiel anklickbare Inhaltsverzeichnisse und miniaturisierte Seitenvorschauen.

[Wikipedia-Seitentitel: Portable Document Format
Datum der letzten Bearbeitung: 17. Mai 2022, 13:10 UTC
Versions-ID der Seite: 222952291
Permanentlink: https://de.wikipedia.org/w/index.php?title=Portable_Document_Format&oldid=222952291
Datum des Abrufs: 13. Juni 2022, 17:22 UTC]

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)

Algorithmische Implementierungsvorbereitungen

Das nebenstehende Bild zeigt die „ARITHMETICA" als Halbrelief an der Ratslaube des Lemgoer Rathauses[1]. Die Arithmetik gehört zum Quadrivium („Vierweg") der sieben freien Künste[2] seit der Antike, also zu den mathematisch ausgerichteten Studienfächern. Die anderen drei sind Geometrie, Musik (!) und Astronomie. Grammatik, Rhetorik und Dialektik bilden das Trivium („Dreiweg"). Ohne Arithmetik lässt sich in der Geometrie und der Astronomie wenig ausrichten. Ebenso verhält es sich mit der algorithmischen Programmierung, die auch dem Code in den Methoden der objektorientierten Programmierung zugrunde liegt.

Augusta Ada King-Noel, Countess of Lovelace, allgemein als Ada Lovelace[3] bekannt, erkannte bei der Mitarbeit an der von Charles Babbage[4] erfundenen „Analytical Engine" das Potential der algorithmischen Programmierung mittels Lochkarten[5], das weit über Babbage' ursprüngliche Idee eines Rechenautomaten hinausreicht. Sie schrieb darüber lt. Wikipedia folgendes in ihren „Notes":

> *»Die Grenzen der Arithmetik wurden in dem Augenblick überschritten, in dem die Idee zur Verwendung der [Programmier]Karten entstand, und die Analytical Engine hat keine Gemeinsamkeit mit schlichten Rechenmaschinen. Sie ist einmalig, und die Möglichkeiten, die sie andeutet, sind höchst interessant.«*

Die *Analytical Engine wurde zu Babbage' Lebzeiten nie fertiggestellt. Sie wäre auch auf dem damaligen Stand der Mechanik vermutlich nicht funktionsfähig gewesen. Heute dagegen verfügen wir über Universalcomputer, welche die vermutlichen Fähigkeiten der Analytical Engine in jeder Hinsicht weit übertreffen. Im folgenden Kapitel geht es um die Umsetzung des JSP-basierten Programmentwurfs in ein algorithmisches Programm, das auf einem Universalcomputer ausgeführt werden kann.*

Wie bereits angedeutet, können erfahrene Anwender der Methode das algorithmisch aufgebaute Programm unmittelbar aus der Programmstruktur mit Operationen ableiten, da diese ja schon alle erforderlichen Informationen enthält (wenn man die Listen der Kontrollen und Operationen mitzählt).

In bestimmten Fällen kann es dennoch sinnvoll sein, ein Flussdiagramm zu zeichnen oder sprachneutralen Code – Schematische Logik nach Jackson beziehungsweise einen anderen Pseudocode – zu schreiben, oder sogar beides zu tun. Ein typischer Anwendungsfall für dieses zunächst als umständlich und überflüssig erscheinende Vorgehen ist die Programminversion. Hierbei wird ein komplexes Abbildungsproblem in Teilprobleme zerlegt, die anschließend mit den JSP-Methoden gelöst und zum Beispiel mittels Unterprogrammtechnik implementiert werden. Die Programminversion ist die Königsdisziplin innerhalb der JSP-Methode, stellt also die höchsten Anforderungen an die Qualität des Entwurfs. Dann lohnt sich auch der Zusatzaufwand.

Für eine OO-Implementierung fehlen an dieser Stelle jedoch noch wesentliche Schritte, die im Kapitel „4 Objektorientierte Implementierungsvorbereitungen" auf Seite 83 ff diskutiert werden. Die COBOL-II- und C-Implementierungen von Example1 befinden sich in den Unterkapiteln „COBOL-II-Code von Example1" und „C-Code von Example1" des Band-2-A1-Kapitels „Example1 in COBOL II und C".

Endnoten: siehe Seite 79

3.1 Flussdiagramm erstellen

In einfachen Anwendungsfällen bedeutet ein Flussdiagramm verschwendete Zeit. Es wird hier daher nur gezeigt, um die Transformation der Programmstruktur mit Operationen zu veranschaulichen und die Brücke zu einem gewohnten Darstellungsmittel zu schlagen. Das Flussdiagramm zum Beispiel sieht wie folgt aus:

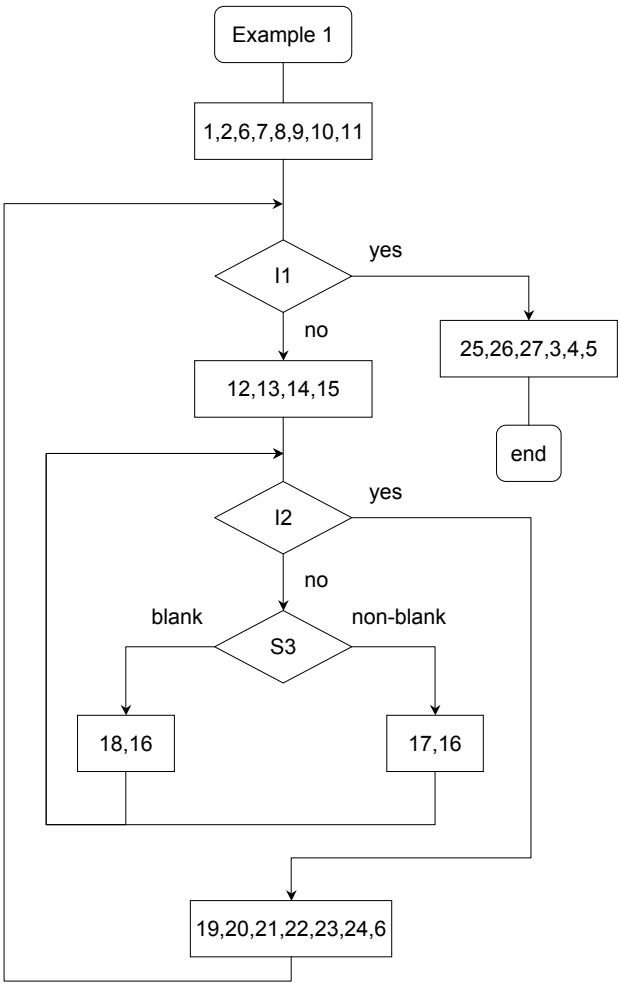


Abbildung 3.1: *Abgeleitetes Flussdiagramm des ersten Programmbeispiels*

Wie findet die Umsetzung aus der Programmstruktur mit Operationen ins Flussdiagramm statt?

a) Sequenzen

Alle sequentiellen Operationen aus den gestrichelten Kästchen werden zusammengefasst, wenn sie in der Relation „besteht aus" direkt aufeinanderfolgen. Befindet sich aber eine Iteration oder eine Alternative unter einem Folge-Kästchen, so unterbricht das die Zusammenfassung.

Beispiel: Die kurzen Sequenzen 1,2,6 und 7,8 und 9,10,11 werden zu 1,2,6,7,8,9,10,11 zusammengefasst. Danach folgt „build extended strings", das eine Schleife repräsentiert.

b) Iterationen

Wie bereits bei der ersten Vorstellung der Iterationen gesagt, repräsentieren diese die abweisende Schleife. Daher ist die Iterations-Ende-Abfrage als Raute oben anzuordnen, in der die jeweilige Kontrolle genannt wird. Bei nicht erfüllter Endebedingung wird der Schleifenkörper ausgeführt, der seinerseits eine innere Struktur aufweisen kann, beziehungsweise aus einer oder mehreren Operationen besteht. An Ende des Schleifenkörpers führt ein Pfeil zurück zur Endeabfrage.

Bei erfüllter Endebedingung wird die Verarbeitung mit dem sequentiell nächsten Schritt fortgeführt, der bei geschachtelten Schleifen – gegebenenfalls über einige Operationen – zur nächsthöheren Schleifenebene zurückführt, also zu deren Endeabfrage. Das zeigt die Grafik mit der Schachtelung der I1- und I2-Schleifen.

Die Kommentare „yes" und „no" an den Linien sind eigentlich überflüssig, da der Schleifenkörper in Flussdiagrammen, die aus Programmstrukturen abgeleitet wurden, immer nur bei *nicht* erfüllter Endebedingung durchlaufen wird. Sie könnten daher weggelassen werden.

c) Alternativen

Auch hier wird die Raute verwendet, um eine Abfrage mit einer oder zwei Verzweigungen darzustellen. Da die Raute viel Platz beim Zeichnen benötigt, kann stattdessen das flache Zeichen für „Programmmodifikation" verwendet werden (siehe „Abbildung 3.3: Flussdiagramm-Varianten für einfache und mehrfache Auswahl" auf Seite 57). Bei mehr als zwei Alternativen gibt es keine festgelegte Symbolik. Diese können als Abfolge einzelner Abfragen dargestellt werden, was schnell unübersichtlich wird, oder als Stapel. Dieser sollte immer mit demjenigen Fall enden, der übrigbleibt, wenn alle anderen Abfragen nicht zutreffen („default", „otherwise", …).

Wichtig ist dabei, dass alle Fall-Zweige nach unten auf eine gemeinsame Linie weitergeführt werden, an die entweder eine sequentiell folgende Struktur anschließt, oder die Rückkehr zur Schleifen-Endeabfrage.

Die „Abbildung 3.2: Äquivalenzen zwischen Strukturblöcken und Flussdiagramm-Elementen" auf Seite 56 und die „Abbildung 3.3: Flussdiagramm-Varianten für einfache und mehrfache Auswahl" auf Seite 57 fassen die Umsetzungen anhand von Beispielen zusammen.

Ergänzende Anmerkungen

Flussdiagramme, die aus JSP-Programmstrukturen abgeleitet wurden, sind immer kreuzungsfrei und enthalten keine Quersprünge. Jedes Konstrukt hat nur einen Eingang und nur einen Ausgang, so komplex es intern aufgebaut sein mag. Somit werden einige Probleme konsequent vermieden, die der Spaghettiprogrammierung – und den sie begleitenden Flussdiagrammen – zu Recht einen schlechten Ruf eingetragen haben.

Der Aufwand für Flussdiagramme ist allerdings relativ hoch. Änderungen, die in den Struktogrammen rasch vorgenommen werden können, erfordern möglicherweise ein mühseliges Um- oder Neuzeichnen von Flussdiagrammen. Ausserdem benötigen Flussdiagramme für denselben Informationsgehalt mehr Platz als Struktogramme, vor allem für die Verbindungslinien. Daher kann es leicht passieren, dass ein großes Struktogramm, das auf eine Seite passt, mehreren Seiten mit Flussdiagrammteilen entspricht, die durch Konnektoren zusammengehalten werden müssen, was den Zeichenaufwand und die Irrtumsmöglichkeiten gleichermaßen erhöht.

Daher sollten Flussdiagramme nur noch verwendet werden, um

- sich in die JSP-Methode einzuarbeiten und die ersten Programme damit zu konzipieren, oder
- in wirklich schwierigen Fällen als Zwischenschritt zu dienen. Für diese letzte dauerhaft verbleibende sinnvolle Verwendungsart werden weiter unten im Rahmen der Pseudocode-Darstellung nach Art der Block Definition Language Vorbereitungen getroffen.

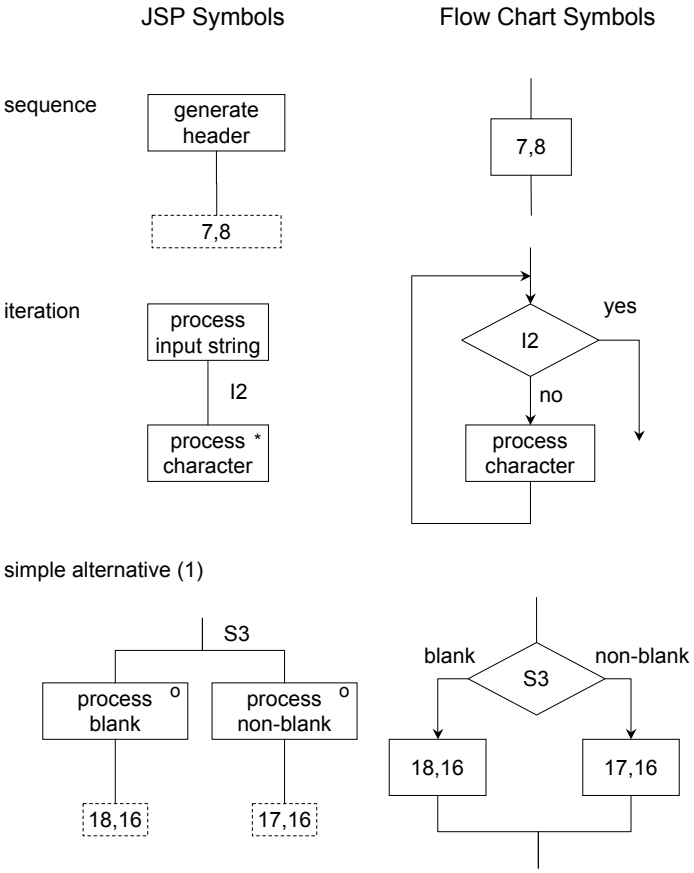


Abbildung 3.2: *Äquivalenzen zwischen Strukturblöcken und Flussdiagramm-Elementen*

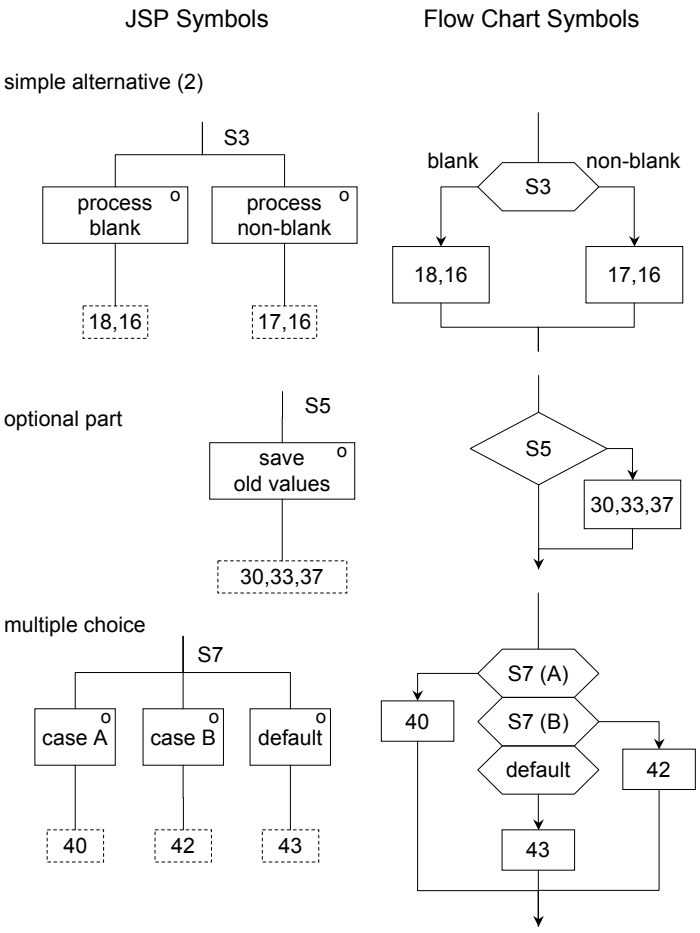


Abbildung 3.3: *Flussdiagramm-Varianten für einfache und mehrfache Auswahl*

3.2 Schematische Logik oder Pseudocode schreiben

Mit etwas Übung kann aus der Programmstruktur mit Operationen direkt die Implementierung abgeleitet werden. Da der Weg zum fertigen Programm hier vollständig erklärt werden soll, bleibt uns dieser Abkürzungsweg momentan versperrt. Der folgende Teilschritt wird von erfahrenen Anwendern der Methode relativ selten benötigt, aber in schwierigen Fällen ist er unabdingbar nötig. Solche Fälle sind zum Beispiel das Backtracking und die Programminversion, die in den Bänden 5 und 6 ausführlich vorgestellt werden.

An die Stelle der Operationsnummern treten in der Schematischen Logik die Operationen selbst, denn diese sollen ja anschließend in der Zielsprache formuliert werden. Das allein macht diesen sprachneutralen Code schon relativ unübersichtlich. Weiter unten wird daher – und aus weiteren Gründen – ein neuerer Pseudocode vorgestellt, bei dem die Operationskästchen sogar noch zusammengefasst werden können, um den Schreibaufwand zu senken und die Übersichtlichkeit zu verbessern.

3.3 Jacksons Schematische Logik

Die von Jackson vorgeschlagene Schematische Logik mutet aus heutiger Sicht altmodisch und umständlich an. Sie wird hier dennoch gezeigt, um interessierten Leser(inne)n die Möglichkeit zum Vergleich und zudem eine COBOL-nahe Alternative anzubieten.

Die elementaren Konstrukte Sequenz, Alternative und Iteration aus dem Struktogramm werden hier 1:1 in Code übertragen. Dafür gibt es zwei Möglichkeiten, die auch kombiniert werden können:

- die geschachtelte Darstellung, oder
- die flache Darstellung, also die Auflösung nach Struktogrammebenen.

