S E M I N A R

SS 87

METHODEN DES SOFTWARE-ENTWURFS

Allgemeine Prinzipien des Software-Entwurfs

Betreuer : M. Hagemann
am Lehrstuhl Prof. Dr. Rembold

Bearbeitung : Friedemann Kienzler

Vortrag : 22. Juni 1987

INHALTSVERZEICHNIS

Kapitel 1 : Einleitung

In dieser Seminar-Arbeit wird ein Überblick über die Entwurfsphase im Software-Lebenszyklus gegeben.

Große Software-Systeme benötigen zur Entwicklung viel Zeit, und meist sind sie dann noch viel länger in Benutzung. Innerhalb dieser Periode der Entwicklung und Benutzung lassen sich verschiedene Phasen identifizieren. Sie bilden zusammen das, was man als *Software-Lebenszyklus* bezeichnet:

1. Bedarfsanalyse und -definition,
2. System- und Software-Entwurf,
3. Implementation und Test der Komponenten,
4. System-Test,
5. Betrieb und Wartung.

Software-Entwurf kann man als einen iterativen, vielstufigen Prozeß bezeichnen, in dem jede Funktion eines Software-Systems so dargestellt wird, daß sie ohne weiteres in ein oder mehrere Programme umgesetzt werden kann.

Die Schwierigkeiten, die bei großen Software-Projekten auftreten, werden gewöhnlich nicht durch mangelnde Programmierpraxis oder unzureichendes Management verursacht, sondern durch Unachtsamkeiten im Entwurfsprozeß. Einfache Programme können in der Regel mit geringem Entwurfsaufwand entwickelt bzw. erstellt werden. Ein großes Software-Projekt hingegen kann ohne einen formalen Entwurf nicht erfolgreich realisiert werden.

In den folgenden Kapiteln werden einige aktuelle Software-Entwurfsverfahren beschrieben. Außerdem wird in eine modulare und objekt-orientierte Entwurfsmethodologie eingeführt.

Was ist eine *Entwurfsmethodologie* ? Darunter versteht man eine Auswahl von Methoden, die, zumindest in der Theorie, einen akzeptablen Software-Entwurf liefern sollten, falls nach ihnen, angewendet auf ein spezielles Problem, exakt vorgegangen wird. Der Entwurf von Software ist ein kreativer Prozeß. Er erfordert vom Designer einen gewissen Instinkt und läßt sich meist nur iterativ über eine Anzahl vorläufiger Entwürfe erreichen. Aus diesem Grund kann eine Entwurfsmethodologie nicht als ein Rezept angesehen werden, nach dem blindlings vorgegangen werden kann, um einen guten Entwurf zu erzielen. Form und Struktur sind die Charakteristika, die dem Entwurfsprozeß durch die Entwurfsmethodologie verliehen werden. Dies wird erreicht durch Forderung des Gebrauchs von Entwurfshilfsmitteln und des Erstellens passender Dokumentation, womit man eine klare Aussage des Entwurfs und seiner Ziele gewinnen will. Durch die Dokumentation wird der Zustand des Projektes sichtbar; dadurch kann das Management den Prozeß der Software-Entwicklung besser verfolgen.

Gutes Design ist der Schlüssel zu erfolgreichem Software-Engineering. Das Entwerfen von Software läßt sich nicht aus einem Buch erlernen. Um ein guter Software-Entwerfer zu werden, bedarf es zum einen der in der Praxis gewonnen Erfahrung. Es verlangt zum anderen aber auch ein sich Halten an bewährte

Prinzipien des Software-Entwurfs. Entwurfsmethodologien wie *Top-Down-Entwurf, Bottom-Up-Entwurf, HIPO-Tabellen, Warnier-Orr-Diagramme, Jackson-Entwurfsmethodologie, Datenstrukturentwurf* und *Strukturtabellen* erleichtern den Entwurfsprozeß.

In den folgenden Kapiteln wird nun zunächst ein Überblick über diese Entwurfsmethodologien gegeben, dann werden zwei neuere, an Bedeutung zunehmendere Vorgehensweisen dargestellt, die auf den Programmiersprachen Ada und Modula-2 basieren: *objekt-orientierter Entwurf* und *modulare Software-Konstruktion.*

Die Zielsetzung des Entwurfsprozesses besteht darin, eine einheitliche, entworfene Darstellung der verlangten Software zu liefern. Das hauptsächliche Entwurfsziel ist es festzulegen, wie das System arbeiten wird. Die Gesamtarchitektur des Systems wird in der Entwurfsphase festgelegt. Vereinbarungsabweichungen werden gemacht, um verschiedenen Anforderungen zu genügen, als da sind Zuverlässigkeit, Allgemeingültigkeit, Portabilität und Benutzerfreundlicherkeit. Es bedarf üblicherweise mehrerer Detailebenen im Entwurfsprozeß, um dies zu erfüllen.

Die letzten Kapitel behandeln den Einsatz von Prototypen im Entwurfsprozeß und automatischen Werkzeugen, die zur Unterstützung im Entwurfsprozeß dienen, und einige Strategien für die Entwurfsvalidation.

Es sei noch angemerkt, daß hier nicht alle Entwurfsmethodologien beschrieben werden, sondern nur einige wenige von denen, die sich in der Praxis bewährt haben bzw. die sich voraussichtlich in der Praxis bewähren werden.

Zusammenfassend kann man sagen, daß guter Software-Entwurf wesentlicher Bestandteil von effizientem Software-Engineering ist. Das Entwurfsstadium ist der kritischste Teil des Software-Lebenszyklus. Ein gut entworfenes Software-System ist ohne Umwege implementierbar und zudem wartungsfreundlich, leicht verständlich, zuverlässig und kann verifiziert werden. Schlecht entworfene Systeme können korrekt arbeiten, aber Wartung ist wahrscheinlich aufwendig und damit teuer, Testen mag schwierig oder gar unmöglich sein, und die Software ist womöglich unzuverlässig.

Kapitel 2 : Betrachtung der Hardware

Um eine optimale Durchführung auf irgendeinem Computer erzielen zu können, ist es notwendig, die Software zurechtzuschneidern, d.h. sie an die Computer-Architektur anzupassen.

In jüngster Zeit geht der Trend in Richtung Parallelität, um komplexere Probleme lösen zu können. Die Entwicklung von Software für parallele Prozessoren erfordert Parallelismus im Entwurf, um die Vorteile der Rechnerarchitektur ausschöpfen zu können.

Der Entwurf von Software für Microcomputer erfordert die sorgfältige Abwägung der Einschränkungen, auferlegt durch begrenzten Speicher und zu einem gewissen Teil auch durch begrenzte Ausführungsgeschwindigkeit.

Kapitel 3 : Ein Überblick des Entwurfsprozesses (vor dem Einsatz von Ada und Modula-2)

Ein typischer Entwurfsprozeß besteht aus mehreren verschiedenen Phasen. Bei den nun folgenden Betrachtungen von Entwurfsmethodologien ,basierend auf Vorstufen von Ada und Modula-2,untergliedert sich der Entwurfsprozeß in den Entwurf auf höherer, mittlerer und niedriger Ebene.

3.1 Entwurf auf höherer Ebene

In dieser Phase findet die Untergliederung in die wesentlichen Teile des Systems, Module oder Untersysteme, statt, werden die wichtigsten Entscheidungen, die Struktur des Entwurfs betreffend, gefällt und die Wahl zwischen in Frage kommenden Entwurfsverfahren getroffen. Die System-Ein- und -Ausgaben, der Haupt-Datenfluß durch das System und die globalen Datenstrukturen werden an Hand der Software-Spezifikation und des Datenflußdiagrammes der höheren Ebene erkannt.

3.2 Entwurf auf mittlerer Ebene

Der Entwurf auf mittlerer Ebene ist ein iterativer Prozeß, der die Module oder Untersysteme in verschiedene Unterprogramme zerlegt. Die logischen Beziehungen von Steuerung und Daten zwischen den Komponenten werden festgelegt. Dies umfaßt die Spezifikation der Schnittstellen, der Ablaufsteuerung und die Verknüpfung der Teile des Systems mit anderen Systemen, die die Software ansteuern muß. Wartungsfreundlichkeit und Zuverlässigkeit sollten auf dieser Ebene bereits in verstärktem Maße in den Entwurf miteinbezogen werden.

3.3 Entwurf auf niedriger Ebene

Der Entwurf auf niedriger Ebene spezifiziert die Wirkungsweise der einzelnen Unterprogrammeinheiten des Systems. Der Algorithmus für jede Programmeinheit wird entworfen. Dies umfaßt die Definition von lokalen Datenstrukturen, Bezeichnernamen und eventuell lokalen Funktionen. Als Ergebnis liefert diese Entwurfsebene Pseudo-Code und Flußdiagramme.

Ein Programmierer sollte in der Lage sein, einen Entwurf auf niedriger Ebene in ein Programm umzusetzen, welches auf einem spezifischen Computer ablauffähig ist.

Kapitel 4 : Entwurfsmethodologien (vor Ada und Modula-2)

In diesem Kapitel werden kurz einige aktuelle Entwurfsmethodologien beschrieben, die vor Computersprachen wie Ada und Modula-2 entwickelt wurden.

4.1 Top-Down- und Bottom-Up-Entwurf

Top-Down-Entwurf wird auch als "*Schrittweise Verfeinerung*" [19] bezeichnet. In jedem Entwurfsschritt wird eine von mehreren Funktionen oder Tasks zerlegt in eine Anzahl von Unterfunktionen oder -tasks. Mit dem Fortschreiten des Entwurfs wird jede Komponente in eigene fundamentale Operationen verfeinert, und dieser Prozeß wird solange fortgeführt, bis der Entwurf auf der tiefsten Ebene in Pseudo-Code oder einer Programmiersprache formuliert ist. Programm- und Datenverfeinerung sollten parallel auf jeder Ebene des Entwurfs bzw. der Verfeinerung durchgeführt werden.

Jeder Verfeinerungsschritt impliziert, daß eine Entwurfsentscheidung gefällt wurde. Bei diesen Überlegungen spielen Entwurfskriterien wie Effizienz, Speicherressourcen, Leserlichkeit und Modularität eine wichtige Rolle. Oft kommt es vor, daß erst in tieferen Ebenen des Entwurfs Fehler, herrührend aus höheren Ebenen, erkannt werden, die es dann gilt, mit möglichst geringem Aufwand zu beheben. Eine systematische Vorgehensweise besteht etwa darin, zu einer höheren Entwurfsstufe zurückzugehen und die notwendigen Modifikationen zusätzlich zu den erneut durchzuführenden Top-Down-Verfeinerungen zu vollziehen.

Bottom-Up-Entwurf geht vom Speziellen ins Allgemeine, das System wird von unten (Maschinenschnittstelle) nach oben (Benutzerschnittstelle) entworfen. Nützliche Operationen oder Funktionen erleichtern dem Software-Entwickler die Arbeit.

4.2 HIPO-Charts

HIPO (hierarchy und input-process-output) Charts sind ein nützliches Entwurfshilfsmittel beim Top-Down-Systementwurf. Unter der HIPO-Methode versteht man ein von IBM [16] entwickeltes Entwurfsverfahren, das davon ausgeht, daß die Ausgabedaten eines Programmentwurfs eine Funktion der Eingabedaten sind; diese Funktion ist beim Entwurf zu bestimmen.

HIPO-Charts bestehen aus zwei Komponenten (siehe Anhang S.19): ein Hierarchie(H)-Diagramm und Eingabe-Prozeß-Ausgabe(IPO)-Diagramme. Das Hierarchie-Diagramm zeigt in Blockdarstellung die Beziehungen zwischen den Hauptfunktionen, Unterfunktionen und Modulen des Systems auf. Die IPO-Diagramme stellen jede Funktion und jedes Modul im H-Diagramm mit den dazugehörigen Ein- und Ausgabevariablen dar und beschreiben den Transformationsprozeß, der innerhalb der betreffenden Funktion oder des betreffenden Moduls abläuft.

