Jedes Programm wird so lange erweitert,
bis aller verfügbarer Speicherplatz belegt ist.

Seine Komplexität wächst so lange,
bis das Leistungsvermögen des Pogrammierers überschritten ist.

6. Die Realisierungsphase

6.1 Phasenaufgabe und Phasenziele

Nachdem in der Planungsphase die einzelnen Module des zu erstellenden Softwaresystems in ihren auszuführenden Tätigkeiten durch ihre Schnittstelle spezifiziert wurden, ist das Ziel der Realisierungsphase die Erstellung von programmtechnischen Lösungen für die so spezifizierten Module. Aufgrund der Parallelarbeit ist der Übergang von der Planungs- zur Realisierungsphase individuell für jedes Modul; bei einer top-down-Entwicklung (vgl. Bild 5.1) verläuft die Realisierung benutzermaschinennaher Module oft gleichzeitig zur Planung der basismaschinennahen Module.

Weitergegeben aus der Realisierungsphase an die sich anschließende Testphase werden die in der gewünschten Programmiersprache realisierten und syntaxfehlerfreien Module. Dabei können Module, deren Inhalt von einem Übersetzer (Compiler) als formal den Regeln der Realisierungs-Programmiersprache genügend anerkannt wurde (syntaktische Fehler), können durchaus noch logische, d. h. gedankliche Fehler (semantische Fehler) enthalten. Weiterhin sollen während der Realisierungsphase Überlegungen zum Modultest durchgeführt werden. Die Ergebnisse dieser Überlegungen gelangen in Form von Testhinweisen und Vorschlägen für Testdaten (Testfälle) an die Testphase zur Testkartenerstellung.

Von grundlegender Bedeutung ist im Zusammenhang der Realisierungsphase der Begriff des Algorithmus. Ein Algorithmus ist eine rechnerunabhängige, genaue Vorschrift, welche kleinsten Einzelschritte eine Aufgabe zu ihrer Lösung durchläuft. Die intuitive oder formale Entwicklung eines Algorithmus ist also die Zerteilung oder Aufteilung einer Aufgabe in kleinste sequentiell zu durchlaufende Schritte (z. B. Kontrollstrukturen einer Programmiersprache).

Eine Teilaufgabe, die durch das Innere eines Moduls repräsentiert wird, ist (Voraussetzung der Planungsphase) für den menschlichen Geist ohne Schwierigkeiten durchschaubar und erfassbar. Sie wird jedoch in der Realisierungsphase weiter in noch feinere Einzelschritte aufgespalten und formal mit Hilfe einer Programmiersprache beschrieben, um auch vom Digitalrechner erfassbar zu sein.

Der Algorithmus als Methode und Beschreibungsmittel für sequentiell ablaufende Aufgaben eignet sich in diesem Zusammenhang besonders gut, denn der Digitalrechner ist ja ein sehr schneller sequentieller Automat. Für zeitlich parallel ablaufende und voneinander abhängige Aufgaben werden andere Beschreibungsmittel

(z. B. Petri-Netze) eingesetzt, die in diesem Zusammenhang nicht besprochen werden sollen.

Die Vorgehensweise innerhalb der Realisierungsphase lässt sich in folgende vier Einzelschritte aufteilen, die in dieser Reihenfolge durchlaufen werden sollen:

1. Konzept für Daten und Algorithmen

Die ersten Gedanken zur Aufspaltung der Teilaufgabe gelten den internen Datenstrukturen und den Kontrollstrukturen des Ablaufs. Je nach Schwerpunkt in den einzelnen Modulen - stark datenorientiert mit wenigen Kontrollstrukturen oder wenige Daten mit vielen komplexen Entscheidungen - werden hier unterschiedliche Verfahren eingesetzt.

2. Dokumentation der Lösung

Die jetzt festgelegten Einzelschritte sollen anschließend dokumentiert werden. Das benutzte Dokumentationsverfahren soll programmiersprachenunabhängig sein. In der Praxis fließen die Punkte 1. und 2. zusammen, denn die meisten der in 1. verwendbaren Verfahren erzeugen durch ihre Anwendung automatisch Dokumentation, vor allem dann, wenn sie als rechnergestütztes Werkzeug existieren. Punkt 2 wird dann degradiert zur Aufbereitung der existierenden Dokumentation.

3. Umsetzung in eine Programmiersprache

Der bis hierher rechnerunabhängig formulierte Algorithmus wird nun in Statements der Zielprogrammiersprache umgesetzt, was meist nach grundlegender Vorarbeit in Punkt 1. und 2. ein formaler Vorgang darstellt. In neuerer Zeit werden mehr und mehr automatische Programmgeneratoren eingesetzt, die geeignet formulierte Algorithmen automatisch, d. h. durch den Rechner selbst, in die gewünschte Programmiersprache umsetzen. Das ist vor allem dann vorteilhaft möglich, wenn schon der Algorithmus bereits mit Hilfe eines rechnergestützten Werkzeugs erstellt wurde. Die Programmiersprachenumsetzung ist dann eine Erweiterung dieses Werkzeugs.