Dabei gelten folgende Abbildungsregeln.

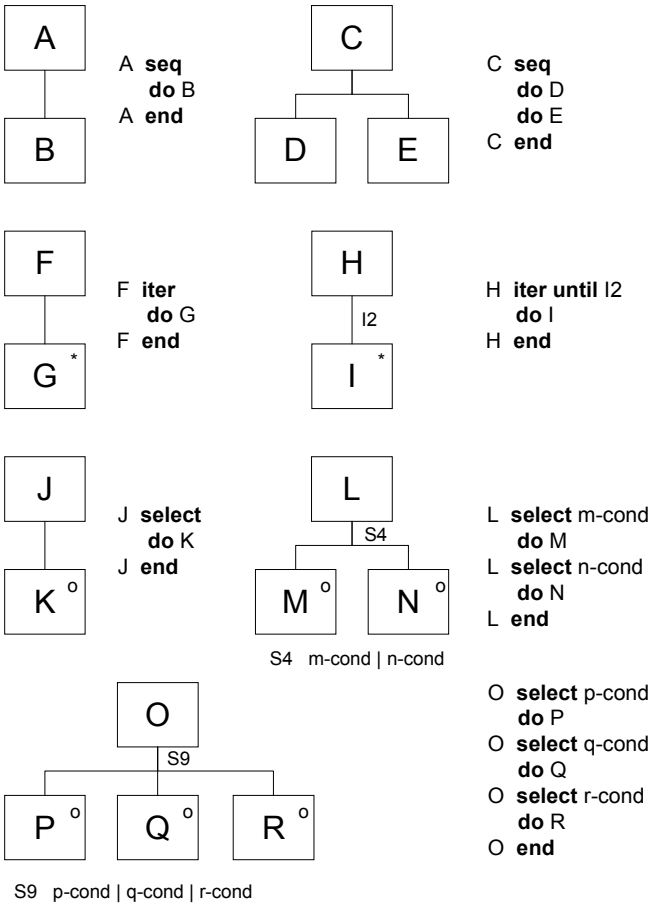


Abbildung 3.4: *Umsetzung von Strukturblöcken in schematische Logik*

Die linke Diagrammspalte zeigt zwei Fälle fehlender Kontrollen. Nur die Markierungen o oder * sind vorhanden. Das ist diagrammtechnisch vorläufig zulässig, muss aber spätestens bei der Umsetzung in die Schematische Logik spezifiziert werden.

Auf der rechten Seite fällt auf, dass Jackson verlangt, die Auswahlbedingungen für alle Zweige einzeln anzugeben. Da n-cond die Negation von m-cond sein muss, und r-cond alle Fälle umfasst, die nicht von p-cond oder q-cond abgedeckt werden, handelt es sich um eine durchaus störende Redundanz: Die übrigbleibenden Fälle – hier: n-cond beziehungsweise r-cond – müssen exakt definiert werden, obwohl sie sich aus der boole'schen Algebra als Restmengen ergeben.

Daher wurde bis auf dieses Unterkapitel durchgehend darauf verzichtet, die S-Kontrollen in Jacksons Sinn zu überspezifizieren. Es reicht völlig, die Positiv-Fälle anzugeben. Die Negativ-Fälle (ganz) rechts bleiben dann übrig. Im vorliegenden, recht einfachen Fall ist die geschachtelte Darstellung noch halbwegs übersichtlich. Sie sieht hier so aus:

```
Example1 seq
    open input file
    open output file
    read input record = in_string
    gen. header seq
        print „Text ...“ line
        print empty line
    gen. header end
    build body seq
        total_chars := 0
        total_blanks := 0
        total_len := 0
        build extended strings iter until eof_indicator = 'E'
            build extended string seq
                chars := 0
                blanks := 0
                len := length(in_string)
                ind := 1
                process input string iter until ind > len
                    process character select substr(in_string, ind, 1) = ' '
                        blanks := blanks + 1
                        ind := ind + 1
                    process character select substr(in_string, ind, 1) <> ' '
                        chars := chars + 1
                        ind := ind + 1
                    process character end
                process input string end
                total_chars := total_chars + chars
                total_blanks := total_blanks + blanks
                total_len := total_len + len
                copy input string seq
                    copy in_string to output area
                copy input string end
                cnv. string counter seq
                    convert string counters to counter columns
                cnv. string counter end
                print copied in_string & counter columns
```

```
                read input record = in_string
            build extended string end
        build extended strings end
    build body end
    gen. footer seq
        convert total counters to counter columns
        print === line
        print „totals“ & counter columns
    gen. footer end
    close input file
    close output file
    stop
Example1 end
```

Dennoch würde dieser Code keinen Audit überstehen, bei dem unter anderem überprüft wird, ob die Schachtelung mehr als drei Ebenen tief ist. Die „flache" Darstellungsform derselben Schematischen Logik sieht wie folgt aus:

```
Example1 seq
    open input file
    open output file
    read input record = in_string
    do gen. header
    do build body
    do gen. footer
    close input file
    close output file
    stop
Example1 end

gen. header seq
    print „Text ...“ line
    print empty line
gen. header end

build body seq
    total_chars := 0
    total_blanks := 0
    total_len := 0
    do build extended strings
build body end

gen. footer seq
    convert total counters to counter columns
    print === line
    print „totals“ & counter columns
gen. footer end
```

```
build extended strings iter until eof_indicator = 'E'
    do build extended string
build extended strings end

build extended string seq
    chars := 0
    blanks := 0
    len := length(in_string)
    ind := 1
    do process input string
    total_chars := total_chars + chars
    total_blanks := total_blanks + blanks
    total_len := total_len + len
    do cnv. string counter
    print copied in_string & counter columns
    read input record = in_string
build extended string end

process input string iter until ind > len
    do process character
process input string end

process character select substr(in_string, ind, 1) = ' '
    blanks := blanks + 1
    ind := ind + 1
process character select substr(in_string, ind, 1) <> ' '
    chars := chars + 1
    ind := ind + 1
process character end

copy input string seq
    copy in_string to output area
copy input string end

cnv. string counter seq
    convert string counters to counter columns
cnv. string counter end
```

Nun scheint die Übersichtlichkeit teilweise verlorengegangen zu sein. Sinnvoller wäre wohl eine Kombination beider Darstellungformen, bei der einerseits zu tiefe Schachtelungen vermieden, andererseits nicht so viele kleine Code-Sequenzen gebildet werden.

Was fällt ausserdem an der Schematischen Logik auf?

- Sie verdeutlicht, dass die bislang als formal bezeichneten Strukturblöcke oberhalb der I-Kontrollen in Wirklichkeit die Schleifen selbst repräsentieren. Dort sind die Iterations-Endebedingungen anzugeben. Beispiel: Build Extended Strings mit dem Schleifenkörper „build extended string".

- Die Auslagerung von Strukturblöcken mit **do**, das in Jacksons Buch übrigens nicht fett formatiert ist, entspricht dem PERFORM-Konstrukt in COBOL. Besonders die flache Darstellungsform erinnert an COBOL I, eine Sprache, die ausser dem Punkt keine explizite Beendigung einer Anweisung anbot. Erst mit COBOL II wurden Ende-Verben eingeführt – END-IF, END-EVALUATE, END-PERFORM etcetera – und es wurden Schleifenkonstrukte eingeführt, die nicht mehr zur Auslagerung der Schleifenkörper in eigene Paragraphen zwangen. Insofern war Jacksons Schematische Logik ihrer Zeit voraus. In seinem Buch sind allerdings viele Beispiele in COBOL I zu finden, das mittlerweile völlig antiquiert anmutet, was die Lektüre für heutige Programmierer(innen) stark erschwert.
- Jede benannte Struktur beginnt mit ihrem Namen, gefolgt von **seq** oder **iter until** plus Schleifenendebedingung oder **select** plus Auswahlbedingung des ersten Strukturblocks. Sie endet mit einer Zeile, die den Namen wiederholt, gefolgt von **end**. Bei Alternativen mit zwei oder mehr Fällen sind weitere Zeilen eingeschoben, die mit demselben Namen und einem weiteren **select** beginnen, gefolgt von der Auswahlbedingung des betreffenden Strukturblocks. Dadurch wirkt die Schematische Logik aufgebläht und erfordert viel Schreibarbeit, sowie große Sorgfalt. Nachträgliche Änderungen sind sehr aufwendig, vor allem, wenn auch noch Ein- und Ausrückungen korrigiert werden müssen, die sich über viele Zeilen erstrecken.
- Diese Ein- und Ausrückungen spiegeln die Tiefe wieder, die in der Programmstruktur mit Operationen jeweils erreicht ist. Das erleichtert zwar das Lesen der Schematischen Logik, erfordert aber große Disziplin. Es ist ziemlich wahrscheinlich, dass nach einigen Änderungen diese Schachtelungen nicht mehr stimmen.
- Für S3 wurde die SUBSTR-Formulierung gewählt. Diese ist am neutralsten.
- Es gibt keine Datendeklarationen und keine Parameter. Da die Schematische Logik dem Schreiben des eigentlichen Programms unmittelbar vorangeht, in dem zum Beispiel auf JSP basierende Unterprogramme mit ganz anders entwickelten Code-Teilen zusammen auftreten können, ist diese Selbstbeschränkung fragwürdig. Das Bestreben, möglichst universell nutzbar zu sein, und das Erfordernis der Klarheit stehen sich hier im Weg.
- Die Auflösung der Operationslisten in die sequentiell angeordneten Operationstexte macht die Schematische Logik zwar lesbarer, aber zugleich bei Änderungen anfälliger. Der Austausch eines Operationstexts gegen einen anderen muss an allen Stellen stattfinden, wo sonst die zugehörige Operationsnummer stünde, also bei häufig vorkommenden Operationen vielmals. Da kann schon mal etwas übersehen werden.

Diese Mängel haben es der Schematischen Logik verwehrt, zu einer populären Darstellungsform aufzusteigen. Da geübte Anwender der Methode ihr Programm sowieso meistens direkt aus der Programmstruktur mit Operationen ableiten, könnte der Nutzen der Schematischen Logik allenfalls in besonders vertrackten Fällen zum Tragen kommen – und dort hilft er nicht wirklich weiter. Für diese Fälle folgt daher anschließend ein weiterer Pseudocode, der sich schon oft bewährt hat.

Fazit: Die von Jackson propagierte Schematische Logik ist der Zeit um 1975 geschuldet, in der das Programmieren noch unter den Paradigma von ALGOL, COBOL I und FORTRAN zu stehen schien, obwohl damals bereits die ersten objektorientierten Sprachen aufkamen und die Vorgänger moderner Sprachen entstanden. Für diese Buchreihe hat sie nur noch historische Bedeutung.

3.4 BDL-Pseudocode

Dieser Pseudocode stammt aus dem Buch „Principles of Information Systems Analysis and Design" [7] von Harlan D. Mills[6], Richard C. Linger und Alan R. Hevner, Kapitel 2.6 „Introduction to Box Description Language" ab Seite 77ff. Die Programmstruktur mit Operationen wird in einen der „Box Definition Language" (BDL) ähnlichen Pseudocode transformiert, um diesen Code anschließend zu implementieren. Das ist natürlich nur dann sinnvoll, wenn

- dieser Zwischenschritt zur Erfüllung einer Vorschrift zwingend erforderlich ist, beispielsweise dann, wenn Programmvorgaben immer in Pseudocode abzufassen sind (und Struktogramme nicht akzeptiert werden), oder
- ein besonderer Schwierigkeitsgrad vorliegt, der eine Pseudocode-Darstellung zwingend nötig macht, zum Beispiel bei Programminversionen.

Die BDL wird in dem genannten Buch genau definiert und ausgiebig verwendet. Sie passt hier auch deshalb gut hin, weil sie sich einerseits klar von fachlich orientiertem Pseudocode, andererseits von den gängigen Implementierungssprachen abgrenzt. Verwechslungen sind somit leicht vermeidbar. BDL verwendet eine einfache, ALGOL- oder C-ähnliche Syntax.

3.4.1 Exkurs zum Hintergrund von BDL

Der Ausdruck „Box" bezieht sich auf die im Buch gelehrte Methode. Diese versucht, prozessorgesteuerte Geräte, organisationsinterne Prozesse und sogar ganze Anwendungssysteme durch hierarchische Zerlegung zu analysieren und zu beschreiben. Hierfür wird eine Systemsicht eingenommen, die von dem realen Substrat – Technik, Organisation, Programme / Daten – und den damit stattfindenden physischen Prozessen abstrahiert.

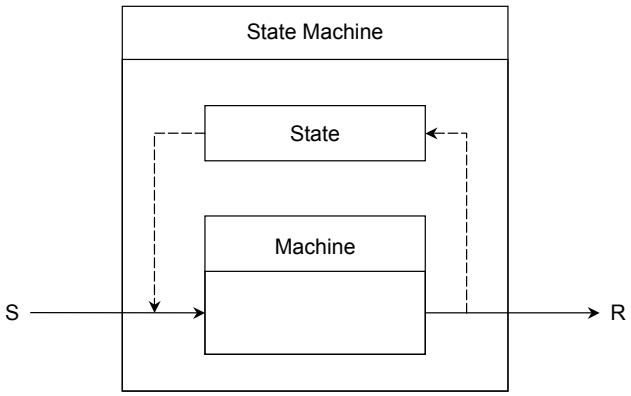
Jedes solche System wird als eine Black Box aufgefasst, die auf Stimuli (S) reagiert, also zum Beispiel auf Dialogeingaben, und daraufhin Antworten = Responses (R) erzeugt:

Original: Figure 2.5-1. Stimulus/Response Description of a Black Box.

Abbildung 3.5: *Systemsicht als Black Box*

Das Verhalten einer solchen Black Box ist durch die Stimulus History bedingt, also durch die Geschichte aller früheren Interaktionen mit dem System. Je nachdem, welche Spuren diese Historie zurückliess und wie die inneren Bestandteile der Black Box konstruiert sind, können die Antworten auf einen wiederholten – identischen – Stimulus unterschiedlich ausfallen. Natürlich kann man in der Praxis keine beliebig weit zurückreichende Stimulus History aufbewahren und vor jeder Systemreaktion konsultieren. Daher wird diese Historie in den

Status-Vektor einer Zustandsmaschine = State Machine transformiert. Darunter kann man sich einen möglicherweise sehr komplexen Automaten vorstellen:

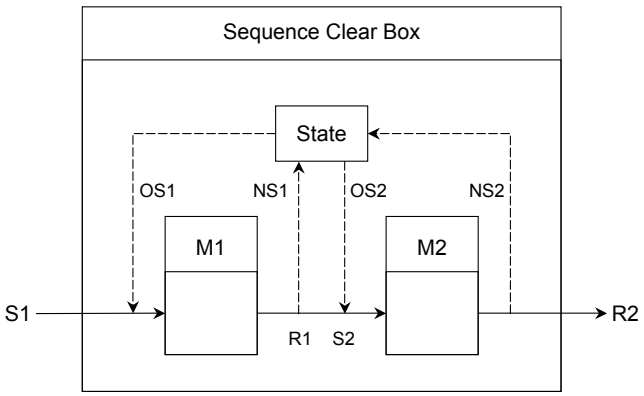


Original: Figure 3.1-1. State Machine.

Abbildung 3.6: *Diagramm der State Machine*

Die State Machine muss nun in Arrangements einfacherer Komponenten zerlegt werden. Diese sind ihrerseits Black Boxes. Sie werden durch Sequenz, Alternative, Iteration oder Parallelverarbeitung miteinander verknüpft. Diese Komponenten haben Zugang zum gemeinsamen Status-Vektor, besitzen aber auch ihre eigene Stimulus History und somit jeweils einen Status-Vektor. Sie heißen „Clear Boxes".

Die folgenden Grafiken zeigen die vier Clear Box-Arten:



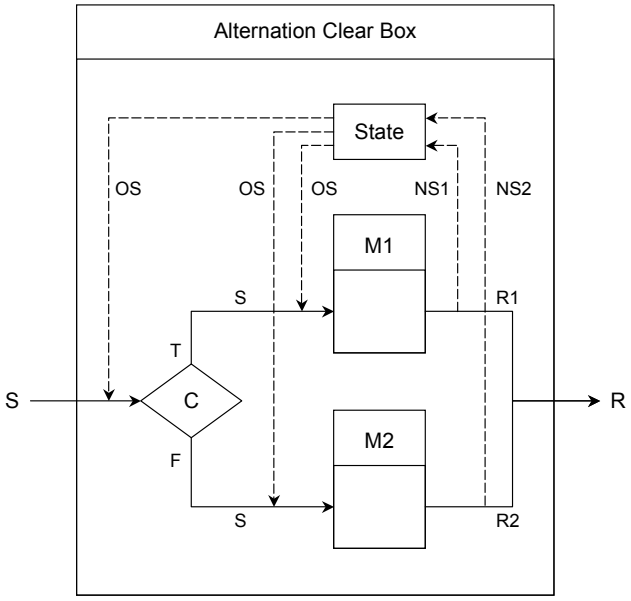
Original: Figure 4.1-2. The Sequence Clear Box Structure.