Der gesamte Entwurf eines Systems nach der HIPO-Methode besteht aus einem H-Diagramm, einem globalen IPO-Diagramm, das die Eingabevariablen, Prozeßfunktionen und Ausgabevariablen für die höchste Stufe im Hierarchie-Diagramm aufzeigt, und einer Menge von detailierten IPO-Diagrammen der niedrigeren Entwurfsebenen für jeden der obersten Stufe untergeordneten Block im H-Diagramm. Steuer- und Datenflüsse werden oft durch Pfeile dargestellt.

Die HIPO-Entwurfsmethode ist auf viele verschiedenartiger Projekte erfolgreich angewendet worden. HIPO-Diagramme werden vor allem beim Entwurf auf höherer und mittlerer Ebene eingesetzt, da hierbei die Zahl der IPO-Diagramme überschaubar bleibt.

4.3 Warnier-Orr-Diagramme

Ein Entwurfshilfsmittel, welches von Warnier[17] eingeführt und von Orr [11] weiterentwickelt wurde, ist das Warnier-Orr-Diagramm (siehe Anhang S.20). Dieses Diagramm ist aufgebaut aus geschachtelten Mengen von geschweiften Klammern, gewissem Pseudo-Code und logischen Symbolen, um die Struktur des Systems aufzuzeigen.

Mittels Warnier-Orr-Diagramm läßt sich eine Entwurfsmethodologie definieren, welche als LCP (=logical Construction of Programs) bekannt ist. Bei der LCP-Entwurfsmethode wird von der Spezifikation der Eingabe- und Ausgabe-Datenstrukturen durch Warnier-Orr-Diagramme ausgegangen. Warniers Methode geht davon aus, daß das Programm lediglich eine Informationsstruktur darstellt, die durch Warnier-Orr-Diagramme dargestellt und durch eine systematische Vorgehensweise aus den Ein-/Ausgabe-Diagrammen gewonnen werden kann.

Die Darstellung von Software nach Warnier und Orr kann mühelos in eine konventionelle Flußdiagramm-Darstellung umgesetzt werden (siehe Anhang S.21).

4.4 Die Jackson-Entwurfsmethodologie

Die Jackson-Entwurfsmethodologie [8,9] versucht, Datenstruktur in Programmstruktur zu transformieren. Jackson definiert eine Datenstruktur-Notation, die einem hierarchischem Diagramm gleicht. Die Methodologie besteht aus einer Menge von Abbildungen und Transformationen, die auf die verschiedenen Datenstrukturen angewendet werden, um eine Programmstruktur zu erhalten.

Jacksons Strategie beginnt mit der Darstellung der Datenstruktur in der eigens definierten Notation (siehe Anhang S. 22). Dem schließt sich in der nächsten Stufe eine Abbildung der Datenstruktur auf eine Prozeßhierarchie an. Das Hauptaugenmerk bei Jacksons Entwurfsmethodologie ist auf die Entwicklung der Prozeßhierarchie gerichtet. Jedoch kann diese Hierarchie auch noch in eine prozedurale Darstellung des Programms in Form von Pseudo-Code umgesetzt werden. Die prozedurale Darstellung nach Jacksons Methode bildet im wesentlichen den Entwurf auf niedriger Ebene oder den Detailentwurf.

4.5 Datenfluß-Entwurf und Struktur-Charts

Datenfluß-Diagramme beschreiben den System-Entwurf auf einer sehr hohen Abstraktionsebene. Sie dokumentieren, wie die Eingabedaten in die Ausgabedaten transformiert werden, wobei jeder Schritt im Diagramm eine bestimmte Transformation darstellt. Der Datenfluß-Entwurf versucht, den Informationsfluß in den Entwurfsprozeß miteinzubeziehen.

Die Datenfluß-Entwurfsmethode ist besonders dann nützlich, wenn noch keine Datenstrukturen festgelegt worden sind oder es Probleme ohne formale Datenstrukturen zu lösen gilt.

Einer der grundsätzlichen Vorteile der hierbei verwendeten Entwurfsnotation, der Datenfluß-Diagramme, liegt darin, daß diese die Transformationen von Eingabe- zu Ausgabedaten darstellen, ohne auf deren Implementation vorzugreifen. Ein Datenfluß-Diagramm läßt sich am ehesten erstellen, indem man sich von den Eingaben des Systems zu den Ausgaben hinarbeitet.

Komponenten, aus denen sich Datenfluß-Diagramme zusammensetzen, sind (siehe Anhang S.23): beschriftete Pfeile, welche den Datenfluß anzeigen, beschriftete Kreise, die den Transformationen entsprechen, und die Operatoren * und +, die zur Verknüpfung von Pfeilen dienen. Zur generellen Struktur eines Datenfluß-Diagrammes gibt es keine Regeln; die Erstellung eines solchen Diagramms gehört zu den kreativen Aspekten des Systems und ist auch ein iterativer Prozeß, bei dem sich das endgültige Diagramm durch Verfeinerung erster Versuche ergibt.

Wesentlicher Bestandteil des Datenflußentwurfs bildet das Festlegen einer Abbildung vom Datenflußdiagramm (DFD) auf die Software-Struktur. Diese Methode erkennt zwei verschiedene Datenflußtypen: *Transformation* und *Transaktion*. Transformationsfluß ist charakterisiert durch einen Fluß entlang ankommender Pfade (als afferent bezeichnet), einen Übergangskern (= Transformationszentrum) und einen Fluß entlang fortführender Pfade (als efferent bezeichnet). Wesentliches Merkmal des Transaktionsflusses ist eine einzelne Dateneinheit, welche andere Datenflüsse entlang einem von mehreren Pfaden auslöst.

Die Datenflußentwurfsmethode beginnt mit einer genauen Überprüfung des Software-Anforderungskataloges, um genügend Einzelheiten für einen vorbereitenden Entwurf der Software-Struktur zu erhalten. Als nächstes werden die Datenflußdiagramme daraufhin untersucht, ob Transformations- oder Transaktionsfluß vorliegt; große Systeme weisen in der Regel beides auf. Bei Transformationsfluß müssen die Bereiche (Afferent-,Transformations-,Efferentbereich) festgelegt, bei Transaktionsfluß die Transaktionszentren bezeichnet werden.

Der Abbildungsprozeß von dem Datenfluß-Diagramm zur Software-Struktur verlagert sich auf eine Transformations- oder Transaktionsanalyse, abhängig vom Datenflußtyp.

4.5.1 Transformationsanalyse

Die Transformationsanalyse (siehe Anhang S.23/24) untergliedert sich in mehrere Schritte, in denen ein DFD mit Transformationsfluß-Charakter in eine vordefinierte Schablone für die Software-Struktur umgesetzt wird.

Zunächst müssen Afferent- und Efferentbereichsgrenzen spezifiziert werden. Hierbei bewirken interpretative Einflüsse eine gewisse Mehrdeutigkeit bezüglich der Einstellung der Grenzen. Alle Objekte zwischen diesen Grenzen bilden den Transformationsast des DFD. Der nächste Schritt besteht darin, eine Faktorisierung der Software-Struktur auf der ersten Ebene durchzuführen. Ziel ist es, das DFD in einen Struktur-Chart überzuführen. Dabei werden abstrakte Transformationen in eine Hierarchie von Programm-Einheiten umgewandelt, was einen wichtigen Schritt auf dem Weg von der abstrakten Problem-Lösung zur konkreten Realisation darstellt. Struktur-Charts stellen das Programm-System als Hierarchie seiner Teile dar, die graphisch (als Baum) repräsentiert wird, wobei die Beziehungen (Kontrollfunktionen) zwischen den Programm-Einheiten dokumentiert werden, ohne irgendwelche Informationen über die Reihenfolge der Aktivitäten dieser Einheiten zu beinhalten. Nach diesem ersten Faktorisierungsprozeß ergeben sich ein Kontroll-Modul für afferente Flüsse, das alle einströmenden Daten koordiniert, ein Transformations-Kontroll-Modul, welches die Operationen der

zentralen Transformationen überwacht, ein Kontroll-Modul für efferente Flüsse, das die Ausgabe koordiniert, und ein System-Kontroll-Modul, das als übergeordnete Einheit dieser drei Kontrollfunktionen fungiert.

In den folgenden Schritten wird der Faktorisierungsprozeß für die Einheiten der ersten Ebene solange wiederholt, bis alle Kreise des DFD in dem Struktur-Chart repräsentiert sind. In den seltensten Fällen liegt hier eine 1:1-Abbildung vor. Oftmals werden mehrere Kreise des DFD durch ein Modul repräsentiert oder ein einzelner Kreis muß in mehrere Module aufgesplittet werden.

Ziel ist die Entwicklung eines Entwurfs mit *stark kohäsiven* und *lose gekoppelten* Programm-Einheiten. Eine Programm-Einheit wird als stark kohäsiv bezeichnet, wenn ihre Elemente einen hohen Grad funktionaler Verwandtschaft zeigen. Dies bedeutet, daß jedes Element der Programmeinheit essentiell zur Ausführung der von dieser Einheit wahrzunehmenden Aufgabe beiträgt. Lose gekoppelte Systeme bestehen aus nahezu unabhängigen Einheiten. Der Vorteil stark kohäsiver und lose gekoppelter Systeme besteht darin, daß Programm-Einheiten leicht und fast ohne Änderungen des restlichen Systems modifiziert oder gar durch äquivalente Einheiten ersetzt werden können.

4.5.2 Transaktionsanalyse

Der Vorgang der Umsetzung des Datenflußdiagramms mit primär Transaktionscharakter in einen Grobentwurf für die Software-Struktur untergliedert sich wie bei der Transformationsanalyse in mehrere Schritte (siehe Anhang S.25/26).

Zunächst muß das Transaktionszentrum lokalisiert werden. Dies bereitet gewöhnlich keine Schwierigkeiten, da das Transaktionszentrum leicht zu erkennen ist als Ausgangspunkt mehrerer Informationsfluß-Pfade, die radial von diesem wegführen. Der Eingabe-Pfad zum Transaktionszentrum hin- und alle Aktionspfade von selbigem wegführend müssen als solche erkannt und bezeichnet werden. Der Transaktionsfluß kann in eine Software-Struktur überführt werden, welche einen Daten empfangenden und einen verarbeitenden Ast aufweist. Unterstrukturen des verarbeitenden Astes kontrollieren alle auf der Transaktion basierenden Prozesse. Faktorisierung auf der ersten Ebene liefert einen Eingabe-Ast, dessen Struktur mittels Transformationsanalyse ermittelt wird, und einen Verarbeitungs-Ast, dessen Struktur von den Datenfluß-Charakteristiken der jeweiligen Pfade bestimmt wird.

In weiteren Schritten werden die Strukturen eines jeden Aktion-Pfades durch Transformations- oder Transaktionsanalyse, abhängig vom Datenfluß-Typ, und die des Eingabe-Astes durch Transformationsanalyse verfeinert.

4.6 Strukturierte Programmierung

Strukturierte Programmierung basiert auf dem Prinzip der *Schrittweisen Verfeinerung* (siehe Abschnitt 4.1). In seiner klassischen, für die strukturierte Programmierung grundlegende Arbeit "Goto Statement Considered Harmful" (1968) befürwortete Dijkstra [6] die Entfernung der goto-Anweisung aus allen höheren Programmiersprachen. Bei der strukturierten Programmierung werden nur die Kontrollstrukturen *Sequenz* (Ausführung eines Tasks

unmittelbar gefolgt von der Ausführung eines anderen Tasks), *Selektion* (Auswahl eines aus mehreren möglichen Tasks durch eine Bedingung) und *Iteration* (wiederholte Ausführung eines oder mehrerer Tasks bis eine vordefinierte Bedingung erfüllt wird) verwendet, denen gemeinsam ist, daß sie nur einen Eingang und einen Ausgang besitzen (=Strukturblock). 1966 wurde von Bohm und Jacopini [4] demonstriert, daß jedes Programm ohne goto formuliert werden kann, wenn die Programmiersprache die drei obigen Konstrukte zur Verfügung stellt.

Wo liegen die Unterschiede zwischen Top-Down-Entwurf und strukturiertem Programmieren?