Abbildung 3.7: *Diagramm der Sequence Clear Box*

Die Abkürzungen bedeuten:

S1 = Stimulus für Machine M1
R1 = Response von M1
S2 = Stimulus für Machine M2 = R1
R2 = Response von M2
OS1 = Old State 1

NS1 = New State 1
OS2 = Old State 2
NS2 = New State 2



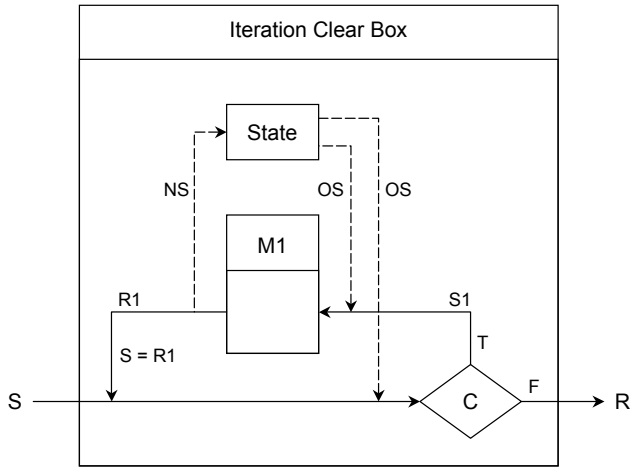
Original: Figure 4.1-3. The Alternation Clear Box Structure.

Abbildung 3.8: *Diagramm der Alternation Clear Box*

Die neuen Abkürzungen bedeuten:

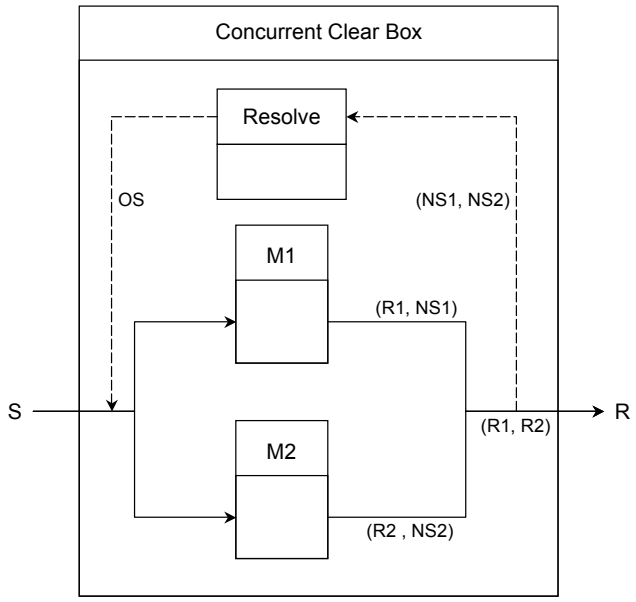
C Conditional test
T `true`
F `false`
OS Old State

Anmerkung: Es existiert zusätzlich eine „Case Clear Box Structure", die eine Auswahl aus mehr als zwei Black Boxes trifft. Sie ist eine Generalisierung der hier gezeigten Alternation Clear Box Structure.



Original: Figure 4.1-5. The Iteration Clear Box Structure.

Abbildung 3.9: *Diagramm der Iteration Clear Box*



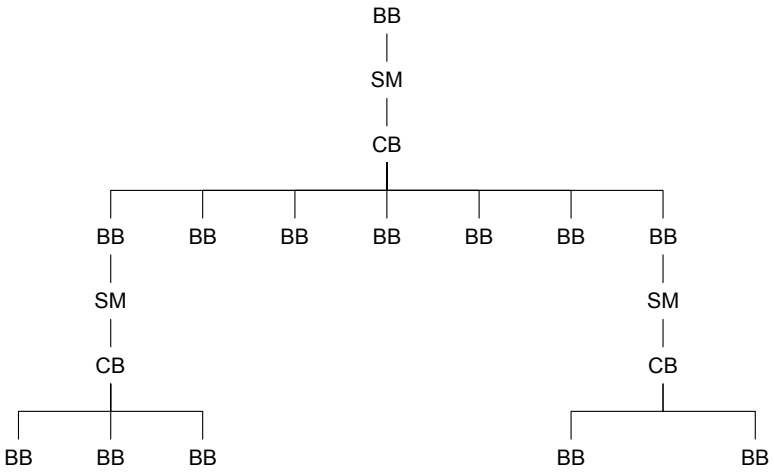
Original: Figure 4.1-5. The Concurrent Clear Box Structure.

Abbildung 3.10: *Diagramm der Parallelverarbeitungs-Clear Box*

Es ist nicht möglich und auch nicht sinnvoll, hier auf diese Strukturen detailliert einzugehen. Die Komplexität der Modellierung wird jedoch schon beim Betrachten der vielen Pfeile deutlich, denen jeweils konkrete Bedeutungen im Rahmen der zugrundeliegenden Automatentheorie zuzuordnen sind.

Durchgezogene Pfeile entsprechen Datenflüssen, die bis auf die Ebene einzelner Datenpakete heruntergebrochen werden müssen. Als Datenpaket kann beispielsweise auch der Transfer gedeutet werden, den ein einzelner Tastendruck auf einem Taschenrechner verursacht. Gestrichelte Pfeile deuten an, wie Statusinformationen auf die Behandlung der Datenflüsse oder Datenpakete einwirken. Die Grafiken implizieren, dass Statusinformationen und Aktionsdaten zusammengeführt werden, bevor diese eine Black Box oder ein Verzweigungselement erreichen. Es ist jedoch visuell unklar, wie das geschehen soll, da an diesen Stellen keine dafür zuständige Instanz erkennbar ist. Besser wäre es gewesen, die Statusinformationen an Steuereingänge der Black Boxes oder der Verzweigungselemente anzulegen beziehungsweise diese von Steuerausgängen zu entnehmen.

Der Zerlegungs- und Abbildungsvorgang Black Box ⇨ State Machine ⇨ Clear Box wird so lange wiederholt, bis die Clear Boxes einfach genug sind, um in BDL beschrieben zu werden. Dabei kann zum Beispiel folgender Zerlegungsbaum entstehen:



Original: Figure 1.4-1. A Box Structure Hierarchy.

Abbildung 3.11: *Eine Hierarchie Black Box (BB) – State Machine (SM) – Clear Box (CB) – Black Box etcetera*

Die Analogie zu den Jackson-Struktogrammen ergibt sich daraus, dass in beiden Fällen eine hierarchische Zerlegung stattfindet. Allerdings behandelt Jackson die Datenstrukturen und die Programmstrukturen gleich, während die Box-Methode eine rein prozedurale Sicht einnimmt. Sie kann daher auch als ereignisorientierte Analogie zur JSP-Methode aufgefasst werden, bei der allerdings die in der Abfolge der „Ereignisse" möglicherweise enthaltene Information *nicht* verwendet wird, um daraus die Programmstrukturen abzuleiten. Deshalb ähnelt sie am ehesten einem transaktionsorientierten Vorgehen.

Die Box-Methode eröffnet theoretisch die Möglichkeit, ein komplexes Anwendungssystem durch wiederholtes Anwenden derselben Partitionierungsprinzipien in seine elementaren Bestandteile zu zerlegen und diese zu implementieren. Aufgrund des Konstruktionsprinzips müsste das fertige System genau das gewünschte Verhalten aufweisen.

Dieser Ansatz ist einer der Vorgänger von Edward Yourdon's „Modern Structured Analysis" (MSA), die sich unter anderem des sogenannten Event Partitioning bedient. Die MSA ist eine äußerst nützliche Methode, mit

der komplexe Anwendungssysteme auf nachvollziehbare und stringente Weise analysiert und entworfen werden können.

Das kann man von der Box-Methode leider nicht behaupten. Sie führt letztendlich zu einer unüberschaubaren Menge von Diagrammen mit den drei Strukturen, sowie des entsprechend fragwürdigen BDL-Codes. Zudem ist die Datenstrukturkomponente der Methode so unterentwickelt, dass sie etwa mit dem Entity Relationship-Ansatz bei weitem nicht mithalten kann. Es ist wohl auch eine Überforderung (und eine Hybris), kleinste Objekte wie Taschenrechner und riesige Objekte wie ganze Anwendungssysteme mit ein- und derselben Methodik beschreiben zu wollen.

Zudem ist der Ansatz, Anwendungssysteme hierarchisch zu zerlegen, oftmals gescheitert, weil er von der Annahme ausgeht, dies sei überschneidungsfrei möglich. Tatsächlich zeigt sich jedoch häufig, dass Funktionalitäten eines Zerlegungsstrangs auch in einem anderen benötigt werden, was entweder zu unerwünschter oder sogar gefährlicher Redundanz führt, oder mit einer alternativen Zerlegung zu vermeiden versucht wird. Letztlich stellt es sich nicht selten heraus, dass die eingangs gewählte Zerlegung weiter unten nicht mehr umgestossen werden kann, umgekehrt aber die auf der Detailebene neu gefundene Zerlegung nicht zur Partitionierung des Gesamtsystems taugt.

Beispiel: Ein Lagerhaltungssystem behandelt Einlagerung, Umlagerung und Auslagerung. Das sind scheinbar getrennte Funktionen, aber die Umlagerung zerfällt in eine lagerinterne Auslagerung mit nachfolgender Einlagerung. Gibt es also nur zwei Black Boxes namens Einlagerung und Auslagerung? Nein, denn eine Umlagerung unterscheidet sich von Einlagerung und Auslagerung in so vielen Punkten, dass sie darunter nicht subsumiert werden kann. Diese Unterschiede erzwingen eine eigene Modellierung, aber diese führt nach einigen Zerlegungen unumgänglich zum mehrfachen Auftreten derselben Prozesse beziehungsweise Funktionen oder Methoden in getrennten Systemteilen.

Die Ursache dieses Problems ist die unbegründete Annahme, Anwendungssysteme müssten hierarchisch zerlegbar sein. Verzichtet man auf diese Zwangsjacke, so wie es Yourdon tat, so entsteht eine natürliche Modellierung, die das Auftreten identischer Teile in unterschiedlichen Prozessen nicht als Problem, sondern als Vereinheitlichungs- oder Wiederverwendungsoption ansieht.

Folglich war die Mills-/Linger-/Hevner-Methode zwar chancenlos, aber nicht völlig nutzlos. Die von den Autoren vorgestellte Modellierungssprache „Block Definition Language" ist hinreichend modern, um Jacksons Schematische Logik abzulösen und somit den Übergang von der Modellierung zur Implementierung relativ aktuell und zugleich plattformneutral zu gestalten. Eine weitere mögliche Anwendung der BDL ist die Transformation unübersichtlicher oder problematischer Flussdiagramme in einfachere Strukturen. Sie wird von den drei Autoren im Kapitel „Deriving Clear Boxes from Natural Procedures" vorgeführt. Da die Modellierung nach Jackson stets wohlgeformte Flussdiagramme hervorbringt, ist die Anwendung dieser Transformation im Allgemeinen nicht erforderlich, ausser bei Programminversionen, beim Backtracking oder beim Restart von Programmen, die beispielsweise Datenbanken modifizieren.

3.4.2 Die BDL-Syntax

… enthält weit mehr Schlüsselworte und Formulierungsmöglichkeiten, als im Jackson-Kontext sinnvoll verwendet werden können. Die methodenspezifischen Sprachanteile, die letztlich aus der Automatentheorie stammen, sind hier nicht anwendbar. Daher wurden diejenigen Sprachmittel ausgewählt und angepasst, die einen modernen Pseudocode für JSP-Zwecke ergeben.

Folgende Regeln gelten:

- Schlüsselworte werden fett gedruckt beziehungsweise handschriftlich unterstrichen.
- Einige Schlüsselworte bilden wie folgt jeweils eine Klammer: **proc** – **corp**, **do** – **od**, **case** – **esac** und **if** – **fi**. Für ausgelagerte Sequenzen wurde die Klammer **seq** - **qes** hinzugefügt. Nach **proc** oder **seq** folgt die Bezeichnung, die im jeweils obersten Block der Programmstruktur steht.
- Eine Schleife wird wie folgt geschrieben: **until** <k> **do** … **od**, wobei <k> das Schleifenendekriterium ist. In der BDL wird die – abweisende – Schleife mit **while** codiert, was dazu zwingen würde, entgegen dem JSP-Standard die Schleifenendekontrollen zu invertieren.
- Fallkonstrukte werden mit **case** und **part** <p> geschrieben, wobei <p> das jeweilige Auswahlkriterium ist. Der letzte Fall, wenn alle expliziten Kriterien erschöpft sind, heißt **default**. **part**s können unmittelbar aufeinanderfolgen, was bedeutet, dass in all diesen Fällen dieselbe Verarbeitung auszuführen ist, die dem letzten **part** in einer solche Liste folgt. **part** und **default** eröffnen jeweils eine Klammer, die durch den nachfolgenden Term **part** | **default** | **esac** geschlossen wird.
- **proc**- oder **seq**-Inhalte, Schleifenkörper und bedingte Anweisungen werden eingerückt.
- Zuweisungen sind in ALGOL-Syntax geschrieben.
- Einfache zusammengesetzte Anweisungen werden durch Semikolon getrennt. Beispiel: a := 0; b := 1
- Wenn aus dem Kontext nicht klar ist, welche Anweisungen zusammengehören, werden sie zwischen **do** und **od** eingeschlossen. Diese Klammern können also weggelassen werden, wenn – zusammengesetzte – Anweisungen ihrerseits unmittelbar von äußeren Klammern umschlossen sind, wie zum Beispiel **if** <s> **then** … **else** … **fi**, wobei <s> die Auswahlbedingung ist.
- Ausgelagerte Pseudocode-Teile werden mit **use** „ausgeführt". Sie beginnen mit **seq** und enden mit **qes**.
- Unterprogramm- und Methodenaufrufe erfolgen mit **call**.
- Datendeklarationen und Parameter sind möglich. Da die BDL-Deklarationsmittel für Daten nicht präzise genug sind, wird stattdessen eine an SQL/PL angelehnte Syntax verwendet (siehe spätere Beispiele).
- Kommentare werden in Klammern geschrieben oder mit // eingeleitet. Sie können überall dort eingestreut werden, wo dies syntaktisch eindeutig möglich ist.
- Falls erforderlich oder sinnvoll, können zusammengesetzte Anweisungen als „Procedure Statements" zusammengefasst und mit PSn abgekürzt werden, wobei n eine laufende Nummer ist. Jedes Konstrukt, das nur einen Eingang und einen Ausgang hat, eignet sich für diese Abkürzung.

Die „Abbildung 3.12: Umsetzung von Strukturblöcken in Pseudocode" auf Seite 70 veranschaulicht einige dieser Regeln.

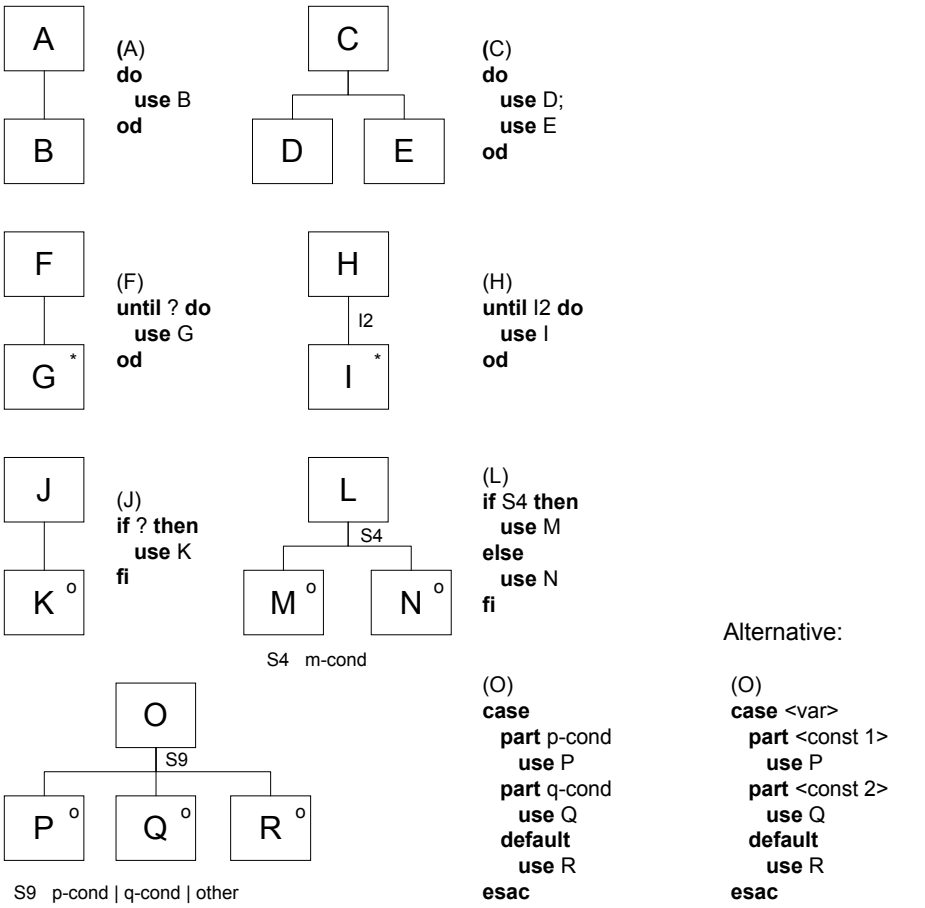


Abbildung 3.12: *Umsetzung von Strukturblöcken in Pseudocode*

Damit wird die Beispiel-Programmstruktur mit Operationen in den folgenden Pseudocode transformiert, ohne die Operationskürzel in ihre Volltexte zu expandieren:

```
proc Example1;
    1,2,6,7,8,9,10,11;
    until I1 do
        12,13,14,15;
        until I2 do
            if S3 then
                18,16
            else
                17,16
            fi
```

```
        od;
        19,20,21,22,23,24,6
    od;
    25,26,27,3,4,5
corp
```

Das folgende Flussdiagramm zeigt einen weiteren Vereinfachungsschritt

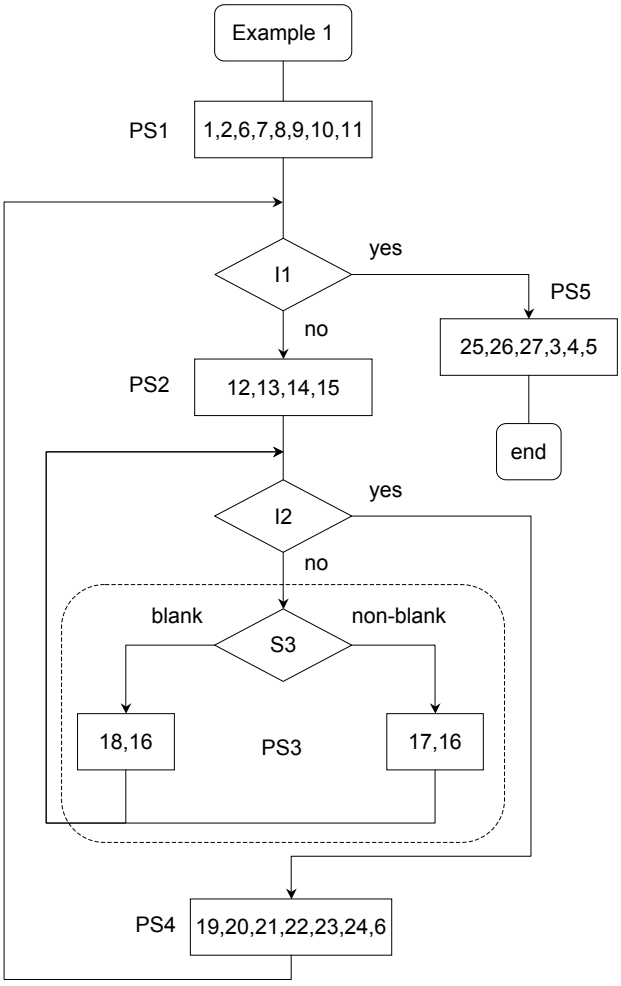


Abbildung 3.13: *Zusammenfassung von Anweisungen zu Procedure Statements*

Aus der Operationsfolge 1,2,6,7,8,9,10,11 wird PS1. Die Abfrage S3 und die nachfolgenden Operationen 18,16 beziehungsweise 17,16 werden zu PS3 zusammengefasst etcetera.

Nun könnte man noch weitergehen und beispielsweise die I2-Schleife als Procedure Statement PS6 zusammenfassen. Danach käme die I1-Schleife dran. Am Ende wäre das ganze Programm ein einziges Procedure Statement. Das ist zwar sehr übersichtlich, zugleich aber auch sinnlos. Da zumindest die Schleifen-Kontrollstrukturen und wesentliche Alternativ-Abfragen gezeigt werden sollen, wird die Zusammenfassung bei hinreichender Vergröberung abgebrochen. Was hinreichend ist, wird von Fall zu Fall entschieden.

Das vorliegende Flussdiagramm schrumpft daher zu folgendem Gebilde zusammen:

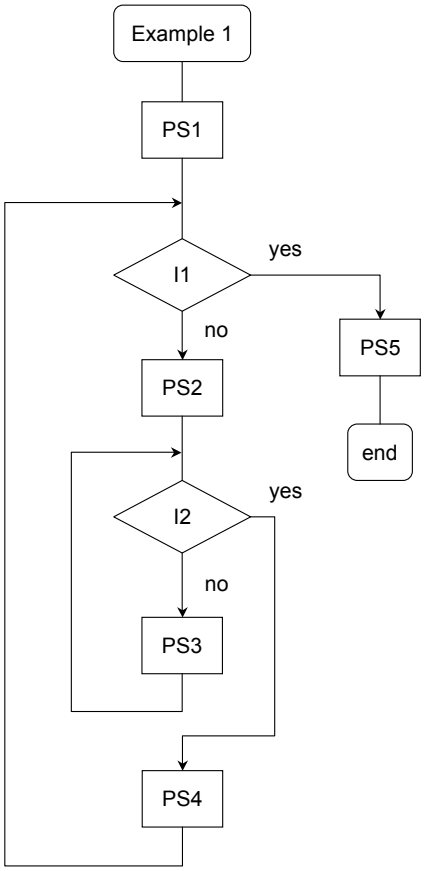


Abbildung 3.14: *Stark verdichtetes Flussdiagramm mit Procedure Statements*

Der folgende Pseudocode entspricht diesem Flussdiagramm:

```
proc Example1;
    PS1;
    until I1 do
        PS2;
        until I2 do
            PS3
        od;
        PS4
    od;
    PS5
corp
```

Kürzer geht es kaum. Beim Programmieren müssen natürlich die Procedure Statements wieder in ihre Bestandteile aufgelöst werden, was aber nicht schwierig ist. Ausserdem wird so die Redundanz vermieden, die

sich in der Schematischen Logik bei Änderungen von Operationstexten so nachteilig bemerkbar macht. Der Umweg über das Flussdiagramm – und der damit verbundene Zeichenaufwand, sowie die Irrtumsmöglichkeiten – oder über einen umfangreichen BDL-ähnlichen Pseudocode können ausserdem leicht vermieden werden, wenn die Zusammenfassungen schon in der Programmstruktur mit Operationen vorgenommen werden.

Dies zeigt die folgende Grafik:

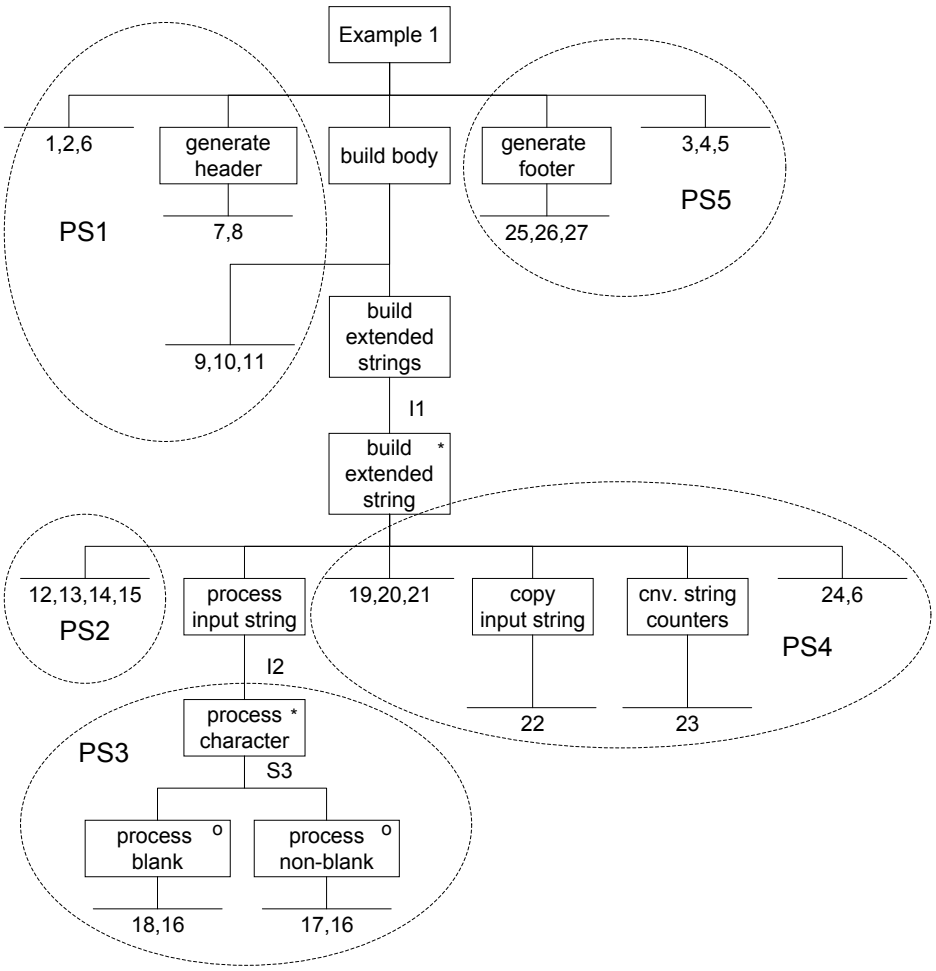


Abbildung 3.15: *Mögliche Zusammenfassungen zu Procedure Statements in der Programmstruktur*

Auch komplexe Struktogramme können rasch in diesen Pseudocode übersetzt werden. Daher ist er als Übergang zur Programmierung der Schematischen Logik allemal vorzuziehen. Allerdings wird ab hier die Bezeichnung BDL nicht mehr verwendet, weil sie zum einen an die Top-Down-Analysemethode gebunden ist, zum anderen nach und nach weitere Pseudocode-Elemente definiert werden, die mit dem ursprünglichen Zweck der Sprache nichts mehr zu tun haben. Da Jackson die etwas fremdartige Bezeichnung „Schematische Logik" einführte, wird die an die BDL angelehnte Form ab hier einfach **Pseudocode** genannt.

3.5 Anmerkungen zum Pseudocode

Der von der BDL abstammende Pseudocode wird – ausser für die vorangehenden Zwecke – auch dazu verwendet, die Operationen und Kontrollen zu definieren. Erste Beispiele hierzu liefert die Liste der Operationen im Kapitel „2.8 Programmstruktur mit Operationen ergänzen" auf Seite 34 ff. Dort – und auch in den vielen noch folgenden Operationslisten – sind meistens einfache Zuweisungen oder arithmetische Operationen beziehungsweise String-Manipulationen zu finden. Es ist aber auch zulässig, einfache Abfragen und sogar Schleifenkonstrukte als Operationen zu formulieren, um die grafische Modellierung zu erleichtern. Das darf aber nur sehr diszipliniert verwendet werden, damit die Programm-Struktogramme nicht durch versteckte Funktionalität entwertet werden. Im geeigneten Fall kann diese Option aber dazu dienen, unnötig aufwendige Grafiken zu vermeiden.

Da die Operationen häufig zielsystem-neutral formuliert werden, sind auch Texte wie „convert string counters to counter columns" (Op. 23) zulässig – vorausgesetzt, dass die Bedeutung solcher Anweisungen andernorts exakt definiert wird. Auch das ist also Pseudocode. Die Operationen sollen zwar möglichst klar formuliert werden, aber auch dann gibt es gegebenenfalls noch Interpretationsspielräume. Daher enthalten die künftigen Operationslisten eine weitere Spalte „Kommentar", die nicht für jede Operation ausgefüllt werden muss. Diese Spalte bietet vor allem die Möglichkeit, Klarstellungen vorzunehmen oder nützliche Anmerkungen zu machen. Beispiel: Kapitel „7.6 Operationen" auf Seite 143 ff.

Am Anfang dieses Beispiel-Kapitels findet sich auch das erste Exemplar einer Array-Deklaration, das sich an den Stil von PL/SQL[7] anlehnt. Wer es unbedingt möchte oder es für erforderlich hält, kann auch einfache (nicht zusammengesetzte) Datenelemente in ähnlicher Form deklarieren. Das ist vor allem dann sinnvoll, wenn die Implementierung von Programmierer(inne)n vorgenommen werden soll, die „harte" Vorgaben benötigen. Die Herkunft der JSP-Methode macht sich bei alledem insofern bemerkbar, als es keine Deklarationsform gibt, die den OO-Klassendefinitionen entspräche. Diesen Mangel kann man aber auch insofern als Vorteil deuten, dass die Entwicklung sinnvoller Klassen zum Zeitpunkt des Erstellens von Operationslisten häufig noch als zu schwierig erscheint. Es wäre an dieser Stelle zudem verfrüht, das komplexe – im Folgekapitel „4 Objektorientierte Implementierungsvorbereitungen" auf Seite 83 ff beginnende – Thema „OO-Programmierung auf JSP-Basis" anzuschneiden.

Kontrollen können am Beginn einer Modellierung häufig nur unscharf definiert werden. Dafür ist jede geeignete verbale Form zulässig, auch wenn sie zunächst als umständlich erscheinen mag. Erst nach und nach werden die Datenelemente und Informationseinheiten (= Aggregate einfacherer Strukturen oder Datenelemente) sichtbar, auf welche sich die Kontrollen stützen. Dann kommt die Zeit, um die Kontrollen so weit zu präzisieren, wie möglich und sinnvoll. Das heisst, dass an die Stelle der verbalen Beschreibungen Ausdrücke treten, die sich auf die Daten stützen. Diese Ausdrücke müssen der boole'schen Logik folgen, also jeweils ein wohldefiniertes Ergebnis abliefern. „Fuzzy Logic" ist nicht vorgesehen. Im Verlauf dieser Präzisierung wird zwingend klar, ob die Kontrollen problemlos umsetzbar sind, also anhand des aktuell im Speicher des Programms befindlichen Datenmaterials an ihrem jeweiligen Ort sofort entschieden werden können, oder ob das nicht zutrifft. Im zweiten Fall liegt ein Problem vor, das entweder durch eine andere Modellierung desselben Sachverhalts gelöst werden kann, oder das einer speziellen Lösung bedarf. Die Stichworte hierzu lauten: Mehrfaches Vorlesen, Backtracking und Programminversion. Diese Optionen werden in künftigen Kapiteln intensiv diskutiert.

Der Pseudocode umfasst also die Operationen, die Kontrollen, die Definitionen von Datenelementen beziehungsweise Informationseinheiten, sowie – optional – die Umsetzung von Programm-Struktogrammen in Text.

3.6 Nassi-Shneiderman-Diagramme zeichnen

Nassi-Shneiderman-Diagramme[8] sind, historisch gesehen, jünger als die JSP-Diagramme. Sie sind heutzutage noch hauptsächlich im Informatikunterricht der Sekundarstufe II relevant. Strikte zeichnerische Vorgaben für diese Diagramme sorgen dafür, dass der damit modellierte Code streng den Prinzipien der strukturierten Programmierung folgt. Sie sind weniger abstrakt als die JSP-Struktogramme, erfordern jedoch – vor allem bei nichttrivialen Programmstrukturen – einen erheblichen Zeichenaufwand. Das zeigen die folgenden Gegenüberstellungen von JSP-Diagrammelementen zu ihren Nassi-Shneiderman-Äquivalenten und – später – das damit modellierte Beispiel anhand Example 1.

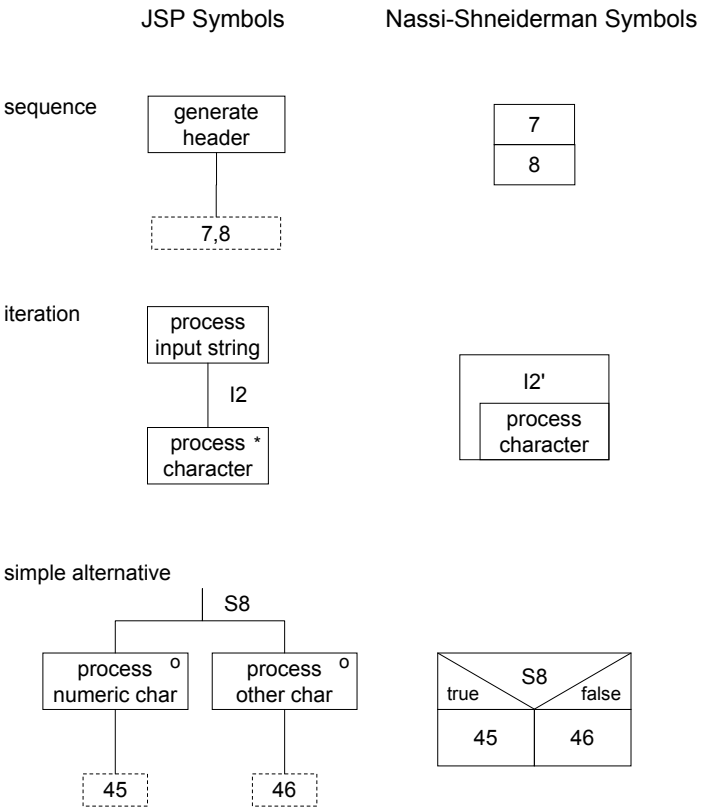


Abbildung 3.16: *Äquivalenzen zwischen Strukturblöcken und Nassi-Shneiderman-Elementen*

Aus Gründen der besseren Vergleichbarkeit und der Fairness stehen in den rechteckigen Anweisungsblöcken auf der rechten Seite ebenfalls Operationsnummern anstelle der üblichen knappen verbalen Bezeichnungen. Jede Operation benötigt einen eigenen Anweisungsblock. Der Block „process character" müsste eigentlich in diejenigen Anweisungsblöcke aufgelöst werden, aus denen dieser JSP-Strukturblock besteht, aber hier geht es nur darum, die grafischen Vorgaben zu demonstrieren:

- Eine Sequenz besteht aus der Abfolge der Operationen in den Anweisungsblöcken als Stapel von oben nach unten.
- Eine Iteration besteht aus einer Umhüllung und dem Schleifenkern. Die Iterationskontrolle I2 erscheint hier invertiert, weil die Nassi-Shneiderman-Diagramme vom Verhalten der abweisenden „while"-Schleife

ausgehen. Diese endet erst, wenn die Schleifenbedingung nicht mehr erfüllt ist. Im Gegensatz dazu verwendet JSP die abweisende „until"-Schleife.