Top-Down-Entwurf ist eine Technik, um ein Problem zu zerlegen, unabhängig von irgendwelchen Kontroll-Strukturen. Dies bedeutet, daß ein Top-Down-Entwurf in strukturierter oder unstrukturierter Weise implementiert werden könnte. Die Vorteile des strukturierten Programmierens bestehen darin, daß strukturierte Programme im allgemeinen eine klare und logische Kontrollstruktur aufweisen, wodurch der Entwurf leicht zu verstehen wird. Programmierer, die diese Methode praktizieren, gelten auch als produktiver. Die Verfügbarkeit verschiedener Konstrukte höherer Programmiersprachen erleichtern die Konstruktion zuverlässiger und wartungsfreundlicher Software. Diese Konstrukte unterstützen die Abstraktion der Kontrolle und der Daten. Die Möglichkeiten zur Daten-Abstraktion befreien den Programmierer von dem Wissen um die interne Darstellung der Daten im Computer, und die Kontroll-Abstraktion erlaubt es, den Kontroll-Fluß in einem Programm bequem zu steuern.

4.7 Methoden für den Entwurf auf niedriger Ebene

Viele der in den vorangegangenen Kapiteln behandelten Entwurfsmethoden werden auch dafür eingesetzt, den Entwurf auf niedriger Ebene zu unterstützen oder zu dokumentieren. In diesem Abschnitt werden Methoden beschrieben, die ausschließlich auf der niedrigen Ebene Anwendung finden: *Flußdiagramme, Nassi-Shneiderman-Diagramme* und *Pseudo-Code.*

Flußdiagramme (siehe Anhang S.27) stellen den Programmablauf und insbesondere die alternativen Programmpfade unter Verwendung fest definierter Symbole dar. Dabei sind die am häufigsten gebrauchten Symbole Rechtecke, die einen Ausführungsschritt darstellen, Rauten, welche Entscheidungspunkte anzeigen, und Pfeile, die den Kontroll-Fluß signalisieren. Es gibt zwei Arten von Flußdiagrammen: Macro-(High_Level-)Flußdiagramme und detailierte Micro-Flußdiagramme. Micro-Flußdiagramme können oftmals direkt in Code umgesetzt werden; direkt heißt, daß eine 1:1-Abbildung zwischen jedem Symbol im Flußdiagramm und jeder Zeile Code vorliegt. Als Mittel zur leserlichen Dokumentation sind derartige Flußdiagramme unbrauchbar. "Richtiger" Gebrauch von (Macro-)Flußdiagrammen liefert eine Hierarchie von High_Level-Flußdiagrammen. Ausgehend von einem Macro-Flußdiagramm, welches die Module und Unterprogramme bezeichnet, werden beim Entwurfsprozeß High_Level-Flußdiagramme für jedes wichtigere Modul oder Unterprogramm entwickelt.

Das Nassi-Shneiderman-Diagramm [10] (siehe Anhang S.27) ist eine graphische Darstellung der Strukturblöcke (siehe Abschnitt 4.6), die unstrukturiertes Programmieren nicht zuläßt. Grundsätzlich werden die Strukturblöcke als Rechtecke

dargestellt. Diese können wiederum Strukturblöcke enthalten. Auf diese Art und Weise kann ein Algorithmus nur unter Verwendung dieser Strukturblöcke dargestellt werden.

Ein weiteres Hilfsmittel zur Formulierung eines Algorithmus' nach den Richtlinien der strukturierten Programmierung ist Pseudo-Code, eine verkürzte Notation für Kontrollstrukturen und andere Elemente von Programmiersprachen. Dabei werden die Strukturblöcke nicht graphisch dargestellt, sondern in Anlehnung an höhere Programmiersprachen verbal beschrieben. Eine genaue Syntax ist nicht festgelegt. Ist ein Algorithmus in Form eines Pseudo-Codes gegeben, so kann man diesen - gegebenenfalls unter zusätzlicher schrittweiser Verfeinerung - relativ einfach in eine beliebige Programmiersprache umsetzen. Der Vorteil von Pseudo-Code gegenüber Fluß- oder Nassi-Shneiderman-Diagrammen liegt gewöhnlich bei größerer Flexibilität, da Pseudo-Code mit Hilfe eines Texteditors leichter zu erstellen, zu modifizieren, zu reproduzieren und zu verfeinern ist.

4.8 **SADT** (= Structured Analysis Design Technique)

Als Spezifikations- und Analyse-Methode ermöglicht SADT, die individuellen Anforderungen vieler Benutzer zu sammeln und zu integrieren. SADT kann auch für den Entwurf eingesetzt werden, unterscheidet sich aber gänzlich von den bisher beschriebenen Methoden. SADT versucht ein Problem durch wiederholende Aufteilung in kleinere Teil-Probleme anzugehen, bis das Problem schließlich vollständig verstanden ist. Eine Umsetzung in ein Programm bzw. Programm-Module findet nicht statt.

SADT stellt ein nützliches Analyse-Werkzeug für ein gestelltes Problem dar. Seine Anwendung sollte vor dem Entwurf auf höherer Ebene erfolgen.

Kapitel 5 : **Modularer und objekt-orientierter Entwurf unter Benutzung von Ada und Modula-2**

Mit der Entwicklung der Software-Engineering-Sprachen Ada und Modula-2 fand eine neue Entwurfsmethode, der modulare und objekt-orientierte Entwurf, starke Verbreitung.

Jeglicher Software-Entwurf basiert auf dem Prinzip der Abstraktion. Die Idee bzw. das Problem wird als abstrakte Einheit ohne Details über die konkrete Realisierung dieser Einheit betrachtet.

Zu Beginn der Aera der Software-Entwicklung etwa zwischen 1940 und 1950 war der Software-Entwurf gleichzusetzen mit einer Umsetzung des gestellten Problems in 0/1-Sequenzen der zu programmierenden Maschine. Abstraktionsdenken war kaum möglich bzw. nötig.

Ende der 50er und Anfang der 60er Jahre wurde mit der verbreiteten Anwendung von höheren Programmiersprachen wie FORTRAN, ALGOL und COBOL der erste große Schritt in Richtung Software-Abstraktion gemacht. Es konnten nun Objekte und Operationen, welche innerhalb der Problemstellung auftauchten, mit in diesen ersten höheren Programmiersprachen verfügbaren vordefinierten Daten- und Kontrollstrukturen ausgedrückt werden.

Mit Einführung von Sprachen wie PASCAL in den 70er Jahren wurde für die Programm-Entwickler die Palette der grundlegenden Daten- und Kontrollstrukturen noch reichhaltiger. Der Software-Entwurf wurde damit immer mehr zum Abstraktionsprozeß.

Das Problem, das ein auf die Darstellung mit deratigen Konstrukten basierender Entwurf mit sich bringt, besteht darin, daß die Zuverlässigkeit und Wartungsfreundlichkeit von Software gefährdet sein kann, wenn die Vollständigkeit und Korrektheit der Software von einer speziellen Objekt-Darstellungsweise abhängig ist. Falls diese für die Darstellung gültigen Details etwa zur Anpassung an eine neue Umgebung (z.B. neuer Computer oder neues Betriebssystem) geändert werden müssen, ist es in der Regel mit weniger Aufwand und mehr Zuverlässigkeit verbunden, das gesamte Software-System neu zu konzipieren, als das existierende zu modifizieren.

Mit modularem und objekt-orientiertem Entwurf ist nun der zweite bedeutende Schritt in Richtung Abstraktion innerhalb des Software-Entwicklungs-Prozesses möglich. Der System-Designer muß die Problemstellung nicht mehr umsetzen in vordefinierte Daten- und Kontrollstrukturen, die die Implementierungssprache bereitstellt, sondern er kann seine eigenen abstrakten Datentypen und funktionalen Abstraktionen einführen. Diese Umsetzungsweise vom Problem zum Entwurf scheint nachvollziehbarer und "natürlicher" zu sein. Darüberhinaus wird damit der System-Entwurf von den system-internen Darstellungsdetails für Daten-Objekte entkoppelt.

Beim objekt-orientierten Entwurf wird das System nicht als eine Anzahl von Funktionen, sondern als eine Ansammlung von Objekten betrachtet, wobei jedes Objekt mit allen anderen kommunizieren kann. Diese Kommunikation erfolgt durch den Austausch von Nachrichten, wobei eine Nachricht normalerweise eine Anweisung zur Aktivierung einer bestimmten Funktion enthält. Zu jedem Objekt gibt es einen Satz auf dieses Objekt anwendbarer Operationen.

Das *Ada-Paket* mit seinen *privaten* Typen, die *Modula-2 Module* mit den *verborgenen* Typen und die vollständige Trennung zwischen Spezifikation und Implementierung dieser Module ermöglichen den modularen und objekt-orientierten Entwurf, der auf dem Begriff der abstrakten Datentypen sowie der Idee basiert, als Grundkriterium für die Dekomposition das "Verbergen von Information" zu verwenden. Die in einem Ada-Paket deklarierten Typen heißen *privat*, wenn von außerhalb des Pakets nicht auf ihre Struktur zugegriffen werden kann. Ein Modul *verbirgt* eine Gruppe von änderungswahrscheinlichen Entwurfsentscheidungen vor dem Rest des Systems (Geheimnisprinzip).

5.1 Das modulare Entwurf-Diagramm; objekt-orientierter Entwurf

Bei einfachen Systemen lassen sich die Objekte leicht identifizieren, im allgemeinen bedarf es jedoch einer Strategie zur Erstellung eines objekt-orientierten Entwurfs.

Eine mögliche Strategie (1983 von Abbot vorgeschlagen [1]) basiert auf einer informalen, in einer natürlichen Sprache geschriebenen Beschreibung, wie das Problem anzugehen ist. Aus einer derartigen Beschreibung sucht man dann die gewöhnlichen Substantive (z.B. "Baum", "Projekt", "Text") heraus und betrachtet sie fortan als abstrakte Datentypen. Bestimmte

Substantive sowie direkte Verweise auf gewöhnliche Substantive (z.B. "das Buch", "mein Termin-Kalender") werden als Instanzen abstrakter Datentypen betrachtet. Verben und Adverben werden zur Identifizierung der Operatoren (z.B. einfügen, erzeugen) und der Attribute (z.B. leer) eines Objekts verwendet. Nach der Einteilung der Objekte und der zugehörigen Operationen wird festgelegt, welche Objekte miteinander kommunizieren müssen. Schließlich wird, so Abbots Vorschlag, jedes der Objekte durch ein Ada-Paket definiert.

Zur Veranschaulichung und zum besseren Verständnis sei nun ein Beispiel für den objekt-orientierten Entwurf und die modulare Software-Konstruktion angeführt. Die einzelnen Phasen von der Problemstellung bis zum fertigen Programm werden zum Teil nur kurz skizziert; die prinzipielle Vorgehensweise ist wichtig, aber etwa das gesamte Listing des lauffähigen Ada-Programms interessiert bei diesen Überlegungen nicht so sehr:

Das Problem

Gesucht ist der jüngste gemeinsame Vorgänger zweier willkürlicher Nachfolger-Knoten eines Binärbaums. Die Anforderungen an das Software-System sind die folgenden :

1. Aufbau eines Binärbaums, dessen Knoten Integer-Werte besitzen.
2. Eingabe der Integer-Werte für die beiden Nachfolger-Knoten ein.
3. Bestimmung und Ausgabe des Baumknotens, der der jüngste gemeinsame Vorgänger der beiden Nachfolger-Knoten ist.
4. Ausgabe einer Fehlermeldung, wenn der Benutzer versucht, im Baum nicht vorhandene Nachfolger-Knoten einzugeben.