- Eine einfache Alternative wird als Trichter dargestellt. Links darunter steht derjenige Anweisungsblock, der ausgewählt werden soll, wenn die S-Kontrolle erfüllt ist – wie bei JSP auch. Rechts daneben steht der „else"-Zweig, der leer sein kann. Das ist der Fall „optional part".

Die folgende Grafik zeigt Abwandlungen dieses Schemas.

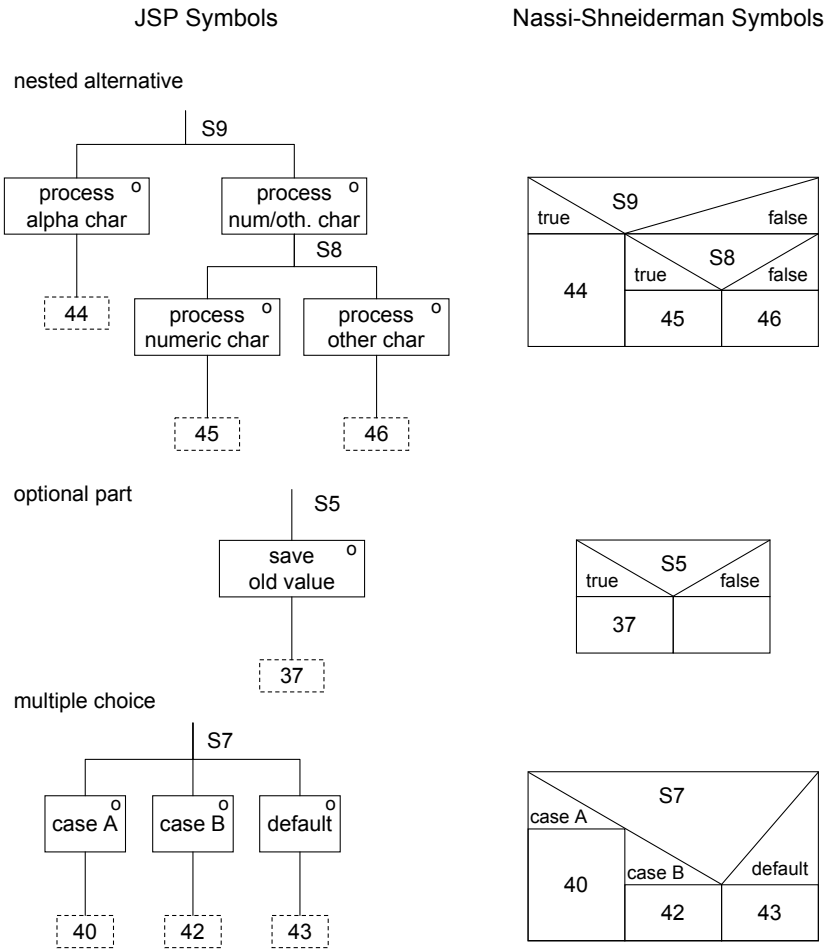


Abbildung 3.17: *Nassi-Shneiderman-Varianten für Schachtelung, einfache Auswahl und Case-Strukturen*

„Nested alternative" zeigt uns zweierlei: a) Der Wegfall der Strukturblockbezeichnungen verringert zwar den Darstellungsaufwand, aber macht es zugleich schwerer, das Diagramm zu verstehen. b) Der Anweisungsblock für die Op. 44 ist hier doppelt so groß wie der für die gleichrangigen Op. 45 und 46.

Die Nassi-Shneiderman-Diagrammtechnik erzwingt einen zunehmenden Verschnitt bei gleichzeitiger Schrumpfung des Platzes, der für die Operationsbezeichnungen verfügbar ist. Das fällt hier nicht so stark auf, weil anstelle der Bezeichnungen die Operations-Nummern stehen, aber es wird natürlich bei jeder Verfeinerung im Sinn der Top Down-Zerlegung immer schwieriger, noch ausreichend Platz zum Schreiben vorzuhalten.

Multiple Choice-Alternativen verwenden eine Abwandlung dieser Grafik, bei der die explizit genannten Positiv-Fälle links unter dem fallenden Schrägstrich als Treppe angeordnet werden und der „default"-Fall, also die Menge aller anderen Fälle, rechts davon unter dem steigenden Schrägstrich wie ein „else"-Zweig aussieht. Bei nur zwei „cases" wie im Beispiel ist das grafische Ergebnis noch harmlos, aber bei fünf und mehr Fällen wird die Grafik relativ unübersichtlich und die Kästchen für die alternativen Anweisungen schrumpfen erheblich.

Über die gezeigten Nassi-Shneiderman-Symbole hinaus existieren weitere, die in der DIN-Norm 66261 festgelegt sind. Deren Inhalt kann kostenpflichtig von mehreren Anbietern bezogen werden. Es handelt sich dabei um Symbole, die keine direkte Äquivalenz zu den JSP-Symbolen besitzen, nämlich a) vier zusätzliche Varianten des Schleifenkonstrukts, b) die Break-Anweisung zum Verlassen eines Anweisungsblocks und c) ein Symbol für Nebenläufigkeit (Parallelverarbeitung). Zusätzlich gibt es d) das – nicht genormte – Symbol für den Aufruf einer Unterfunktion. Die MSP verwendet dasselbe Symbol, um eine Auslagerung auszudrücken, die jedoch in den Nassi-Shneiderman-Diagrammen nur verbal gekennzeichnet werden kann. Das MSP-Auslagerungssymbol gilt allerdings sowohl für Daten- als auch für Programmstrukturen. Erstere können nicht mit Nassi-Shneiderman-Diagrammen dargestellt werden, denn diese sind rein prozedural.

Vorläufiges Fazit: Aus JSP-Struktogrammen für Code können stets die dazu äquivalenten Nassi-Shneiderman-Diagramme hergeleitet werden, weil die Strukturblöcke und Kontrollen mitsamt der „Besteht-aus"-Hierarchie die Regeln der strukturierten Programmierung strikt einhalten. In der Gegenrichtung gilt das wegen der erwähnten Zusatzsymbole nur eingeschränkt: a) Die zusätzlichen Schleifenvarianten müssen in eine ihnen äquivalente JSP-Form transformiert werden. b) Die Break-Anweisung ist generell bei JSP nicht vorgesehen; sie kann aber substituiert werden. Die MSP kennt auch einen „Aussprung"-Pfeil, aber dieser wird ausschließlich für das Backtracking verwendet, das nur in seltenen Fällen zur Anwendung kommt. c) JSP kann keine Nebenläufigkeit modellieren. d) Das Symbol für den Aufruf einer Unterfunktion hat bei JSP/MSP keine grafische Entsprechung, weil Funktions- oder Prozeduraufrufe dort nur den Rang einer Operation besitzen.

Die „Abbildung 3.18: Nassi-Shneiderman-Diagramme von Example 1" auf Seite 78 zeigt dessen Programmstruktur in zwei Formen, nämlich einerseits nur mit den JSP-Operationsnummern und -Kontrollen sowie andererseits mit den dazu korrespondierenden Texten. Wie ist diese Diagrammtechnik zu bewerten?

Positiv

- Die Nassi-Shneiderman-Diagrammtechnik ist intuitiv einleuchtend und leicht zu erlernen.
- Sie erzwingt die Einhaltung der Prinzipien der strukturierten Programmierung – mit dem kleinen Ausreißer der Break-Anweisung, die einem Sprung an das jeweilige Blockende gleichkommt.

Negativ

- Beim Entwurf sollte bereits eine Grobskizze des jeweiligen Diagramms vorliegen, weil der Zeichenaufwand für das detaillierte Nassi-Shneiderman-Diagramm sonst wegen mehrerer „Anläufe" rasch zunimmt und final zu erheblicher Frustration führt.
- Zwangsläufig werden die inneren Anweisungsblöcke bei zunehmender Verfeinerung immer kleiner. Das zwingt dazu, entweder die Lesbarkeit durch Abkürzungen zu verschlechtern oder Diagrammteile auszulagern. Letzteres beeinträchtigt wiederum die Übersicht und erschwert somit die Nachvollziehbarkeit.
- Der Zwang zu möglichst knapper Bezeichnung der Anweisungen wegen des Platzmangels geht zu Lasten der Verständlichkeit. Kommentare können im allgemeinen nicht in die Diagramme aufgenommen werden. Es fehlt der Kontext, den die Bezeichnungen der JSP-Strukturblöcke erzeugen.
- Änderungen erfordern gegebenenfalls einen hohen manuellen Aufwand. Zeichenwerkzeuge unterstützen das Erstellen und Ändern der Diagramme, sind aber nicht „flächendeckend" vorhanden.

- Wie die Flussdiagramme bieten Nassi-Shneiderman-Diagramme keinerlei Unterstützung beim Entwurf. Sie visualisieren lediglich die beim Entwurf antizipierte Programmstruktur. Daher sind sie eigentlich am besten dazu geeignet, vorgefundene Programmstrukturen zu dokumentieren. Das aber kann die JSP-Diagrammtechnik besser.
- Im Fall mehrfach verwendeter Operationen oder Kontrollen besteht die Gefahr inkonsistenter Änderungen, weil dann jedes Vorkommen gefunden und angepasst werden muss.
- Rekursivität wird nicht explizit unterstützt – im Gegensatz zu JSP/MSP.

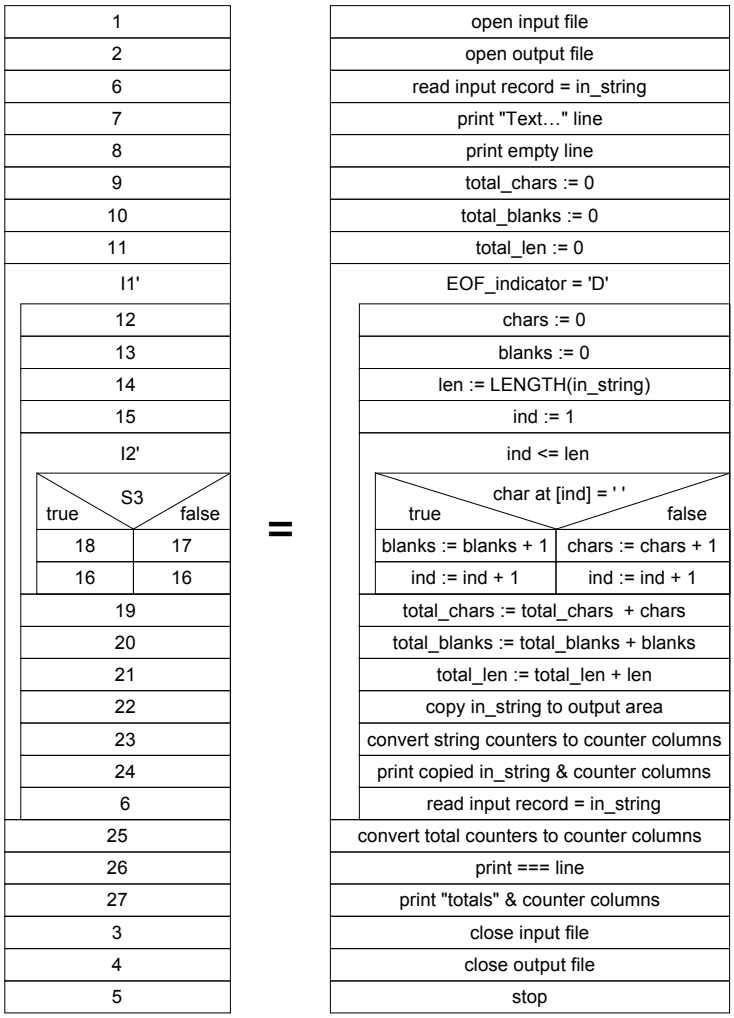


Abbildung 3.18: *Nassi-Shneiderman-Diagramme von Example 1*

Anmerkung: Der Wegfall der Strukturblockbezeichnungen macht es schwer, die links gezeigte Variante zu verstehen. Sie ist zwar deutlich kompakter als die rechte Variante, aber zu abstrakt.

Finales Fazit: Die Nassi-Shneiderman-Diagrammechnik ist wegen ihrer evidenten Mängel für die professionelle Anwendungsentwicklung komplett unattraktiv. Sie hat sich daher unter Profis nicht durchgesetzt und wird in den folgenden Kapiteln nicht mehr aufgegriffen.

Endnoten

1 Das **Rathaus von Lemgo** im Kreis Lippe ist ein epochenübergreifendes Baudenkmal von besonderer Bedeutung. Es steht auf der UNESCO-Liste 1 als Kunstwerk von europäischem Rang. Die Anfänge des Gebäudes liegen im Bau einer gotischen Markthalle, die in der mittelalterlichen Handelsstadt bald als Rathaus ausgebaut wurde, in dem verschiedene kommunale Funktionen eingerichtet wurden. Die Bauphasen liegen in der Blütezeit der Hansestadt Lemgo in Gotik und Renaissance. Die Prunkfassaden der Ratslaube und der Apothekenauslucht nehmen einen besonderen Rang in der regionalen Weserrenaissance ein.

[Wikipedia-Seitentitel: Rathaus (Lemgo)
Datum der letzten Bearbeitung: 14. Januar 2021, 09:18 UTC
Versions-ID der Seite: 207628342
Permanentlink: https://de.wikipedia.org/w/index.php?title=Rathaus_(Lemgo)&oldid=207628342
Datum des Abrufs: 13. Juni 2022, 17:28 UTC]

2 Die **sieben freien Künste** (lateinisch **septem artes liberales**, kurz auch **(sieben) artes liberales**, seltener auch studia liberalia) sind ein in der Antike entstandener Kanon von sieben Studienfächern. Aus den freien Künsten bestand traditionell die einem freien Mann ziemende Bildung, ihre Siebenzahl ist aber erst in der Spätantike bezeugt. Im mittelalterlichen Lehrwesen galten sie als Vorbereitung auf die Fakultäten Theologie, Jurisprudenz und Medizin.

[Wikipedia-Seitentitel: Sieben freie Künste
Datum der letzten Bearbeitung: 15. Mai 2022, 09:50 UTC
Versions-ID der Seite: 222890303
Permanentlink: https://de.wikipedia.org/w/index.php?title=Sieben_freie_K%C3%BCnste&oldid=222890303
Datum des Abrufs: 13. Juni 2022, 17:29 UTC]

3 **Augusta Ada King-Noel**, **Countess of Lovelace**, allgemein als **Ada Lovelace** bekannt (geborene Hon. Augusta Ada Byron; * 10. Dezember 1815 in London; † 27. November 1852 ebenda), war eine britische Mathematikerin und Gesellschaftsdame, die Tochter des Dichters Lord Byron.

Sie arbeitete mit Charles Babbage an der von ihm entwickelten Analytical Engine. Die Analytical Engine wurde zwar niemals fertig gestellt, aber Ada Lovelace erkannte das große Potential dahinter über die Verwendung als Maschine zur Berechnung mathematischer Tafeln hinaus, die ihr Erfinder Charles Babbage zunächst im Auge hatte. Dies manifestierte sich 1843 in selbst hinzugefügten Notizen zu ihrer Übersetzung eines Artikels von Luigi Federico Menabrea über die Analytical Engine, die dreimal so lang waren wie der ursprüngliche Text. Die Erkenntnis, dass die Maschine mehr als nur Zahlen verarbeiten könnte, war bahnbrechend, dies wurde jedoch zu ihrer Lebzeit nicht erkannt. Sie legte in ihren Aufzeichnungen und in der Veröffentlichung auch ein konkretes Programm für die Maschine am Beispiel der Berechnung von Bernoulli-Zahlen vor. Daher gilt sie manchen Historikern als erste Programmiererin der Welt. Dies wurde vor allem von Doron Swade, der sich intensiv mit Charles Babbages Biographie befasst, mit dem Argument kritisiert, konkrete Programmbeispiele hätten sich auch mehrere Jahre zuvor in Babbages Aufzeichnungen befunden. Das ändere nach Swade aber nichts an ihrer eigentlichen Bedeutung, visionär die weit über konkrete Rechnungen hinausgehende Bedeutung des Computers erkannt zu haben.

Die Programmiersprache Ada, die Lovelace Medal sowie der Ada Lovelace Award wurden nach ihr benannt.

[Wikipedia-Seitentitel: Ada Lovelace
Datum der letzten Bearbeitung: 22. Mai 2022, 10:33 UTC
Versions-ID der Seite: 223072265
Permanentlink: https://de.wikipedia.org/w/index.php?title=Ada_Lovelace&oldid=223072265
Datum des Abrufs: 13. Juni 2022, 17:30 UTC]

4 **Charles Babbage** (* 26. Dezember 1791 in Walworth, Grafschaft Surrey, England; † 18. Oktober 1871 in London) war ein englischer Mathematiker, Philosoph, Erfinder und Politischer Ökonom. Babbage ist vor allem

für die von ihm entwickelte Rechenmaschine Analytical Engine bekannt. Diese gilt als Vorläufer des modernen Computers. 1854 gelang ihm als Erstem die Entzifferung einer Vigenère-Chiffre. Nach Babbage sind der Asteroid (11341) Babbage und der Mondkrater Babbage benannt.