Die informelle Lösungsstrategie lautet nun:

1. *Erstelle* einen Binär*baum* (der Typ des Baumes ist unwichtig).
2. *Lokalisiere* die Positionen der beiden Baum-Knoten, deren Integer-Werte mit denen übereinstimmen, die der Benutzer eingibt.
3. *Berechne* die *Ebene*n im Baum für jeden der beiden Nachfolger-Knoten.
4. Wenn die Ebenen nicht gleich sind, *gehe hinauf* im Baum, beginnend mit dem tiefer liegenden Nachfolger-Knoten, bis zu einem Knoten, der auf derselben Ebene liegt wie der höher liegende Nachfolger-Knoten.
5. *Gehe* von beiden Knoten(nun auf dergleichen Ebene liegend) *hinauf* im Baum entlang der beiden Äste, bis ein Zusammentreffen an einem Knoten erfolgt.
6. Gebe diesen zuletzt erreichten Knoten aus.

Bei Eingabe zweier im Baum existierender Nachfolger-Knoten ist eine Lösung garantiert. Im schlimmsten Fall bildet die Wurzel des Baums den jüngsten gemeinsamen Vorgänger zweier Nachfolger-Knoten.

Vorgehen nach der Strategie zur Erstellung eines objekt-orientierten Entwurfs:

Ausgehend von der informellen Beschreibung, wie das Problem anzugehen ist, sind der abstrakte Datentyp "Baum", die auf dieses Objekt anwendbare Operationen "Erstelle_Baum", "Lokalisiere", "Berechne_Ebene", "Gehe_hinauf" und die Objekt-Attribute "Wert", "Ist_gleich" und "Nicht_vorhanden" zu identifizieren.

Der Entwurf verwendet abstrakte Datentypen (im Beispiel "Baum"), wird aber unabhängig von irgendwelchen vorgegebenen Strukturen für diesen abstrakten Datentyp erstellt. Dies bringt aber den Nachteil mit sich, daß einfache Funktionen, auf die bei vordefinierten Darstellungsarten zugegriffen werden kann, nun nicht mehr verfügbar sind. Aus diesem Grund muß die Datenabstraktion durch eine Menge von Prozeduren, welche die notwendigen Operationen ausführen, unterstützt werden.

Der dafür notwendige Overhead rentiert sich, da das Software-System nicht mehr als Ganzes geändert werden muß, falls die verborgenen Darstellungskonventionen für die abstrakte Struktur und die darauf angewendeten Operationen sprich Prozeduren irgendwann geändert werden. Nur der Modulrumpf, der die Implementierungseinzelheiten des abstrakten Datentyps und der ihn unterstützenden Prozeduren beinhaltet, müßte dann modifiziert werden. Es ist schwierig, diese Art der lokalen Wartung mit älteren Software-Entwurfsmethoden zu erreichen.

Ein *modulares Entwurf-Diagramm* bietet einen Überblick über die Software-Systemarchitektur. Es dokumentiert die modulare Gliederung des Systems in seine Komponenten-Module. Jede Modul-Definition und jede Paket-Spezifikation wird durch ein Rechteck dargestellt. Ein Software-Bus veranschaulicht die logischen Verknüpfungen zwischen den Modulen. Dem Diagramm ist klar zu entnehmen, wie die einzelnen Komponenten voneinander abhängen. Schließlich wird auch das Haupt-Programm als Block mit seinen Verknüpfungen zum restlichen System dargestellt.

Das erste modulare Entwurf-Diagramm für das Problem des jüngsten gemeinsamen Vorgängers (siehe Anhang S.28) zeigt die Unterteilung des Systems in drei Haupt-Module:

(1) "Allgemeine_Baum-Operationen" enthält eine Menge von grundlegenden Baum-Abstraktionen, die gewöhnlich direkt von einem Programmierer unter Verwendung der in der Sprache gegebenen Datenstrukturen implementiert werden, und die drei abstrakten Funktionen "Wert", "Ist_gleich" und "Nicht_vorhanden", welche vom Hauptprogramm aufgerufen werden.

(2) "Spezielle_Baum-Operationen" enthält die Prozeduren "Erstelle_Baum", "Lokalisiere", "Berechne_Ebene" und "Gehe_hinauf". Der abstrakte Datentyp "Baum" wird vom Modul (1) eingeführt.

(3) Das Hauptprogramm-Modul muß den Typ "Baum" und die Funktionen und Prozeduren aus den Modulen (1) und (2) übernehmen.

--- Das Beispiel wird in Abschnitt 5.2 fortgesetzt ! ---

5.2 Modulare Software-Konstruktion

Ausgehend von einem ersten modularen Entwurf-Diagramm besteht der nächste Schritt bei der Erstellung eines objekt-orientierten Entwurfs aus der detailierten Spezifikation der Objekte und der entsprechenden Operationen. In Ada geschieht dies durch Verwendung von Paketen, in denen die Deklarationen von Typen, Operationen usw. zusammengefaßt werden.

Fortsetzung des Beispiels aus Abschnitt 5.1:
Bei dem Problem des jüngsten gemeinsamen Vorgängers könnte die Schablone eines ersten Modula-2-Entwurfs etwa für die Definition des Moduls "Allgemeine_Baum-Operationen" wie folgt lauten:

```
DEFINITION MODULE Allgemeine_Baum-Operationen;
  EXPORT QUALIFIED
     (* type *) Baum,
     (* proc *) Nicht_vorhanden,
     (* proc *) Ist_gleich,
     (* proc *) Wert;
  TYPE Baum;
     (* Die Darstellungseinzelheiten dieses
        abstrakten Datentyps sind verborgen *)
  PROCEDURE Nicht_vorhanden (B:Baum): BOOLEAN;
     (* Diese Prozedur gibt den booleschen
        Wert true zurück, falls der Baum-
        Knoten B nicht vorhanden ist      *)
  PROCEDURE gleich (P,Q: Baum): BOOLEAN;
     (* Diese Prozedur gibt den booleschen
        Wert true zurück, falls die Werte
        der Baum-Knoten P und Q gleich sind *)
  PROCEDURE Wert (B: Baum): INTEGER;
     (* Diese Prozedur gibt den Integer-
        Wert des Baum-Knotens B zurück    *)
END Allgemeine_Baum-Operationen.
```

Entsprechend wird das andere Definition-Modul "Spezielle_Baum-Operationen" spezifiziert.

Man erhält ein modulares Entwurf-Listing, welches compiliert werden kann, um sicherzustellen, daß die System-Integration auf höherer Ebene korrekt ist.

Um nun basierend auf dem Software-Rahmen, der auf Entwurfsebene mittels modularem Entwurf-Diagramm aufgebaut worden ist, das System implementieren zu können, d.h. die Implementation-Module zu konstruieren, sind in der Regel weitere Komponenten zu den Definition-Modulen hinzuzufügen. Ausgehend von der informellen Lösungsstrategie werden die Blöcke im Entwurf-Diagramm, welche die Implementation der in der Spezifikation angegebenen Operationen definieren, in derselben Sprache wie die Modul-Definitionsblöcke spezifiziert und können nun auf die definierten Operationen zugreifen.

Um die Implementation-Module zu den Definition-Modulen und dem Modul des Haupt-Programms spezifizieren zu

können, ist es noch notwendig, zusätzliche "Low-Level"-Operationen wie "Linker_Sohn", "Rechter_Sohn" oder "Erzeuge_Wurzel", die bei nicht objekt-orientiertem Entwurf normalerweise direkt in Termen von globalen Baum-Datenstrukturen ausgeführt werden können, in den Definition-Modulen zu spezifizieren.

Vor der endgültigen Implementierung liegt schließlich ein vollständiges modulares Entwurf-Diagramm (siehe Anhang S.28) und Entwurf-Listing vor.

Der Vorteil bei der Verwendung von Sprachen wie Ada oder Modula-2 liegt zum einen darin, daß bereits beim Entwurf diese Sprachen eingesetzt werden können, was die Implementierung eines objekt-orientierten Entwurfs erleichtert, zum anderen an der Eigenschaft dieser Sprachen, Programm-Einheiten unabhängig vom Gesamt-System erstellen, ändern, austauschen oder übersetzen zu können, ermöglicht durch modularen Aufbau und "Verbergen" von Information. Diese Methode des "Zurückhaltens von Information" gewährt jeder Programm-Einheit nur Zugang zu den Programm-Objekten, die zur Ausübung ihrer Funktion benötigt werden. Der Zugang zu anderen, nicht benötigten Objekten kann durch Verwendung von sog. Sichtbarkeitsregeln verhindert werden, um die Existenz dieser Objekte zu verleugnen. Damit wird auch erreicht, daß die verborgene Information in keinster Weise von einer "unbefugten" Programm-Einheit beeinflußt werden kann.

Zusammenfassend kann man sagen, daß der Ansatz für den objekt-orientierten Entwurf unter Verwendung einer natürlichsprachlichen Beschreibung des Systems in vielen Fällen nützlich, bei komplexen Systemen mit vielen zusammenarbeitenden Funktionen allerdings sehr schwierig zu sein scheint. Bei der informellen Beschreibung des Problems finden sich dann Fehler, Widersprüche und Auslassungen. Ein objekt-orientierter Ansatz zum Software-Entwurf auf den obersten und untersten Ebenen des Entwurfs ist wohl die natürlichste Sichtweise. Auf diesen Ebenen kann er zu stärkerer Kohärenz und schwächerer Kopplung (siehe Abschnitt 4.5.1) der einzelnen Komponenten führen, was der Wartungsfreundlichkeit des Entwurfs zugute kommt. In den dazwischenliegenden Ebenen scheint jedoch eine funktionale Betrachtungsweise passender. Dies bestätigt auch die Ansicht, daß große Software-Projekte zu komplex sind, als daß man sie sinnvoll unter der Verwendung eines einzigen Ansatzes entwerfen könnte.

Kapitel 6 : Schnelle Prototyp-Entwicklung (Rapid Prototyping)

Prototyp-Entwicklung ist eine Phase im Software-Produktionsprozeß, in der sukzessive ein Modell gebildet wird, das alle wesentlichen Leistungen bzw. Eigenschaften des Endprodukts enthält und das direkt in eine reale Produktionsumgebung überführt werden kann.

Die *schnelle Prototyp-Entwicklung* entspricht einer Komponente der Prototyp-Entwicklung, die eine sehr schnelle Modellierung einer Problemlösung erlaubt. Der erzeugte Prototyp kann hierbei nicht als End-System übernommen werden.

Warum (schnelle) Prototyp-Entwicklung ?
Die Erfahrung zeigt, daß der größte Teil der Wartungs- und damit der Gesamtkosten nicht aus Fehlern im System, sondern aus Änderungen der Bedürfnisse resultiert. Oftmals weiß der Endbenutzer selbst nicht genau im vorhinein, wie die erwünschte Schnittstelle gestaltet sein soll, wann die geforderte Software seiner Meinung nach als benutzerfreundlich gilt. Für den Benutzer ist es oft schwierig, seine Anforderungen für die Software-Entwickler in ausreichend detailierter Weise anzugeben.

Das Erstellen eines Prototyps liefert eine Möglichkeit, die tatsächlichen Bedürfnisse des Benutzers genau zu erfassen. Die Idee ist die, den Anwender so schnell wie möglich mit einem Prototyp des Systems zu konfrontieren, damit er mit diesem experimentieren und die Entwickler über seine Erfahrungen informieren kann. Der Prototyp wird daraufhin den geänderten Bedürfnissen des Anwenders angepaßt, bis der Benutzer mit dem gelieferten System zufrieden ist.

Bei der Verwendung von Prototypen offenbaren sich bei der Vorführung der System-Funktionen Mißverständnisse zwischen Entwickler und Anwender, fehlende Funktionen und fehlende oder widersprüchliche Anforderungen werden während der Entwicklung des Prototyps entdeckt.

Die beiden wichtigsten Techniken des Rapid Prototyping sind *wiederverwendbarer Code* und *ausführbare Spezifikationen*. Moderne Sprachen wie Ada und Modula-2 erleichtern die Wiederverwendung von Code dadurch, daß sie es ermöglichen, große Software-Bibliotheken aufzubauen und , falls erforderlich, auf individuelle Pakete oder Module zugreifen zu können. Das Problem bei der zweitgenannten Vorgehensweise, den ausführbaren Spezifikationen, besteht in der Wahl einer geeigneten Spezifikationssprache, die von Software-Systemen direkt gelesen und umgesetzt werden kann. Der Prototyp kann als Spezifikation für die Entwicklung eines Systems in Produktionsqualität dienen.

Bei kleinen Systemen ist es durchaus ratsam, das endgültige System aus dem Prototypen zu entwickeln. Bei großen Systemen, bei denen die Wartungsfreundlichkeit wesentlich wichtiger ist, wird es meist besser sein, den Prototypen beiseite zu legen und das System neu aufzubauen.

Das Hauptargument gegen die Verwendung von Prototypen ist, daß die Kosten für die Entwicklung eines Prototyps unerträglich hoch sind, da das Endprodukt Software nicht durch Massenproduktion wie z.B. bei der Herstellung elektronischer Bauteile diese Ausgaben zu einem geringen Bruchteil der Gesamtkosten schrumpfen läßt. Gomaa [7], der über die erfolgreiche Anwendung von APL zur Erstellung von Prototypen berichtet, schätzt die Kosten für die Entwicklung eines Prototypen für ein Prozeß-Verwaltungs- und Informationssystem auf weniger als 10% der Gesamtkosten.

Kapitel 7 : Validation des Entwurfs

Zunächst eine Erklärung der Begriffe *Validation* und *Verifikation*:

Validation und Verifikation werden oft synonym verwendet, doch es bestehen erhebliche Unterschiede zwischen diesen

Begriffen. Die wohl prägnanteste Formulierung dieses Unterschieds gelang Boehm [3]:

Verifikation: "Bauen wir das Produkt richtig ?"
Validation: "Bauen wir das richtige Produkt ?"

Kurz gesagt prüft die Verifikation, ob das in der Entwicklung befindliche Produkt der Bedarfsdefinition entspricht, und die Validation, ob die Funktionen des Produkts den Wünschen des Benutzers genügen.

Die Validation des Software-Entwurfs bildet einen wichtigen Schritt im Entwurfsprozeß. Die Korrektur unerkannter Fehler und Auslassungen, die sich durch die Entwicklungsphase des Projekts fortgepflanzt haben und bis zum Test des Systems unerkannt geblieben sind, kann einen vollständig neuen Entwurf und eine neue Implementation von System-Komponenten erforderlich machen.

Die Ziele des Prozesses der Entwurf-Validation bestehen darin, zu verifizieren, daß der Entwurf korrekt ist, und zu zeigen, daß der Entwurf den Anforderungen genügt (Gültigkeit der Software).

Der sicherste Weg zur Verifikation eines Software-Entwurfs besteht aus einem mathematischen Beweis der Korrektheit einer detailierten Entwurfsbeschreibung für jede einzelne Software-Komponente. Da diese formalen Methoden der Verifikation sehr zeit- und arbeitsaufwendig sind, sollten sie nur den kritischen Teilen des Systems vorbehalten sein. Weniger exakt, aber wesentlicher einfacher, schneller und billiger läßt sich die Korrektheit eines Programms argumentativ informal zeigen.

Der Beweis der Software-Gültigkeit läßt sich durch eine Musterung des Entwurfs erreichen. Man unterscheidet *informale* und *formale Musterung*.

Eine informale Musterung des Entwurfs besteht in der Hauptsache aus dem Vergleich der Spezifikation mit dem Entwurf der Software. Die Entwurfsarbeit des Designers wird durch einen Kollegen des gleichen Entwurfsteams begutachtet. Informale Entwurfsmusterungen sollten in regelmäßigen Abständen durchgeführt werden, um so viele Fehler wie möglich im Entwurf zu finden.

Eine formale Musterung kümmert sich mehr um die Validation der Zusammenarbeit der einzelnen Komponenten und um den Vergleich des Entwurfs mit den Anforderungen des Benutzers. Die Entwurfsarbeit eines Einzelnen oder eines Teams wird durch einen Ausschuß begutachtet, der mit Mitgliedern des Projekts sowie des technischen Managements besetzt ist. Vor der Musterung sollte allen Beteiligten die Spezifikation des Entwurfs sowie der Anforderungskatalog zur Verfügung gestellt werden.

LITERATUR

1. Abbot ,R., "Program Design by Informal English Descriptions", Comm. ACM, 26(11), 1983, pp. 882-894.

2. ACM SIGSOFT Rapid Prototyping Workshop, April 1982, published as ACM Software Eng. News, Vol. 7, No. 5, December 1982.

3. Boehm, B.W. ,*Software Engineering Economics*, Englewood Cliffs, NJ. Prentice Hall, 1981.

4. Bohm, C., and G. Jacopini, "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules", Commun. ACM, Vol. 9, No. 5, May 1966, pp. 366-371.

5. Dijkstra, E.W., "Programming Considered as a Human Activity", Proc. IFIPS Congr., 1965, pp. 213-217.

6. Dijkstra, E.W., "GoTo Statement Considered Harmful", Commun. ACM, Vol. 11, No. 3, March 1968, pp. 14 -148.

7. Gomaa, H., "The Impact of Rapid Prototyping on Specifying User Requirements", ACM Software Engineering Notes, 8(2), 1983,pp. 17-28.

8. Jackson, M.A., *Principles of Program Design*, New York: Academic Press, 1975.

9. Jackson, M.A., *System Development*, Englewood Cliffs, N.J.: Prentice-Hall, 1983.

10. Nassi, I., and B. Shneiderman, "Flowchart Techniques for Structured Programming", SIGPLAN Notices ACM, Vol. 8, No. 8, August 1973, pp. 12-26.

11. Orr, K.T., *Structured System Design*, New York: Yourdon Press, 1978.

12. Page-Jones, M., "Transform Analysis", *The Practical Guide to Structured Systems Design*, New York: Yourdon Press, 1980, pp. 181-203.

13. Page-Jones, M., "Transaction Analysis", *The Practical Guide to Structured Systems Design*, New York: Yourdon Press, 1980, pp. 207-219.

14. Schneider, H.-J., *Lexikon der Informatik und Datenverarbeitung*, Oldenbourg, 2. Auflage, 1986.

15. Sommerville, J., *Software Engineering*, Internationale Computerbibliothek, 1987.

16. Stay, J.F., "HIPO and Integrated Program Design", IBM Syst.J., Vol. 15, No. 2, 1976, pp. 143-154.

17. Warnier, J.D., *Logical Construction of Programs*, New York : Van Nostrand, 1974.

18. Warnier, J.D., *Logical Construction of Systems*, New York : Van Nostrand, 1981.

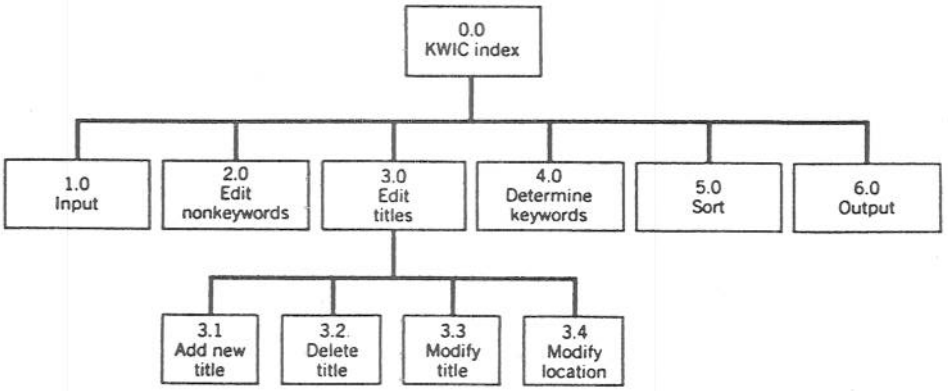
--> 19. Wiener, R., and R. Sincovec, "General Principles of Software Design", *Software Engineering with Modula-2 and Ada*, John Wiley & Sons, pp. 97-160.

20. Wirth, N., "Program Development by Stepwise Refinement", Commun. ACM, Vol. 14, No. 4, April 1971, pp. 221-227.

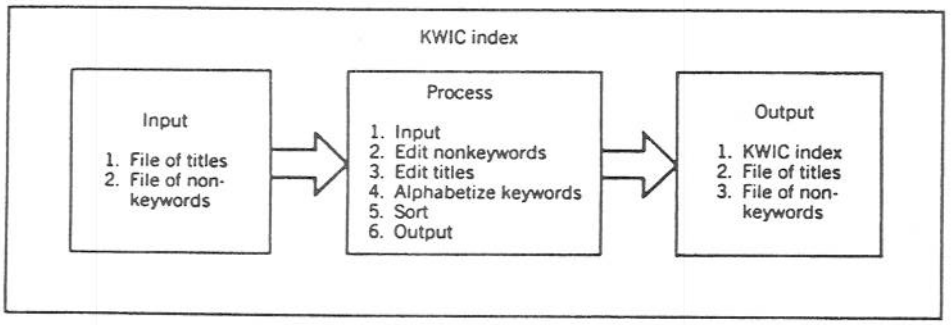
ANHANG

DIAGRAMME (illustriert an Hand von Beispielen)

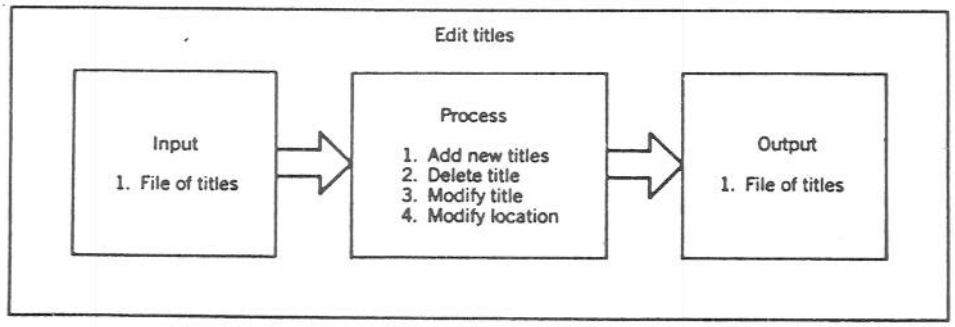
HIPO-Charts:



Hierarchie-Diagramm für ein System, welches eine Liste von Buch-Titeln verwaltet (HIPO-Entwurf).

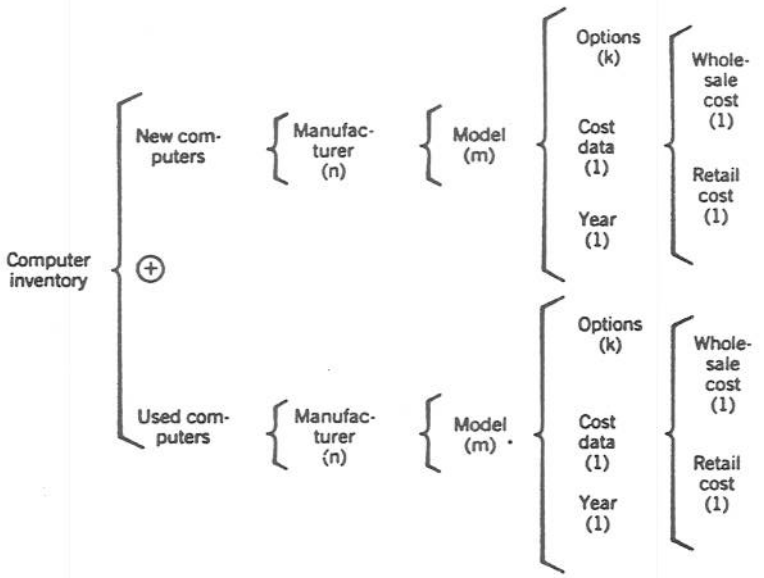


Eingabe-Prozeß-Ausgabe (IPO)-Diagramm zu dem obig genannten Verwaltungssystem (HIPO-Entwurf).