[Wikipedia-Seitentitel: Charles Babbage
Datum der letzten Bearbeitung: 6. Mai 2022, 00:09 UTC
Versions-ID der Seite: 222648767
Permanentlink: https://de.wikipedia.org/w/index.php?title=Charles_Babbage&oldid=222648767
Datum des Abrufs: 13. Juni 2022, 17:31 UTC]

5 Lovelace […] verglich die „analytische Maschine" mit dem zu dieser Zeit hochmodernen dampfbetriebenen Jacquard-Webstuhl. Diese neuartigen Webstühle konnten per Lochkartenprogrammierung beliebig komplizierte Muster ohne direkten menschlichen Einfluss herstellen.

[Quelle: siehe Wikipedia-Artikel „Ada Lovelace" – zuvor zitiert]

6 **Harlan D. Mills** (* 14. Mai 1919 in Liberty Center, Iowa; † 8. Januar 1996) war ein amerikanischer Informatiker, bekannt für Beiträge zur Softwaretechnik und zur Strukturierten Programmierung.

Mills war im Zweiten Weltkrieg Bomberpilot und Fluglehrer und wurde 1952 an der Iowa State University in Mathematik promoviert. Er lehrte an verschiedenen Universitäten (wie der Iowa State, der Johns Hopkins University, der University of Maryland, der Princeton University und der New York University) und war von 1964 bis 1987 bei IBM, wo er 1973 den Status eines IBM Fellow erreichte und Leiter des Software Engineering sowie Mitglied des IBM Corporate Technical Committee war. Zuletzt war er Direktor des von ihm gegründeten Information Systems Institute in Vero Beach und Professor am Florida Institute of Technology.

7 **PL/SQL** (**P**rocedural **L**anguage/**S**tructured **Q**uery **L**anguage) ist eine proprietäre Programmiersprache der Firma Oracle®.

PL/SQL verbindet die Abfragesprache SQL mit einer prozeduralen Programmiersprache. Die Syntax ist stark an die Programmiersprache Ada angelehnt.

Unterstützt werden Variablen, Bedingungen, Schleifen und Ausnahmebehandlungen. Ab Version 8 des Oracle-RDBMS halten auch objektorientierte Merkmale Einzug.

[Wikipedia-Seitentitel: PL/SQL
Datum der letzten Bearbeitung: 28. Juni 2021, 14:45 UTC
Versions-ID der Seite: 213370029
Permanentlink: https://de.wikipedia.org/w/index.php?title=PL/SQL&oldid=213370029
Datum des Abrufs: 13. Juni 2022, 17:36 UTC]

8 Ein **Nassi-Shneiderman-Diagramm** ist ein Diagrammtyp zur Darstellung von Programmentwürfen im Rahmen der Methode der strukturierten Programmierung. Es wurde 1972/1973 von Isaac Nassi und Ben Shneiderman entwickelt und ist in der DIN-Norm 66261 festgelegt.

Da Nassi-Shneiderman-Diagramme Programmstrukturen und Kontrollstrukturen darstellen, werden sie auch als **Struktogramme** bezeichnet.

[…]

Die Strukturierte Programmierung zerlegt das Gesamtproblem, das man mit dem gewünschten Algorithmus lösen will, in immer kleinere Teilprobleme – bis schließlich nur noch elementare Grundstrukturen wie Sequenzen und Kontrollstrukturen zur Lösung des Problems übrig bleiben. Diese können dann durch ein Nassi-Shneiderman-Diagramm visualisiert werden. Die Vorgehensweise entspricht der sogenannten Top-down-Programmierung, in der zunächst ein Gesamtkonzept entwickelt wird, das dann durch eine Verfeinerung der Strukturen des Gesamtkonzeptes aufgelöst wird.

Böhm und Jacopini haben 1966 nachgewiesen, dass sich jeder beliebige Algorithmus ohne unbedingte Sprunganweisung (GOTO) formulieren lässt. Für Nassi-Shneiderman-Diagramme lassen sich trivial die Kontrollstrukturen moderner Programmiersprachen finden; für Programmablaufpläne kann dies wesentlich schwieriger sein.

[…]

In der Softwareentwicklung werden Nassi-Shneiderman-Diagramme heute selten eingesetzt. Dort werden vorrangig erweiterte Programmablaufpläne (Aktivitätsdiagramme der UML) verwendet.

Im Informatik-Unterricht der Sekundarstufe II werden Struktogramme verwendet, damit Schüler den Aufbau logischer Abläufe, die für die Programmierung nötig sind, trainieren können. Die Erstellung von Struktogrammen aufgrund von Beschreibungen betrieblicher Problemstellungen, die wegen wiederkehrender gleicher Vorgehensweise automatisiert werden können, ist immer noch Bestandteil vieler schulischer Abschlussprüfungen.

[Wikipedia-Seitentitel: Nassi-Shneiderman-Diagramm
Datum der letzten Bearbeitung: 6. Oktober 2024, 09:49 UTC
Versions-ID der Seite: 249174001
Permanentlink: https://de.wikipedia.org/w/index.php?title=Nassi-Shneiderman-Diagramm&oldid=249174001
Datum des Abrufs: 9. Oktober 2024, 16:11 UTC

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

Objektorientierte Implementierungsvorbereitungen

Das Bild auf der linken Seite zeigt einen anderen Ausschnitt der Darstellung der „freien Künste" an der Ratslaube des Lemgoer Rathauses. Wir sehen die „GEOMETRIA", die nicht einfach eine Weiterentwicklung der Arithmetik ist, sondern ein eigenes Gebiet der Mathematik darstellt. In der analytischen Geometrie, die beispielsweise die Berechnung von Kegelschnitten zum Thema hat, finden die beiden Gebiete zusammen. Ein anderes Beispiel zu dieser komplexen Beziehung: Der große Satz von Fermat[1] (1607 – 1665), der auf einer einfachen Exponentialgleichung beruht, konnte erst über 350 Jahre später (1994) durch Andrew Wiles und seinen Schüler Richard Taylor mit Hilfe der Theorie elliptischer Kurven bewiesen werden.

1967 entstand mit den Hypothesen von Robert Langland ein neuer Forschungszweig der Mathematik, der es sich zur Aufgabe machte, „verschiedenste Bereiche der Mathematik zusammenzuführen" – so die Ausgabe 4.22 von „Spektrum der Wissenschaft", welche diesem Forschungsthema ihre Titelseite mit der Schlagzeile „Die Weltformel der Mathematik" widmete. Zwei umfangreiche Artikel berichteten im Heft von den Fortschritten auf diesem Gebiet, die das „Spektrum" so zusammenfasste: „Nun haben zwei Forscher einen enormen Fortschritt erzielt, indem sie eine Verbindung zwischen der Zahlentheorie und der Geometrie gefunden haben"[2].

1967 – das war zwei Jahre vor der ersten Mondlandung, für deren Bahnberechnungen damals die Rechner der IBM-7040-Reihe eingesetzt wurden, von denen jeder mit 65.536 Speicherzellen à 36 Bit ausgestattet war…

Dem Verhältnis von Arithmetik und Geometrie ähnelt das Verhältnis zwischen algorithmischer und objektorientierter Programmierung. Letztere ist nicht „besser" als erstere, sondern mit eigenen Qualitäten ausgestattet. Das hat auch Ada Lovelace erkannt. Sie formulierte das so:

> *»[Die Analytical Engine] könnte auf andere Dinge als Zahlen angewandt werden, wenn man Objekte finden könnte, deren Wechselwirkungen durch die abstrakte Wissenschaft der Operationen dargestellt werden können und die sich für die Bearbeitung durch die Anweisungen und Mechanismen des Gerätes eignen.[3]«*

In diesem Satz mischen sich Algorithmik und Objektorientierung auf eine ähnliche Weise wie in dieser Buchreihe, die nicht von einem Gegensatz zwischen den unterschiedlichen Programmiermethoden, sondern von einer friedlichen Koexistenz, ja sogar von einer – begrenzten – wechselseitigen Ineinander-Umwandelbarkeit ausgeht. Das folgende Kapitel konkretisiert diesen Gedanken. Zuvor sollten wir einen Augenblick innehalten und die visionäre Kraft von Ada Lovelace bewundern, die von „Objekten", „Operationen" und „Mechanismen" schrieb, lange bevor die Informatik diese Begriffe mit Leben füllte.

4.1 JSP objektorientiert?

Sind objektorientierte und algorithmische Programmierung nicht wie Feuer und Wasser? Bei einigen Autoren könnte man tatsächlich auf diesen Gedanken kommen. Die Vorteile der objektorientierten Programmierung gegenüber der herkömmlichen algorithmischen Programmierung werden in solch glühenden Farben geschildert, dass so mancher Programmierer und Programmiererin, der oder die hauptsächlich mit den Sprachen der dritten – zum Beispiel C, COBOL II, PL/1, Pascal – und vierten Generation – etwa NATURAL® – arbeitet, sich als Dinosaurier(in) fühlt. Beruht diese Wahrnehmung auf einem tatsächlichen, nicht aufholbaren Rückstand, der letztlich eine vollständige Hinwendung zur objektorientierten Programmierung – und zu den ihr vorausgehenden beziehungsweise sie begleitenden Entwurfschritten – zur Folge haben muss, oder handelt es sich um Übertreibungen, die zu korrigieren wären?

Endnoten: siehe Seite 95

Hier ist nicht der Ort, um diese Debatte mit dem nötigen Tiefgang zu führen. Festzuhalten bleibt jedoch zunächst, dass die Menge algorithmisch konzipierter Programme immer noch kontinuierlich wächst, obwohl die objektorientierte Konkurrenz an Boden gewinnt. Es wäre kurzsichtig, dies mit beharrlicher Fortschrittsverweigerung seitens der algorithmisch arbeitenden Programmierer(innen) zu erklären. Vielmehr hat diese Entwicklungsrichtung durchaus beträchtliche Vorteile aufzuweisen, beispielsweise die folgenden:

- Rasche Erlernbarkeit und kürzere Einarbeitungszeiten, nicht nur für professionelle Anwendungsentwickler, sondern zum Beispiel auch für Wissenschaftler
- Breite Palette algorithmischer Sprachen für vielerlei Anwendungsgebiete
- Aufbauen von Programmen aus Unterprogrammen, die in unterschiedlichen Sprachen geschrieben sind
- In Jahrzehnten ausgereifte Sprachen, Werkzeuge, Compiler und Laufzeitumgebungen
- Stabilität der Entwicklung durch Normung und Rückwärtskompatibilität
- Hohe Performance bei geeigneter Programmgestaltung
- Enge Integration mit modernen Datenbanksystemen
- Weltweite Verbreitung auf zahlreichen Plattformen

Andererseits hat es sich immer wieder gezeigt, dass die anarchische interne Datenhaltung einerseits und die allzu große Bandbreite möglicher Codestrukturen - von riesigen Monolithen bis hin zu konsequenter, kleinteiliger Unterprogrammtechnik - andererseits zwei wesentliche Schwachstellen der algorithmischen Programmierung sind. Hier setzt die objektorientierte Programmierung an, die mit ihren zentralen Konzepten der Klassen, der daraus erzeugten Objekte und der Vererbung diese Schwachstellen von vornherein vermeidet. Aus der Sicht der methodisch vorgehenden Programmstrukturierung ist dies uneingeschränkt als Fortschritt zu werten. Andererseits sind Nachteile zu verzeichnen, beispielsweise folgende:

- Hoher analytischer Aufwand, um die Definitionen der zentralen Klassen, ihrer Variablen und ihres Verhaltens vor Beginn der Programmierung hinreichend gut zu erarbeiten
- Rascher Anstieg der Komplexität mit der Folge, dass die iterative Codierung bevorzugt wird, was wiederum die Steuerung und die Arbeitsteilung in umfangreichen Projekten erschwert
- Enge Bindung an die einmal zu Beginn gewählte Sprache, deren Klassenbibliothek(en) und die jeweilige Laufzeitumgebung (Framework)
- Performancenachteile wegen des Zugriffs auf interne Daten über Methoden gegenüber dem Direktzugriff innerhalb derselben Code-Einheit oder dem Zugriff über Referenz-Parameter bei algorithmischen Unterprogrammen
- Erschwerte Erlernbarkeit und wesentlich aufwendigere Einarbeitung

Auf dieser abstrakten Ebene könnten die Pros und Kontras ohne wesentlichen Nutzen und letztlich unentscheidbar weiter diskutiert werden. Es kommt uns Systemanalytiker(inne)n und Programmierer(inne)n aber hauptsächlich darauf an, was eigentlich erreicht werden soll. Unbestreitbar sind die modernen grafischen Oberflächen und vielfältige weitere Eigenschaften heutiger Desktop-Systeme, Laptops, Tablet-Computer, Smartphones etcetera ohne objektorientierte Programmierung gar nicht vorstellbar. Ebenso unbestreitbar besteht das Rückgrat der heutigen IT ganz überwiegend aus algorithmischen Programmen. Die damit umgesetzten Datenvolumina sind teilweise so groß, dass sie ohne intensives Tuning nicht in der gegebenen Zeit zu bewältigen wären.

Es bedarf daher keiner prophetischen Gabe, um vorherzusagen, dass der Anteil der objektorientierten Programmierung gegenüber der algorithmischen Programmierung einerseits weiter zunehmen, andererseits letztere weiter eine große Rolle spielen wird. Um so erfreulicher ist es, dass die JSP-Methode für beide Entwicklungsrichtungen anwendbar ist.

Diese kühne Behauptung soll anhand des ersten JSP-Beispiels erhärtet werden. Anders als bei der algorithmischen Programmierung ist der objektorientierte Code allerdings nicht anhand der Programmstruktur mit Operationen direkt ableitbar. Es ist ein weiterer Entwurfsschritt nötig, in dem zunächst die Klassen festgelegt werden und über das Anlegen, sowie die Interaktionen der darauf beruhenden Objekte entschieden wird. Dieser Schritt richtet sich allein danach, welche Anforderungen die objektorientierte Programmierung an die Qualität dieser Ergebnisse stellt. Davon ausgehend findet anschliessend die Transformation der JSP-Programmstruktur in die objektorientierte Implementierung statt. Eine solche Realisierung wird im Band-2-A1-Kapitel „Example1 in Java" vorgestellt.

4.2 Klassen und Objekte bestimmen

Der Ausgangspunkt dieses neuen Entwurfsschritts ist die Erkenntnis, dass die Programmstruktur mit Operationen von ihrer Implementierung unabhängig ist. Nichts zwingt uns, sie algorithmisch umzusetzen. Sie kann genauso gut objektorientiert aufgefasst und in Code übertragen werden, denn sie sagt lediglich aus, was in welcher Reihenfolge geschehen soll. Wenn der objektorientierte Code genau dieselben Arbeiten wie der algorithmische Code in derselben Abfolge verrichtet, sind beide äquivalent.

Anders formuliert:

Jede Methodendefinition enthält algorithmischen Code. Die bei der OO-Programmausführung durchlaufene Gesamtstrecke dieses Codes - sei er an Objekte oder an statische Klassen geknüpft - muss der JSP-Programmstruktur folgen und die Daten in der dort definierten Operationsfolge verwenden, um eine im Sinn der JSP-Methode korrekte Implementierung darzustellen.

Deshalb ist es notwendig, zunächst die Operationen und die von ihnen bearbeiteten Daten zu betrachten, denn diese Daten sollen ja in Klassen – beziehungsweise in daraus generierten Objekten – gekapselt und die Operationen in Methoden transformiert beziehungsweise ebenfalls darin gekapselt werden.

Welche konkreten Daten verwendet das JSP-Beispiel?

a) Den jeweiligen Eingabe-String und seine Attribute (Länge, Anzahl Blanks, Anzahl lesbare Zeichen), sowie den für die Zählung verwendeten Index
b) Die Gesamtsummen dieser Attribute
c) Den jeweiligen Ausgabe-String, der eine Kopie des Eingabe-Strings und die aufbereiteten Attributwerte enthält
d) Starre Kopf- und Fußzeilen
e) Die letzte Fußzeile mit den aufbereiteten Gesamtsummen

Um die bei jeder Operation beteiligten Daten klarer identifizieren und Zusammengehörigkeiten besser erkennen zu können, bietet es sich an, die Liste der Operationen – mit Indexstart 0 wie bei C – hierfür zu erweitern:

Op.	Bedeutung	a)	b)	c)	d)	e)
1	open input file					
2	open output file					
3	close input file					
4	close output file					
5	stop					
6	read input record = in_string	X				
7	print "Text ..." line				X	

Op.	Bedeutung	a)	b)	c)	d)	e)
8	print empty line				X	
9	total_chars := 0		X			
10	total_blanks := 0		X			
11	total_len := 0		X			
12	chars := 0	X				
13	blanks := 0	X				
14	len := length(in_string)	X				
15	ind := 0	X				
16	ind := ind + 1	X				
17	chars := chars + 1	X				
18	blanks := blanks + 1	X				
19	total_chars := total_chars + chars	X	X			
20	total_blanks := total_blanks + blanks	X	X			
21	total_len := total_len + len	X	X			
22	copy in_string to output area	X		X		
23	convert string counters to counter columns	X		X		
24	print copied in_string & counter columns			X		
25	convert total counters to counter columns		X			X
26	print === line				X	
27	print "totals" & counter columns					X

Bis auf die Operationen 1 bis 5 liegen die Zuordnungen auf der Hand. Open, Close und Stopp haben mit den Daten a) bis e) nicht unmittelbar zu tun. Das stört aber zunächst nicht weiter, denn es ist vorstellbar, dass diese Operationen in der `main()`-Methode verwendet werden.

Bei diesem einfachen Beispiel sind die intuitiv naheliegenden Kandidaten für Klassen leicht erkennbar:

a) Zum Eingabestring und seinen Attributen gehören Methoden, um den String zu füllen, die Zähler zu initialisieren, den jeweiligen String zu durchsuchen, und die Gesamtsummen aufkumulieren zu lassen. Hierfür wird eine eigene Klasse `TextData` benötigt, von der jeweils nur ein Objekt zu existieren braucht. Das Einlesen des Strings (Op. 6) sollte dagegen in der `main()` stattfinden, um die Jackson-typische Vorlese- und Nachlese-Konstruktion offenzulegen.

b) Die Gesamtsummen bilden mit ihren Methoden drei Objekte einer weiteren Klasse namens `Total`. Ein Interface zwischen den Klassen `TextData` und `Total` wird benötigt, weil die Operationen 19, 20 und 21 beide Klassen betreffen.

c) Der Ausgabestring ist ein sehr einfach aufgebautes Objekt der Klasse `ListData`, dessen Bestandteile für die Operationen 22 und 23 von der Klasse `TextData` abgeholt werden müssen. Auch hierfür wird mindestens ein Interface benötigt. Die Ausgabe selbst (Op. 24) gehört in die `main()`.

d) Die drei starren Zeilen sind zwar pro forma als String-Objekte anzulegen, aber es sind keine eigenen Methoden für die Arbeit mit diesen Strings erforderlich. Daher liegt es nahe, die Operationen 7, 8 und 26 in der `main()` zu realisieren.

e) Auch für die letzte Fußzeile erscheint der Aufwand, dafür eine eigene Klasse anzulegen, als übertrieben. Es bietet sich an, diese Zeile analog zu den Listenkörperzeilen in der `main()` aufzubauen.

Die Zählerkonvertierungen aus der internen in die externe – druckaufbereitete – Darstellung werden für c) und e) benötigt. Hierfür sind keine eigenen Objekte erforderlich, sondern es genügen die String-Objekte, welche

die Programmiersprache anzulegen ermöglicht. Die Konvertierungen ähneln Unterprogrammaufrufen und sind daher als statische Methoden zu realisieren. Hierfür ist die Klasse `Conversion` vorgesehen.

Diese intuitiv naheliegenden Zuordnungen bedürfen der Erhärtung anhand der Struktogramme. Eine mögliche Zuordnung zwischen den Ein- und Ausgabe-Struktogrammen einerseits und den beispielspezifischen Klassen – in UML®-Notation – andererseits könnte dafür nützlich sein. Konstruktoren haben übrigens dieselben Namen wie ihre Klassen. Die Parameter der Methoden sind hier und in den folgenden klassenbezogenen Diagrammen nicht angegeben, um Redundanz zu vermeiden und Platz zu sparen.

Die gestrichelten Pfeile in der nachfolgenden Grafik assoziieren die abstrakten Datenelemente der Struktogramme teilweise mit den neuen Klassen. Wegen der Relation „besteht aus" gehören I2, „character", „blank" und „non-blank" zur Klasse `TextData`, „input string" und „string counters" dagegen zur Klasse `ListData`. Die gestrichelten Linien von `Conversion` zu „footer" beziehungsweise „string counters" deuten die Herkunft der ausgegebenen Ziffern an. Die gemeinsame Verwendung von JSP- und UML-Grafikelementen soll jedoch nicht bedeuten, dass es sich hier um Korrespondenzen handelt, aus denen die objektorientierte Implementierung zwingend hervorgeht. Das trifft nicht zu, wie sich gleich anschliessend zeigen wird.

Die in der „Abbildung 4.1: Relationen zwischen Klassen und Strukturblöcken" auf Seite 88 angedeuteten Methoden der Klassen reichen nicht dafür aus, die gesamte Funktionalität des Beispiels hervorzubringen. Ein – banaler – Teil der Operationen und die Iteration I1 verbleiben in der `main()`-Klasse, weil sich dafür eine komplexere Implementierung in diesem Fall nicht lohnt.

Wie ist diese Grafik zu beurteilen? Ist das tatsächlich eine sinnvolle Lösung oder doch nur ein braver Versuch, der auf einer simplen Zuordnungsschematik beruht? Speziell bei jenen Strukturblöcken, die aus der JSP-Sicht miteinander korrespondieren, also „string" und „extended string", erscheint die gezeigte Abbildung auf Klassen als recht künstlich. Insbesondere fällt auf, dass die Klasse `ListData` fast nur aus demjenigen Code besteht, der zum Zusammenbau einer Ausgabezeile erforderlich ist. Es ist daher schon bei diesem extrem einfachen Beispiel sinnvoll, die Klassen und die aus ihnen hervorgehenden Objekte primär aus der OO-Sicht zu bestimmen und sie anschließend mit denjenigen Methoden auszustatten, welche die Programmstruktur mit Operationen erfordert. Daraus ergibt sich die „Abbildung 4.2: Vereinfachte Relationen ohne ListData" auf Seite 89.

Der Wegfall von `ListData` vereinfacht das UML-Diagramm erheblich. Es fallen etliche Methoden und Interfaces weg, die letztlich nur Daten durchreichen. Anstatt pro Eingabesatz ein eigenes `ListData`-Objekt anlegen und mit Inhalt füllen zu müssen, genügt die Erweiterung um die Override-Methode `toString()`, deren Aufgabe es ist, aus den im `TextData`-Objekt gesammelten Attributen die gewünschte Ausgabezeile zu erstellen.

Damit ist der zusätzliche Entwurfsschritt anhand der Programmstruktur mit Operationen beendet. Die Klassen konnten ohne Schwierigkeiten identifiziert und die Operationen teils auf die Methoden, teils auf die `main()` verteilt werden. Flussdiagramme oder Pseudocode für einzelne Methoden sind wegen der sehr einfachen Verarbeitungsaufgaben nicht erforderlich. Somit steht der eigentlichen Implementierung nichts mehr im Weg. Diese ist im Unterkapitel „Java-Code von Example1" des Band-2-A1-Kapitels „Example1 in Java" dargestellt und anschliessend ausführlich erläutert.

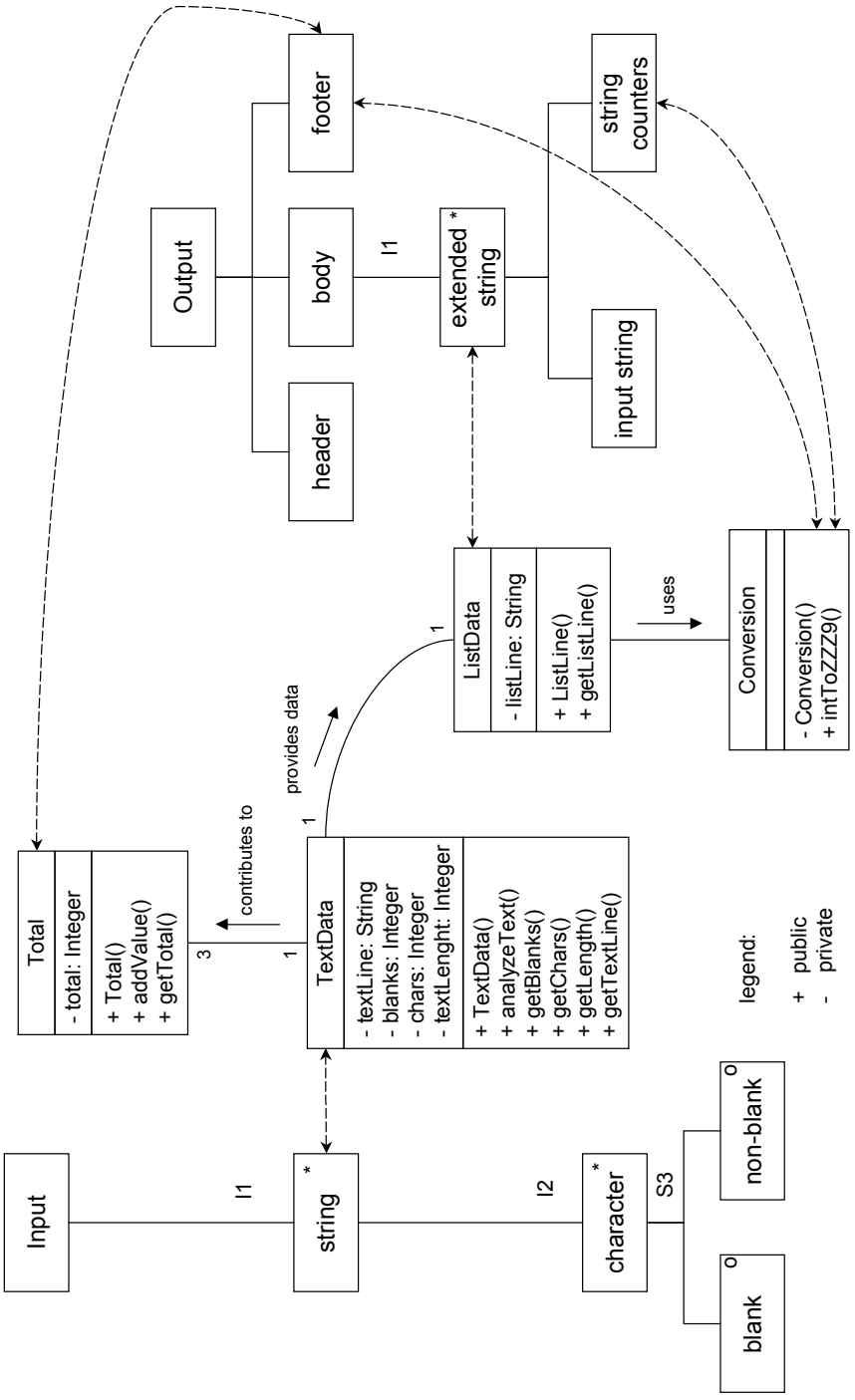


Abbildung 4.1: *Relationen zwischen Klassen und Strukturblöcken*

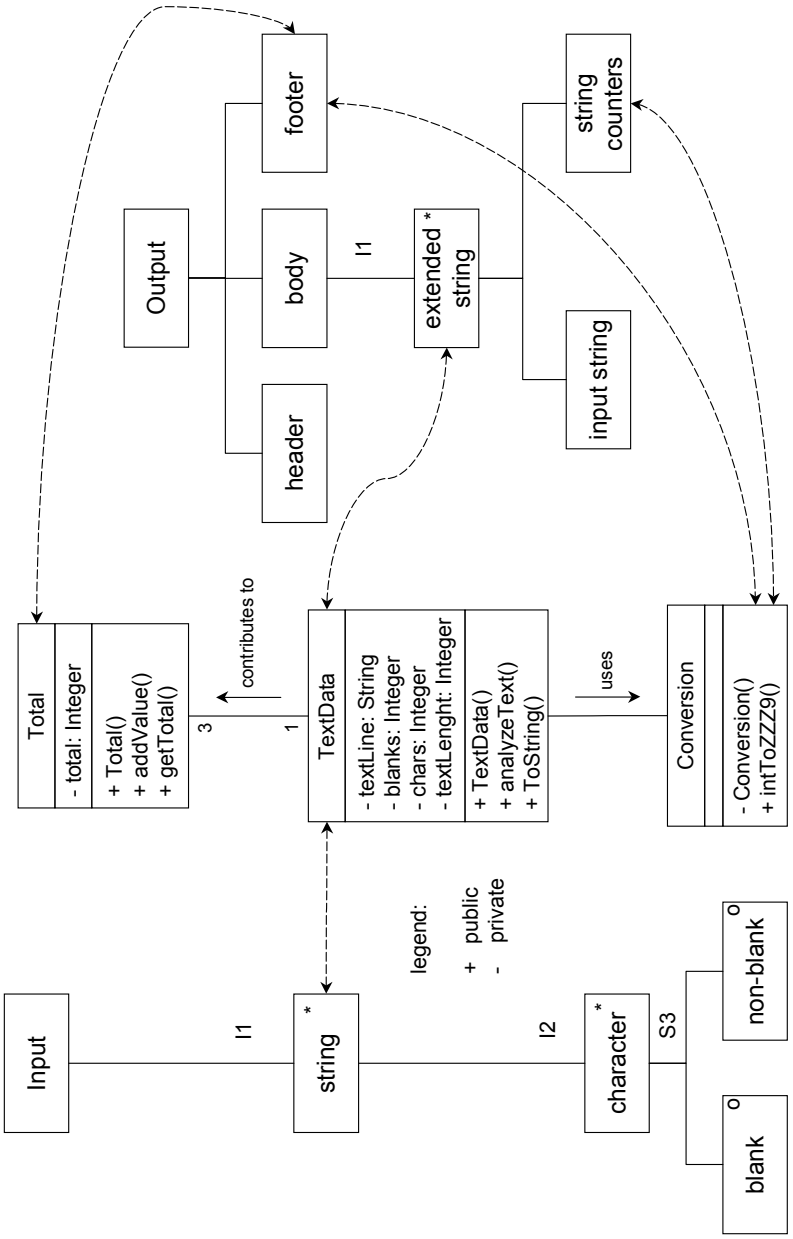


Abbildung 4.2: *Vereinfachte Relationen ohne* `ListData`

4.3 Bewertung der Ergebnisse

Die Idee, algorithmisch abgeleitete Programmstrukturen mit Operationen auf objektorientierte Weise zu implementieren, mag zunächst als befremdlich erscheinen. Diese Welten koexistieren seit vielen Jahren, sind sich aber wegen der häufigen Trennung in rein objektorientiert oder rein algorithmisch arbeitende Entwicklerteams meist nicht begegnet. Das ist bedauerlich, weil beide Seiten voneinander lernen könnten. In größeren Projekten gibt es nicht selten Bedarf für beide Richtungen. Die dafür erforderliche Kooperation scheitert jedoch an den völlig verschiedenen Entwicklungsumgebungen, Sprachen und Terminologien, wenn es niemanden gibt, der dazwischen eine Brücke zu schlagen vermag. Daher werden Aufgaben, die sich besser algorithmisch lösen liessen, von OO-Teams mühsam mit Bordmitteln bewältigt, während mancher algorithmische Programmkoloss nie entstanden wäre, hätte man das objektorientierte Potential bereits bei der Aufgabenstellung erkannt und genutzt.

Die im Band 2-A1 vorgestellte OO-Implementierung ist eigentlich ein Hybrid aus beiden Ansätzen. Die Beibehaltung der `main()` als traditionelle Steuerzentrale für den Kontrollfluss wurde gewählt, um erstens die Strukturen der Implementierungen in COBOL II, C und Java noch halbwegs vergleichen zu können, und zweitens, um unnötige Verrenkungen zu vermeiden.

Die Implementierung mit OO-Techniken weist also Freiheitsgrade auf, die dem algorithmischen Weg fehlen. Das lässt einerseits der Kreativität mehr Raum, führt aber andererseits tendenziell dazu, den ursprünglichen Entwurf beiseitezuschieben und völlig neue Wege zu gehen. Dann lohnt sich jedoch der JSP-Aufwand unter Umständen nicht mehr. Das ist der Grund dafür, dass die Implementierung mit Java sich an die Programmstruktur mit Operationen anschmiegt. Die Auswahl der geeigneten Methoden und Klassen, sowie das Erzeugen von Objekten sind dann – und nur dann – nicht willkürlich, sondern folgen der bereits gefundenen Programmlogik.

Es wäre aber verwegen, aus diesem einzigen Beispiel zu schliessen, dass damit schon alles zur OO-Implementierung auf JSP-Grundlage gesagt sei, und es keine Überraschungen mehr geben wird. Vielmehr ist zu erwarten, dass vor allem das Backtracking und die Programminversion sich nicht auf so einfache Weise objektorientiert umsetzen lassen werden.

4.4 Namenskonventionen

In dieser Buchreihe sind zahlreiche Code-Beispiele aus z.T. recht verschiedenen Programmiersprachen enthalten. Da Autoren in der IT nebenbei auch Vorbilder in Sachen Normsetzung / Normeinhaltung zu sein haben und es Rezensenten gibt, die Bücher über das Programmieren wegen inkonsequenter Namenshandhabung verreissen, ist irgendwann der Zeitpunkt gekommen, hierzu Farbe zu bekennen. Sie werden schon gemerkt haben, dass die Einhaltung fremder oder selbst erfundener Vorschriften in dieser Buchreihe nur dazu dient, eine sinnvolle Einheitlichkeit zu ermöglichen. Hier wird aber nicht gepredigt, welche Regeln Sie im Detail einhalten müssen, um Code in irgendeiner Sprache zu produzieren. Zu diesen Regeln gehören auch die Namenskonventionen für Variable, Konstante, Programme, Unterprogramme, Methoden, Klassen und was sonst noch in diesem großen Zoo existiert. Wie sieht es damit in der sogenannten Realität aus?