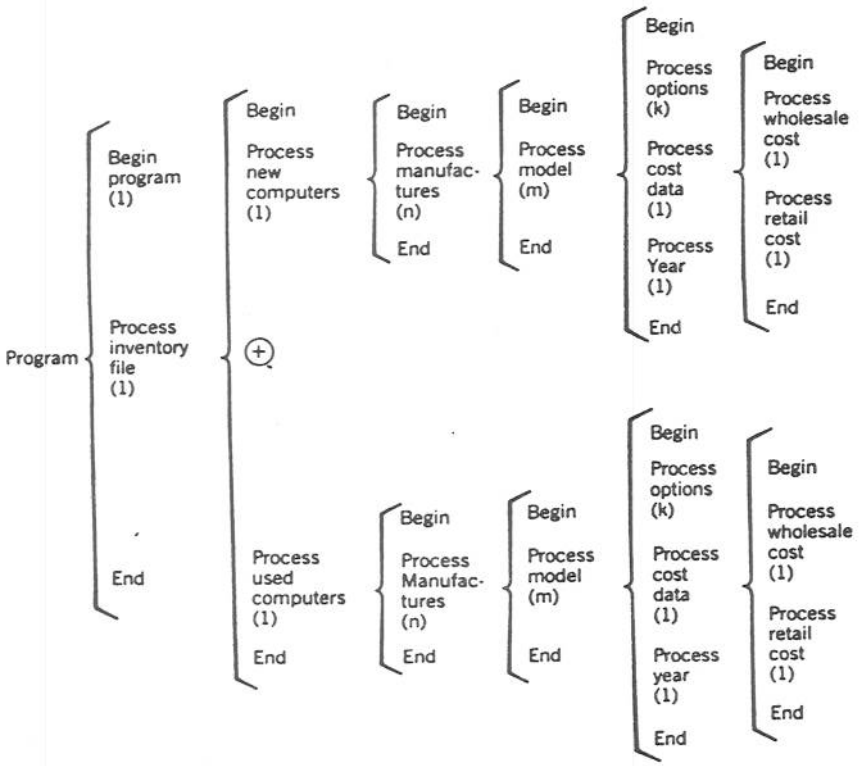


IPO-Diagramm für "Edit Titles" (siehe voriges Diagramm).

Warnier-Orr-Diagramme:

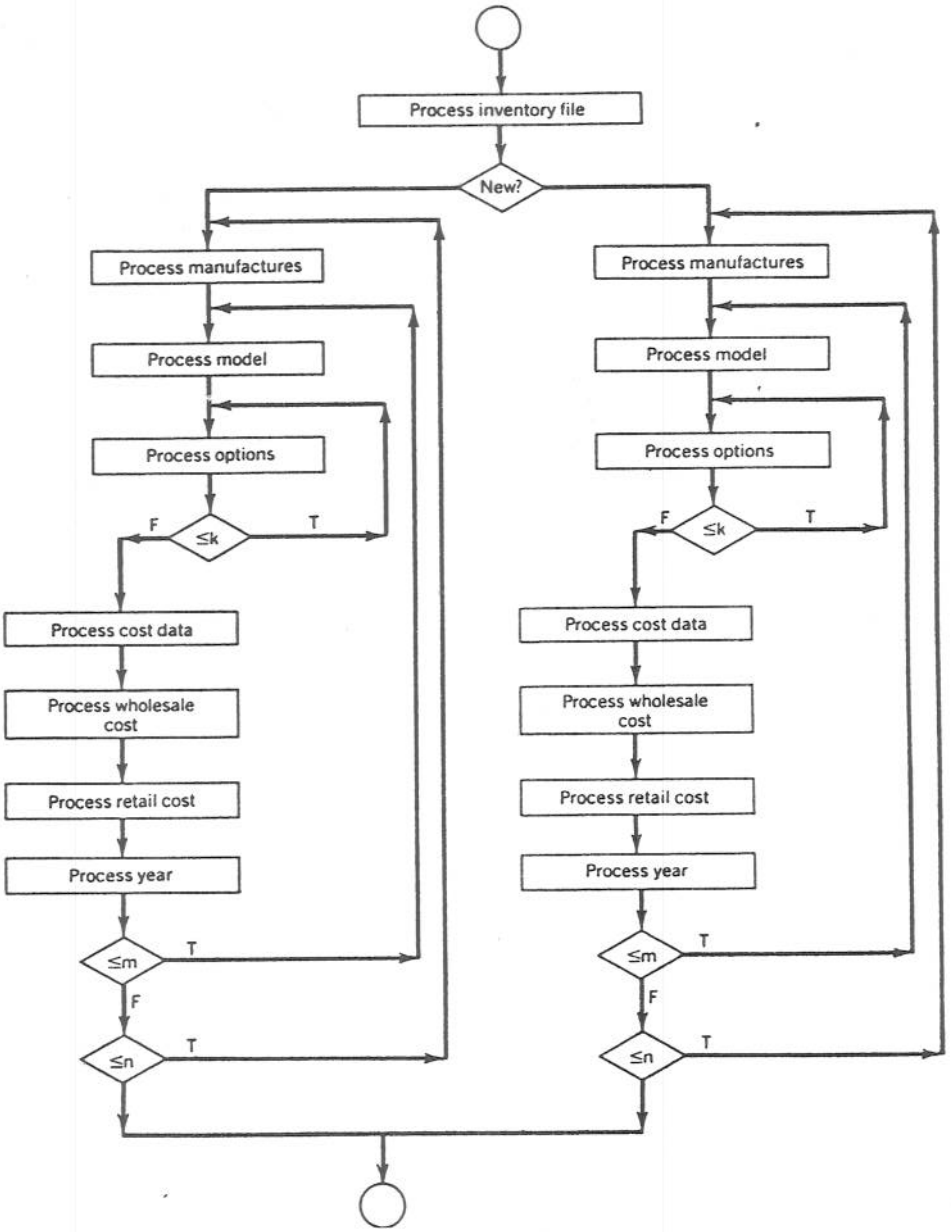


Warnier-Orr-Diagramm für ein System, welches ein Bestandsverzeichnis eines Computer-Versandhauses führt.



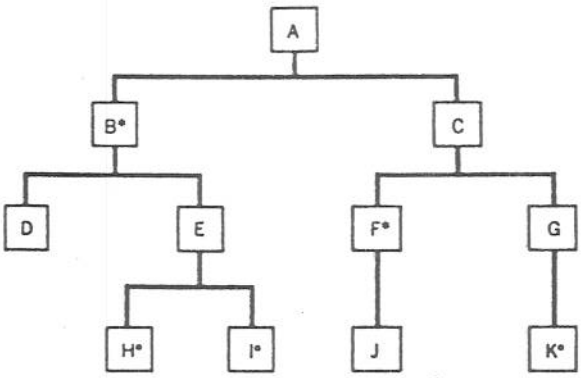
Warnier-Orr-Prozeß-Hierarchie-Diagramm für obiges Problem

Warnier-Orr-Diagramme lassen sich leicht in eine konventionelle Flußdiagramm-Darstellung umsetzen. Das Flußdiagramm auf der nächsten Seite entspricht obigem Warnier-Orr-Diagramm.

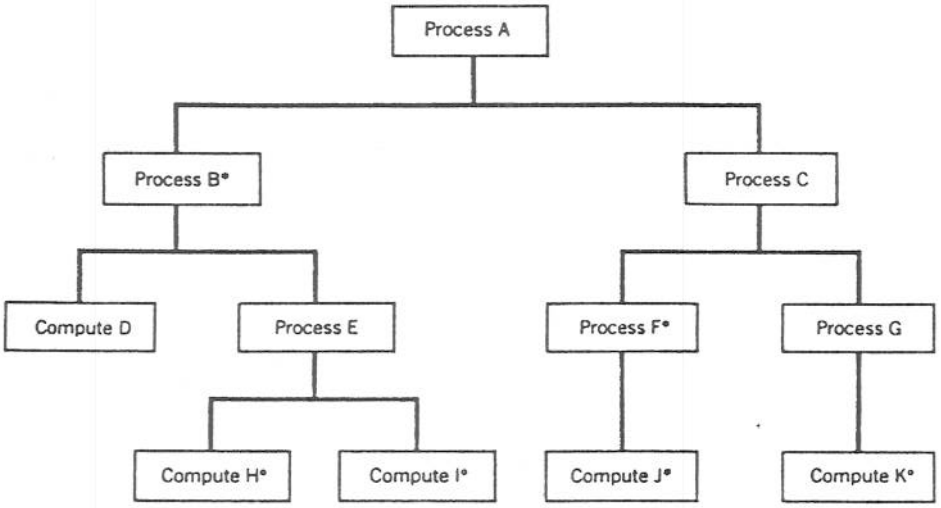


Flußdiagramm, das dem vorigen Warnier-Orr-Diagramm entspricht

Jackson-Entwurf-Notation:



Jackson-Datenstruktur-Notation

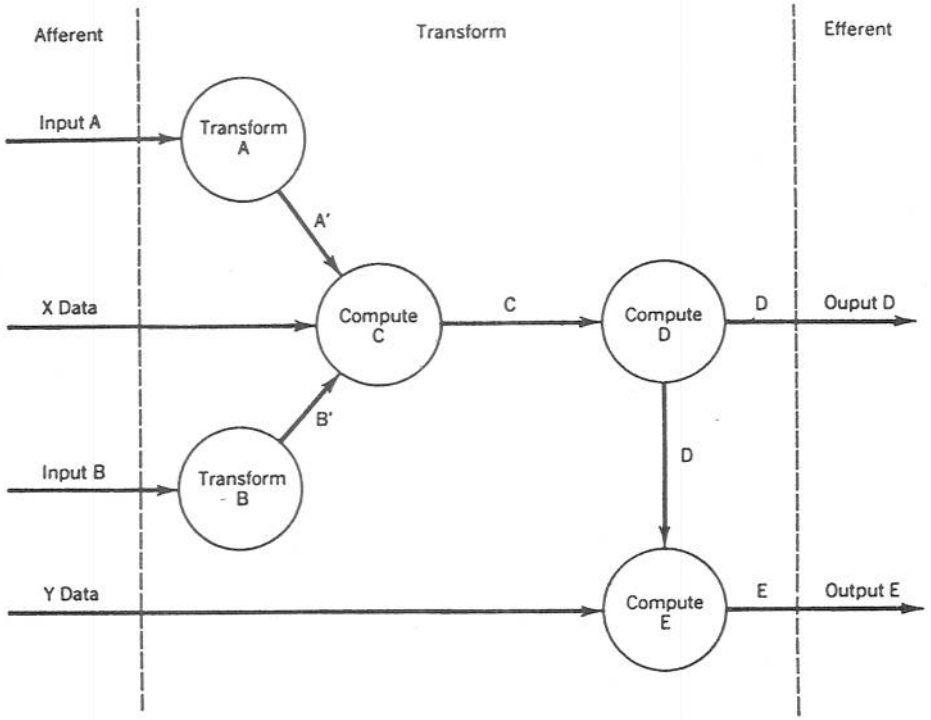


Jackson-Prozeß-Hierarchie (aufbauend auf vorigem Diagramm)

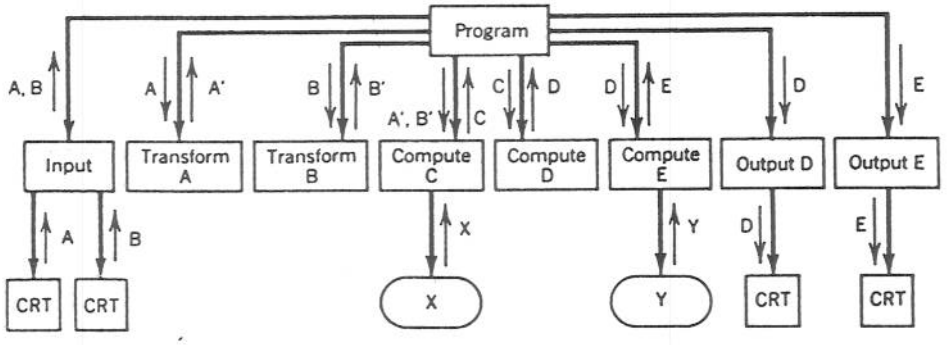
prozedurale Darstellung des Programms (aufbauend auf vorigem Diagramm)

```
A  sequence
   B  iteration
      do D;
      E  select
         do H;
      E  or
         do I;
      E  end
   B  end
   C  sequence
      F  iteration
         do J;
      F  end
      G  select
         do K;
      G  end
   C  end
A  end.
```

Datenflußdiagramme:



Datenflußdiagramm für die Transformationsanalyse mit einer möglichen Festlegung der Grenzen für Afferent- und Efferent-Datenflüsse.