In der algorithmischen Welt haben sich solche Namenskonventionen kaum durchsetzen können. Es gibt wahrscheinlich in jeder größeren IT-Abteilung irgendwelche Namenskonventionen für die Programmierung, aber wohl kaum jemals in zwei Firmen dieselben. Das ist auch nicht weiter tragisch, es sei denn, es käme zur Fusion zweier hierin sehr konträr „aufgestellter" Firmen. Gewöhnlich wird ja bei Fusionen die IT des unterlegenen Parts kannibalisiert, aber es soll auch vorkommen, dass nicht die Macht, sondern die Vernunft siegt. Dann wird die fusionierte Firma aus beiden Programmportfolios die nützlichsten Komponenten auswählen. Anschliessend ist es zugleich nötig und sinnvoll, eine gemeinsame Namenskonvention zu entwickeln und diese schrittweise umzusetzen. Es ist allerdings ziemlich mühsam und deshalb aufwendig, bestehenden Code neuen Konventionen anzupassen. Daher ist es eher wahrscheinlich, dass ein Patchwork von Programmen

entsteht, die trotz gleicher Programmiersprachen recht verschieden mit Benennungen umgehen. Größere Änderungen und Neuentwicklungen setzen dann auf den neuen Konventionen auf, wodurch eine dritte Namenswelt entsteht. Das kann ziemlich verwirrend werden, vor allem bei umfassenden Anpassungen wie etwa der Erweiterung bestehender numerischer Schlüssel in alphanumerische.

In solchen Fällen zeigt sich eine Schwäche vieler Namenskonventionen: Diese geben zwar vor, was groß oder klein, zusammen oder – durch sprachspezifische Zeichen – getrennt zu schreiben ist, welche Zeichen nicht erlaubt oder welche unbedingt zu verwenden sind, aber sie enthalten meist keine Kataloge verbindlich zu verwendender zentraler Begriffe beziehungsweise Namen. Nur selten gibt es sorgfältig gepflegte, firmenweit gültige Datenmodelle mit Attributbezeichnungen, an die sich alle Programmierer(innen), Datenbankadministrator(inn)en und Schnittstellendesigner(innen) halten müssen.

Teure Formatänderung

Vor etlichen Jahren platzte die seit Jahrzehnten numerische und 6-stellige deutsche Wertpapierkennummer aus allen Nähten. Da die Änderung der Länge ein gigantischer Akt gewesen wäre, wurde stattdessen der Wertebereich um Großbuchstaben erweitert. Nun mussten unter anderem nahezu alle Programme, welche diesen Schlüssel verwendeten und ihn als numerisch definiert hatten, geändert werden. In manchen Fällen mag diese Umstellung für das Programm irrelevant gewesen sein; dann war eine Anpassung nicht zwingend erforderlich, aus Gründen der Konsistenz aber empfehlenswert. Das größte Problem bei einer solchen Umstellung ist, dass die umzustellenden Deklarationen innerhalb desselben Programmportfolios in vielerlei Namensvarianten auftreten können, zum Beispiel folgende:

```
WKN / W-K-N / W_K_N / WK-Num / WK_Num / WKNr
WP-KENN / WP_KENN / WPK
WertPapierKennNr
m_wpkn
GSK       (German security key)
```

Eine firmeneinheitlich vorgeschriebene Bezeichnung, die allenfalls um einen fallspezifischen Präfix oder Suffix ergänzt werden darf, hätte hier vielen Banken und Finanzinstitutionen erhebliche Aufwendungen erspart.

Wozu sind Namenskonventionen dann überhaupt da?

Bevor wir uns den objektorientierten Sprachen zuwenden, sei die Frage gestattet, wozu es überhaupt Namenskonventionen gibt, wenn sie ein solches Durcheinander doch nicht verhindern können. Der Regelungsbedarf entstand vor vielen Jahren mit der Einführung „sprechender Namen" in den Programmiersprachen der zweiten und dritten Generation, und er besteht weiterhin.

Hinweis: Um nicht das Thema zu verfehlen, müssen wir hier die technischen Namen für Programme, Datenbankobjekte (Tabellen, VIEWs, Indizes, …), System-Kontrollblöcke etcetera ausklammern. Diese Namen dürfen oft nur nach strengen Normen vergeben werden. Sie sind häufig nur 6- bis 8-stellig und werden in großen Organisationen nach akribischen Vorschriften verwaltet, die uns aber hier nicht interessieren.

Sprechende Namen dienen der Mnemotechnik, sollen also die Programmierer(innen) darin unterstützen, die Bedeutungen der von ihnen oder Anderen geschaffenen Bezeichnungen beim Codieren richtig zu berücksichtigen. Mit „Bedeutung" ist dabei meist sowohl die fachliche als auch die technische Komponente gemeint. Daher beinhalten viele Namen eine Formatangabe *und* eine inhaltliche Bezeichnung. Die Formatangabe soll unsinnige Verwendungen oder inkompatible Operanden von vornherein erkennbar machen:

```
if (iApfel == sBirne) …
```

mit i = Integer und s = String ist Unsinn – aber das merkt spätestens der Compiler oder Interpreter auch. Ausserdem würde ein Upgrade von iApfel in ein Unsigned-Long-Long-Integer-Format dazu zwingen, überall iApfel in ullApfel zu ändern, obwohl dadurch nichts wirklich besser würde. Daher tendieren viele Programmierer(innen) dazu, inhaltsbestimmende Formatangaben bei der Wahl der Namen zu berücksichtigen. Unsinnige Operationen und Vergleiche fallen dann schneller auf. Beispiel:

```
if (ApfelZaehler == BirnenArt) …
```

Hier kann das physische Datenformat von `ApfelZaehler` innerhalb der für Zähler vorgesehenen Varianten verändert werden, ohne dessen Namen anpassen zu müssen. An die Stelle des Formats tritt also eine Verwendungsbezeichnung. Namen wie i, ind, count, len, length und so weiter sind so verbreitet, dass die meisten Programmierer(innen) davon ausgehen, ihr physisches Äquivalent müsse ein Integer-Format oder beispielsweise auf IBM-Hosts ein Packed Decimal-Format sein. Bei den fallspezifischen Variablen etcetera erscheint es dagegen als sinnvoll, deren Typ in der Bezeichnung so zu berücksichtigen, dass eine unmittelbare Bindung an ein physisches Format vermieden wird, wie soeben gezeigt. Namen wie A1, I15QN, N754_FCB sind nicht „sprechend" und sollten daher nur dann verwendet werden, wenn sich ihre Bedeutung aus dem Kontext oder aus einem Kommentar zweifelsfrei ergibt.

Wenn das alles so evident ist, wozu dienen Namenskonventionen für programminterne Datenelemente, Unterprogrammnamen, SECTION-Namen – bei COBOL – etcetera eigentlich? Behindern solche Vorgaben nicht die Programmierer(innen) bei der Findung passender Namen? Ein Teil der schöpferischen Tätigkeit beim Programmieren ist es ja gerade, gute Namen auszuwählen, um das jeweilige Programm dadurch für sich und andere verständlich(er) zu machen. Wird dieser kreative Aspekt durch Namenskonventionen zerstört?

Nun sollten wir das Kind nicht mit dem Bad ausschütten. Es kommt doch darauf an, wie viele sinnvolle Regelungen und wie viele Freiheitsgrade in einer Namenskonvention kombiniert werden:

- Es gibt absolut öde Programmiervorschriften, in denen Großschreibung befohlen und bis ins Detail festgelegt wird, wie die Namensvergabe zu erfolgen hat, womöglich noch mit einer genauen Vorgabe der einzuhaltenden Positionen in den Code-Zeilen. Die Programme, die anhand solcher Vorschriften erstellt werden, haben eine hohe äußerliche Einheitlichkeit um den Preis der Verständlichkeit. Kreative Namensideen haben hier keine Chance. Die Verfechter solch rigider Vorschriften finden es gut, wenn nur solche Programme die Produktionszulassung erhalten können, deren Normkonformität durch eine automatische Prüfung vor dem Check-In sichergestellt wird.
- Angemessener erscheint es dagegen, das Hauptgewicht auf die Verständlichkeit zu legen. Namen, die den angestrebten Zweck knapp und genau bezeichnen, helfen beim Codieren und bei der Einarbeitung in ein fremdes Programm enorm. „Knapp" heisst hier: Allzu lange Namen sind zu vermeiden. Das kann von Fall zu Fall etwas mühsam sein.
  - Zusätzlich sollte geregelt sein, wie verwandte Namen zu behandeln sind, zum Beispiel Indizes. In dieser Buchreihe werden Array-Indizes oft durch Kleinschreibung des Array-Namens und Anhängen von `_i` / `-i` (Bindestrich bei COBOL) benannt. Dadurch lässt der Name sofort erkennen, für welchen Array er angewendet werden soll. So wird auch eine Inflation nichtssagender – kreuz und quer verwendeter – Index-Namen vermieden.
  - Dagegen können beispielsweise Schleifenvariable innerster Schleifen mit `i` benannt werden, ohne dass dadurch die Verständlichkeit leidet oder eine Gefahr geschaffen wird.
  - Analog gilt dasselbe Verständlichkeitsprinzip auch für Prozeduren beziehungsweise COBOL-SECTIONs etcetera. Es spricht auch nichts dagegen, eine SECTION-Hierarchie durch vorangestellte Nummern zu

dokumentieren, etwa `7840-Calculate-Price`. Solche Nummern können allerdings oft erst kurz vor der Fertigstellung des Codes definitiv vergeben werden, und sie verursachen gegebenenfalls einen erhöhten Wartungsaufwand.

Diese Buchreihe versucht nicht, einen strikten Standard für jede darin auftretende algorithmische Sprache einzeln zu definieren und zu beachten. Vielmehr steht immer die angestrebte Verständlichkeit im Vordergrund. Ob etwas groß oder klein, getrennt oder zusammen geschrieben wird, ist dabei sekundär. Wo es angebracht erscheint, werden auch OO-Konventionen übernommen, falls sie passend erscheinen, zum Beispiel in den C-Programmen.

OO-Namenskonventionen

Auch objektorientierte Programme unterliegen möglicherweise firmenspezifischen Namenskonventionen. Ohne hier tief zu graben, dürfen wir getrost davon ausgegehen, dass es sich um einen ähnlich bunten Flickenteppich wie bei der algorithmischen Programmierung handelt. Dennoch haben sich beispielsweise für die Java-Programmierung einige Empfehlungen etabliert, die beanspruchen, allgemein gültig zu sein:

- Klassennamen sind Substantive, keine Verben. Sie beginnen mit einem Großbuchstaben.
- Methodennamen beginnen im Allgemeinen mit einem kleingeschriebenen Verb.
- Konstante sind mit Großbuchstaben zu schreiben.
- Attributnamen – Klassen- und Exemplarvariable – werden normalerweise klein geschrieben. Der Präfix `m_` – für Member – sollte ihnen nach Meinung einiger Autoren vorangestellt werden.
- Zusammengesetzte Klassen- und Methodennamen sind mit der Kamelhöckermethode – englisch: Camel Case – aneinanderzureihen.
- Es sollen keine Unterstriche verwendet werden, ausser in Konstanten (Widerspruch zu `m_`).

Wer sich hier tiefer einarbeiten möchte, sollte sich mit der „Ungarischen Notation[4]“ von Charles Simonyi befassen. Diese Notation sieht spezielle Präfixe für Datennamen vor, deren Beachtung zwar aufwendig, aber nützlich zu sein scheint. Zitat aus Wikipedia:

»Bei der **ungarischen Notation** handelt es sich um eine von Programmierern verwendete Namenskonvention zur Wahl von Bezeichnern für Variablen und Konstanten, Funktionen und Methoden sowie anderen Objekten. Ihren Namen verdankt die ungarische Notation dem in einem (englischen) Programmtext exotisch anmutenden Aussehen der durch bestimmte Regeln zustande gekommenen Bezeichner und der ungarischen Herkunft ihres Erfinders Charles Simonyi.

Die von Simonyi entwickelte Konvention wurde bei Microsoft in der Application Group (Microsoft Office®) mit großem Erfolg angewandt und in der Folge von der Systems Group (Windows) übernommen, wobei es zu einem grundlegenden Missverständnis kam. Simonyi spricht in seinem Paper vom „type“ einer Variablen, was vielfach als „Datentyp“ interpretiert wurde. Gemeint ist vielmehr die Art der Aufgabe einer Variablen im spezifischen Kontext einer Applikation. Es geht also nicht darum, ob eine Variable eine Ganz- oder Bruchzahl speichert, sondern, ob sie einen Zähler, eine Koordinate auf dem Bildschirm, oder einen Index in einem Array repräsentiert. Aus dem Variablennamen sollte man also die Bedeutung und nicht ihren Speichertyp erschließen können.

Durch diese Doppeldeutigkeit existieren zwei Strömungen der Ungarischen Notation, das Apps Hungarian, welches die echte Notation im Sinne Simonyis ist, und das Systems Hungarian, welches durch die Fehlinterpretation der Microsoftschen Betriebssystemabteilung entstanden ist. Letzteres ist für den

schlechten Ruf der Konvention verantwortlich, weil die Benennung einer Variablen nach dem Datentyp wenig zum Verständnis des Inhalts beiträgt und trotzdem viel Aufwand verursacht.

Kern der ungarischen Notation ist es, die Aufgabe und den Typ (Apps Hungarian) bzw. nur Typ (Systems Hungarian) einer Variable (oder Methode) in deren Namen zu verdeutlichen.«

Hier sehen wir wieder ein schönes Beispiel dafür, was passiert, wenn eine gute Idee unter die Räder kommt. In dieser Buchreihe werden Sie allerdings keine OO-Beispiele finden, die hinsichtlich der darin verwendeten Namen eine – von wem auch immer propagierte – „reine Lehre" vollständig befolgen. Über diese „reine Lehre" besteht ja selbst unter den zahlreichen OO-Experten keine Einigkeit. Zudem können solche Namenskonventionen schnell zur Zwangsjacke werden, zum Beispiel die Bezeichnung von Methoden mit Verben. Es lassen sich leicht Beispiele dafür finden, dass Namen in publiziertem OO-Code nur pro forma einer Konvention gehorchen. Politisch korrektes Programmieren ist aber kein gutes Programmieren.

Die Java-Beispiele in dieser Buchreihe halten die Java-Konventionen teilweise ein. Die Lesbarkeit steht auch hier im Vordergrund, nicht das zwanghafte Aneinanderreihen von `CamelCaseWords`. Wo es sinnvoll erscheint und Schreibaufwand spart, werden Abkürzungen verwendet. Diese lassen sich öfters nur mit Unterstrichen an andere Namensteile anfügen. Warum auch nicht? Das Verbot von Unterstrichen, das auch den Präfix `m_` treffen müsste, ist nirgendwo nachvollziehbar begründet, sondern mit einiger Wahrscheinlichkeit wie viele andere unsinnige Verbote erstmals von irgendwelchen Gurus postuliert und dann von deren Anhängern gläubig übernommen worden. Die Einhaltung einer Religion lässt sich eben doch am einfachsten anhand der Befolgung leicht kontrollierbarer Ge- und Verbote nachweisen…

Endnoten

1 Der **Große Fermatsche Satz** wurde im 17. Jahrhundert von Pierre de Fermat formuliert, aber erst 1994 von Andrew Wiles bewiesen. Als schlüssiger Höhepunkt für den Beweis gilt die Zusammenarbeit von Wiles mit Richard Taylor, die sich neben dem endgültigen Beweis durch Wiles in einer gleichzeitigen Veröffentlichung eines Teilbeweises von beiden, Wiles und Taylor, als gemeinsamen Autoren niederschlug.

Der Satz besagt: Ist n eine natürliche Zahl größer als 2, so kann die n-te Potenz keiner positiven ganzen Zahl in die Summe zweier ebensolcher Potenzen zerlegt werden

[Wikipedia-Seitentitel: Großer Fermatscher Satz
Datum der letzten Bearbeitung: 28. Mai 2022, 12:32 UTC
Versions-ID der Seite: 223232400
Permanentlink: https://de.wikipedia.org/w/index.php?title=Gro%C3%9Fer_Fermatscher_Satz&oldid=223232400
Datum des Abrufs: 14. Juni 2022, 12:56 UTC]

2 Dieser Satz stammt aus der Einleitung des Artikels von Kevin Hartnett „Langlands-Programm – Die Vereinheitlichung der Mathematik" ab Seite 13ff.

Der zweite Artikel von Steve Nadis „Interview – Die Brückenbauerin" folgt ab Seite 19ff.
[Ausgabe 4/2022, Spektrum der Wissenschaft Verlagsgesellschaft mbH, Heidelberg]

3 [Quelle: siehe Wikipedia-Artikel „Ada Lovelace" –
im Kapitel „3 Algorithmische Implementierungsvorbereitungen" auf der Seite 79 zitiert]

4 [Wikipedia-Seitentitel: Ungarische Notation
Datum der letzten Bearbeitung: 30. März 2022, 08:29 UTC
Versions-ID der Seite: 221625181
Permanentlink: https://de.wikipedia.org/w/index.php?title=Ungarische_Notation&oldid=221625181
Datum des Abrufs: 14. Juni 2022, 13:20 UTC]