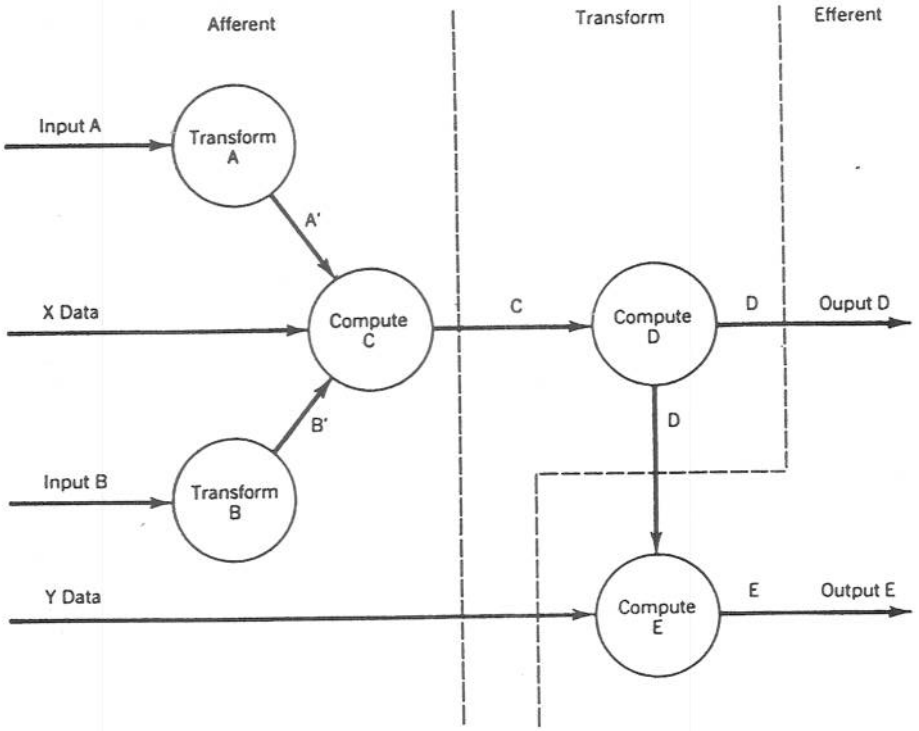


Zu obigem Datenflußdiagramm gehöriger Struktur-Chart.

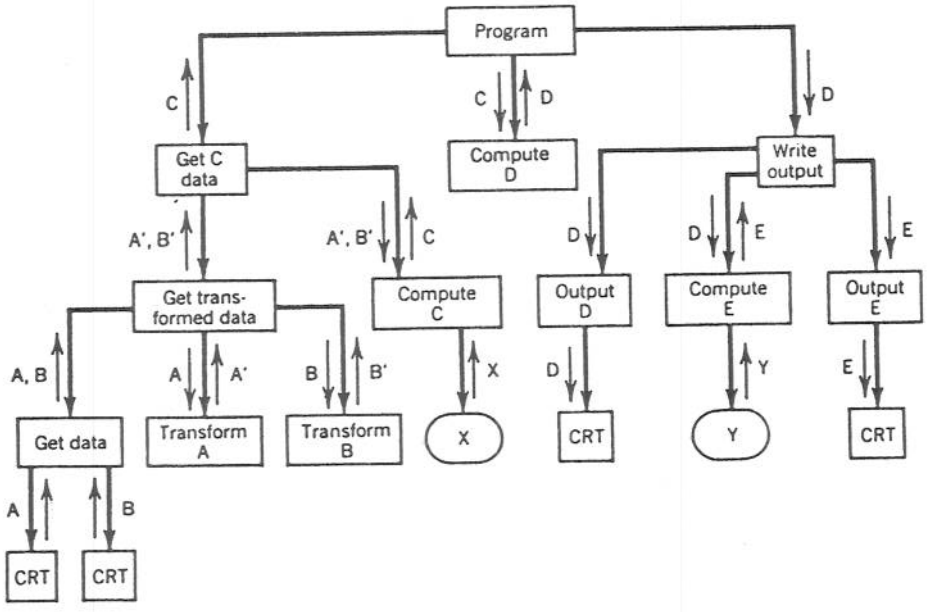
Die Spezifikation der Fluß-Grenzen (Afferent- und Efferent-) ist nicht eindeutig.

Das obige Datenflußdiagramm geht davon aus, daß die Einheiten A bis E Transformationszentren bilden.

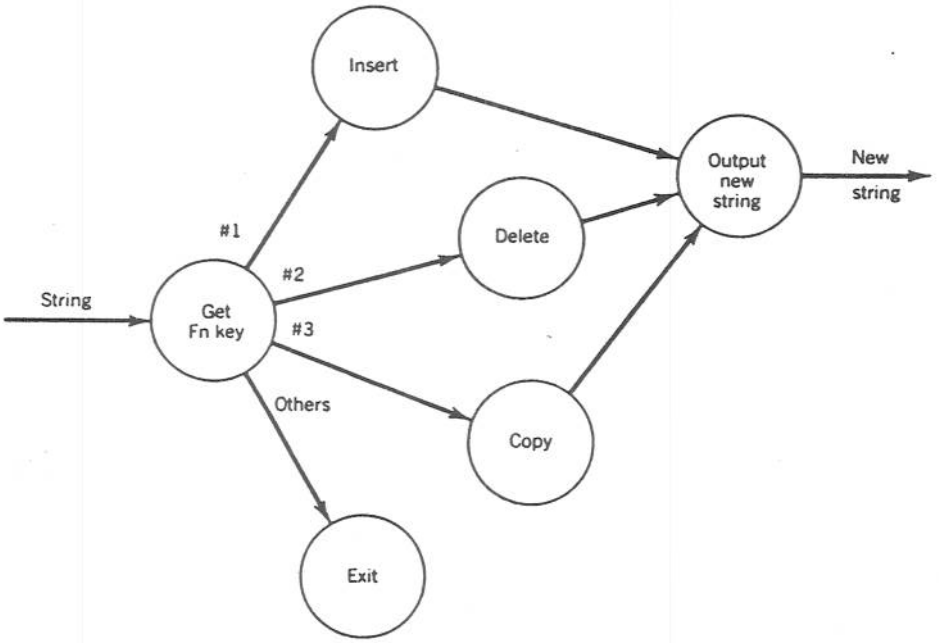
Das auf der nächsten Seite abgebildetete Datenflußdiagramm mit zugehörigem Struktur-Chart zum gleichen Beispiel illustriert eine andere vom Designer gewählte Aufteilung. Hierbei ist nur die Einheit D Transformationszentrum.



Datenflußdiagramm für die Transformationsanalyse mit einer anderen möglichen Festlegung der Grenzen für Afferent- und Efferent-Datenflüsse.

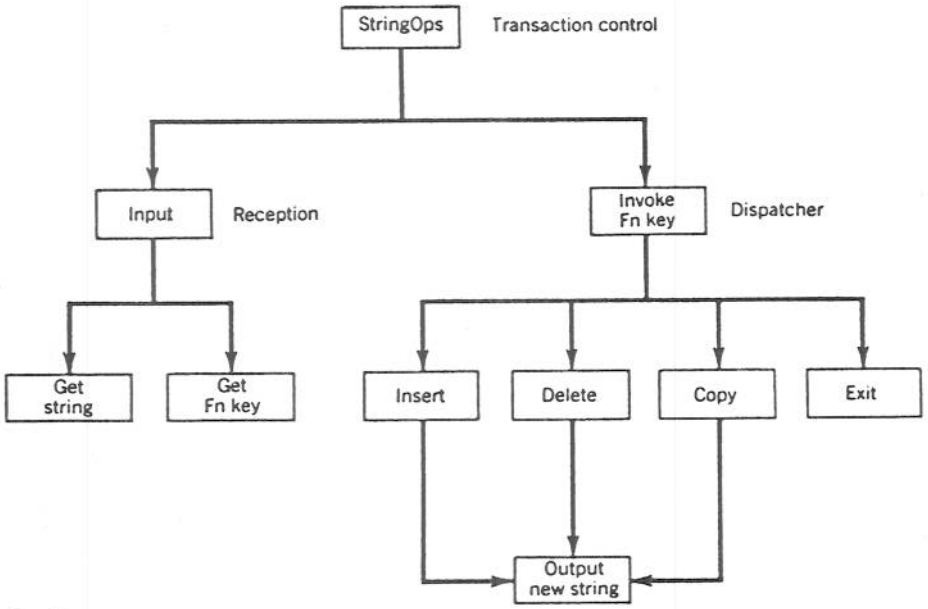


Zu obigem Datenflußdiagramm gehöriger Struktur-Chart.



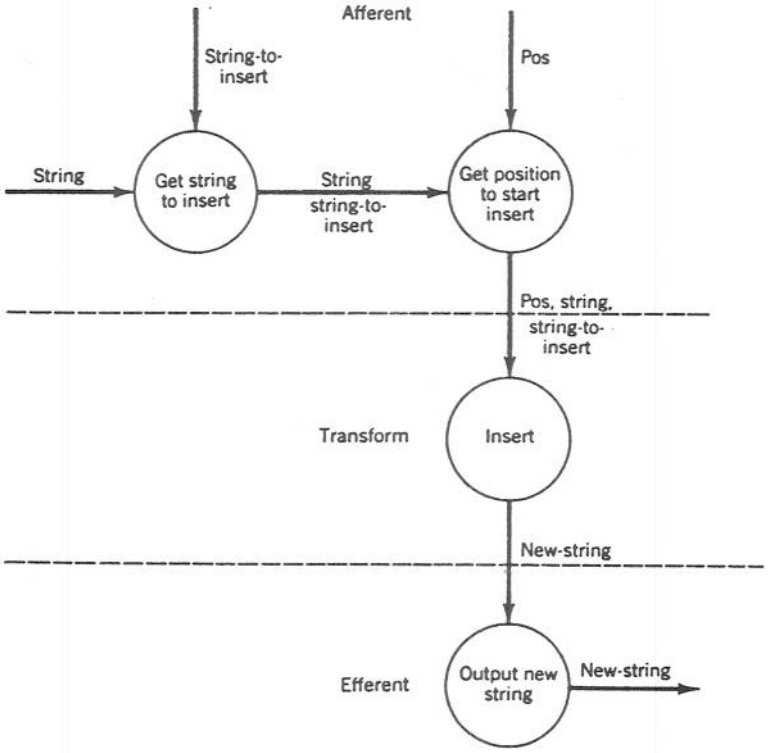
Datenflußdiagramm für das Problem Stringmanipulation mit Transaktionszentrum "Get Fn Key".

Der Transaktionsfluß kann in eine Software-Struktur überführt werden, welche einen Daten empfangenden (Reception) und einen verarbeitenden Ast (Dispatcher) aufweist. Unterstrukturen des verarbeitenden Astes kontrollieren alle auf der Transaktion basierenden Prozesse.

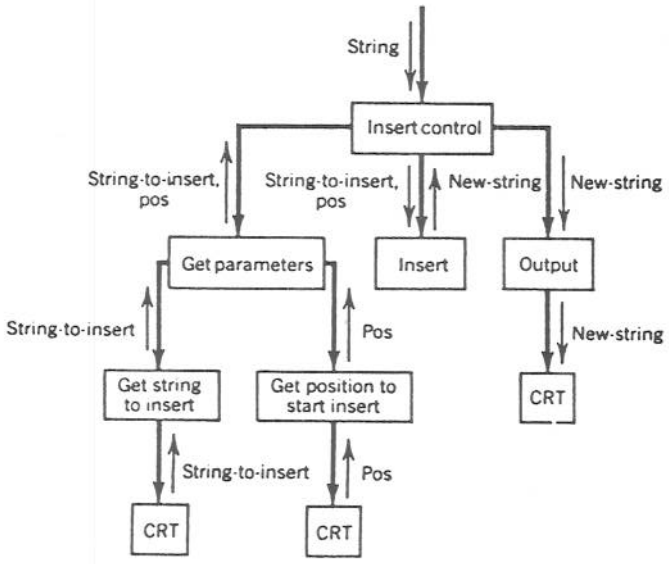


Faktorisierung auf der ersten Ebene für das gegebene Problem.

Faktorisierung auf der ersten Ebene liefert einen Eingabe-Ast, dessen Struktur mittels Transformationsanalyse ermittelt werden kann. Die Struktur eines jeden Aktionspfades kann abhängig vom Datenfluß-Typ durch Transformations- oder Transaktionsanalyse verfeinert werden.

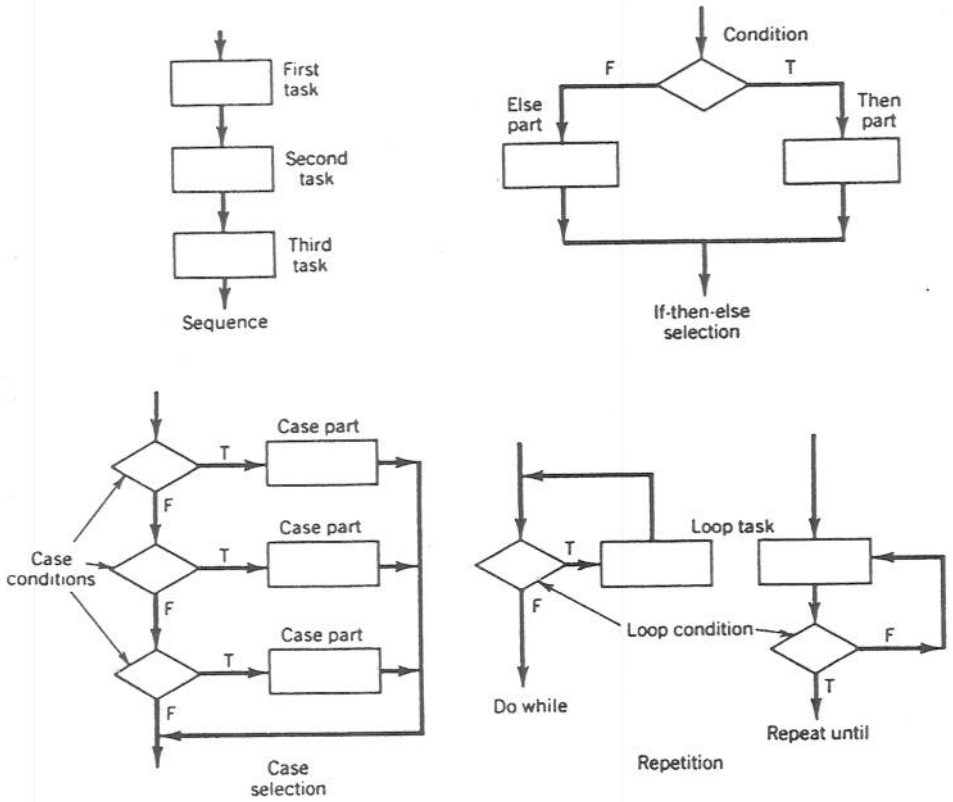


Verfeinertes Datenflußdiagramm für die Insert-Operation.



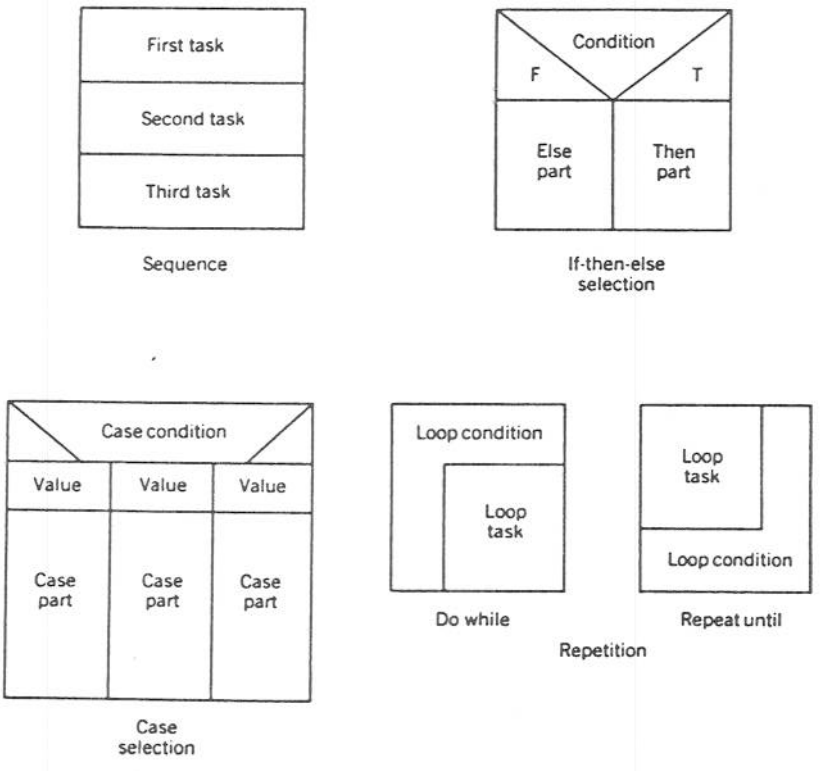
Durch Transformationsanalyse gewonnener Struktur-Chart für das Modul "Insert".

Flußdiagramme:



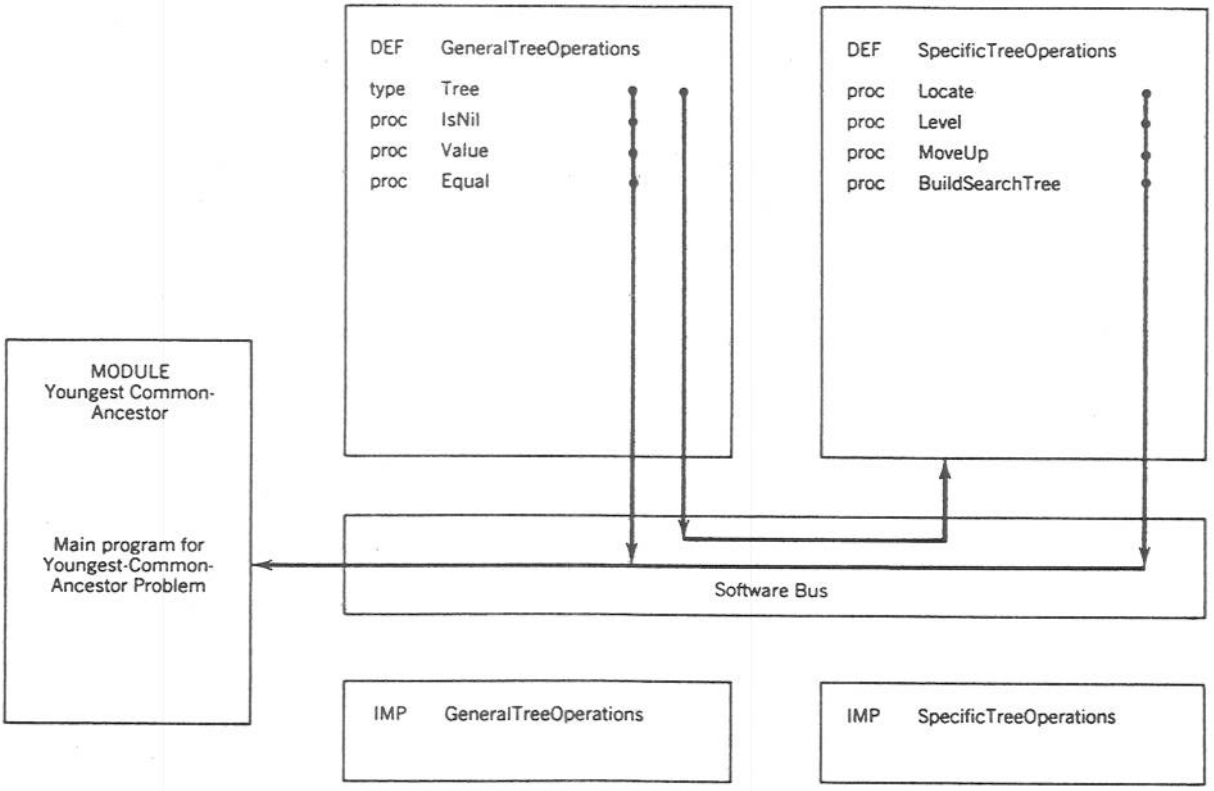
Flußdiagramm-Konstrukte.

Nassi-Shneiderman-Diagramme:

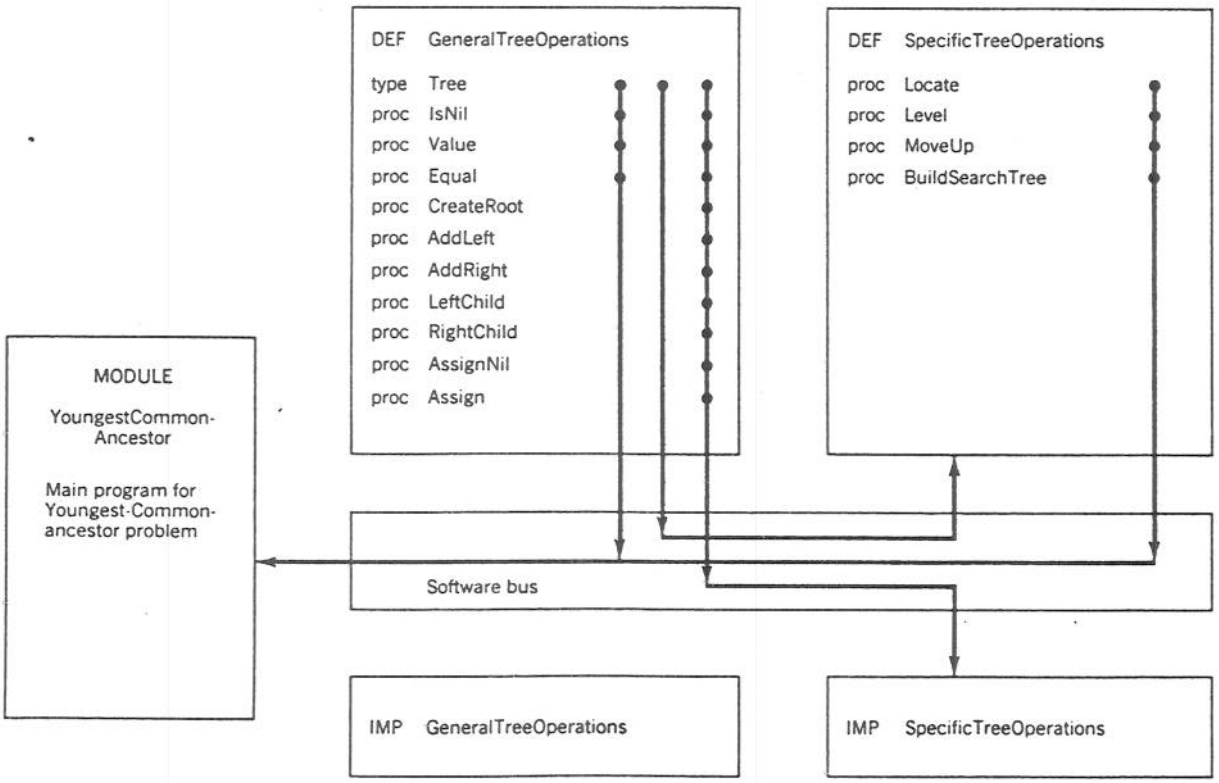


Nassi-Shneiderman-Symbole.

Modulare Entwurf-Diagramme:



Erstes modulares Entwurf-Diagramm zum Problem des jüngsten gemeinsamen Vorgängers.



Vollständiges modulares Entwurf-Diagramm für das Problem des jüngsten gemeinsamen Vorgängers.