4. Codierung mit dem Rechner

Das jetzt in der gewünschten Programmiersprache existierende Modul wird in den Rechner eingegeben. Dieser Teil aus Punkt 4 verschmiert oft mit Punkt 3, denn ein geeignet dokumentarisch vorbereiteter Algorithmus ist vom Fachmann/frau ohne Umweg über eine handschriftliche Form des Programms direkt in den Rechner eingebbar. Bei der automatischen Umformung entfällt dieser Teil.

Anschließend wird der Übersetzer für die gewählte Programmiersprache gestartet, der das Programm auf formale Fehlerfreiheit überprüft und den rechnerinternen Maschinencode erzeugt. Meldet der Übersetzer formale Fehler, so wird die Programmquelle so lange korrigiert und erneut übersetzt, bis das programmtechnisch realisierte Modul keine formalen, d. h. syntaktischen Fehler mehr enthält.

Man beachte, das bis einschließlich Punkt 2. die Entwicklung programmiersprachenunabhängig erfolgt. Diese Vorgehensweise, möglichst lang sprachunabhängige Strukturen zu verwenden, geschieht vor allem vor dem Hintergrund der Arbeitsreduzierung im Fall der späteren Wiederverwendung des Moduls in einer anderen Umgebung und/oder einer anderen Sprache.

Die Schritte der Realisierungsphase sind wohl die bekanntesten Schritte der Programmerstellung. Sie wurden schon verwendet, als der Begriff des Software Engineering noch unbekannt war. Aus diesem Grund existieren zur Realisierungsphase die ältesten und wohl auch die fundiertesten Verfahren und formalen Vorgehensweisen. Einige dieser Verfahren sollen im Folgenden vorgestellt werden. Die vorgestellten Verfahren sind ausnahmslos kontrollflussorientiert und werden bevorzugt im Bereich der wissenschaftlichen und der Prozessdatenverarbeitung eingesetzt. Datenorientierte Verfahren, deren Einsatzgebiet im Bereich der kommerziellen Datenverarbeitung liegt, werden hier nicht beschrieben.

6.2 Strukturierte Programmierung

Es gibt wohl kaum einen Begriff aus dem Bereich des Software Engineering, der in den letzten Jahren mehr strapaziert worden wäre, als derjenige der strukturierten Programmierung. Das Prinzip, das diesem Verfahren zugrunde liegt, basiert auf der Idee der Verringerung von Komplexität und der Verbesserung der Überschaubarkeit durch Vereinbarung geeigneter Einschränkungen, oft **lineare Kontrollstrukturen** genannt.

Man unterscheidet bei einem Algorithmus zwischen einfachen Anweisungen, die den Ablauf nicht beeinflussen, und Kontrollflussanweisungen. Kontrollflussanweisungen führen keine Veränderungen an den zu bearbeitenden Daten durch, sondern sie steuern den Programmablauf und damit den Ablauf der einfachen Anweisungen. Sie geben also an, ob und gegebenenfalls wie oft die einfachen Anweisungen auszuführen sind.

Das Grundidee der Strukturierten Programmierung ist die ausschließliche Verwendung einiger weniger Konstruktionsprinzipien für Kontrollflussanweisungen, die eine allgemein anerkannte und verständliche Darstellungsform des Sachverhalts darstellt. Diese Darstellungsform kann gleichzeitig zur Dokumentation der Algorithmen dienen. Im Folgenden werden die linearen Kontrollstrukturen parallel mittels dreier verschiedener Darstellungsmethoden beschrieben. Diese Darstellungsmethoden sind:

- **Programmablaufpläne**

  Der Ursprung der Programmablaufpläne ist nicht rückverfolgbar. Sie sind seit dem Jahr 1969 in DIN 66001 genormt und stellen auch heute noch das am häufigsten verwendete Verfahren dar. Die Darstellung verwendet graphische Symbole wie Rechtecke (Operation) und Rauten (Verzweigung), die durch gerichtete Graphen miteinander verbunden sind und so den Ablauf der Kontrollstrukturen charakterisieren.

  Programmablaufpläne erlauben die Darstellung beliebiger also auch nichtlinearer Kontrollstrukturen. Im Gegensatz zu den beiden anderen Verfahren hat der Entwickler dafür Sorge zu tragen, sich geeignet auf lineare Kontrollstrukturen einzuschränken; ein Punkt, der es ratsam erscheinen lässt, vom Standpunkt der Strukturierten Programmierung Programmablaufpläne abzulehnen.

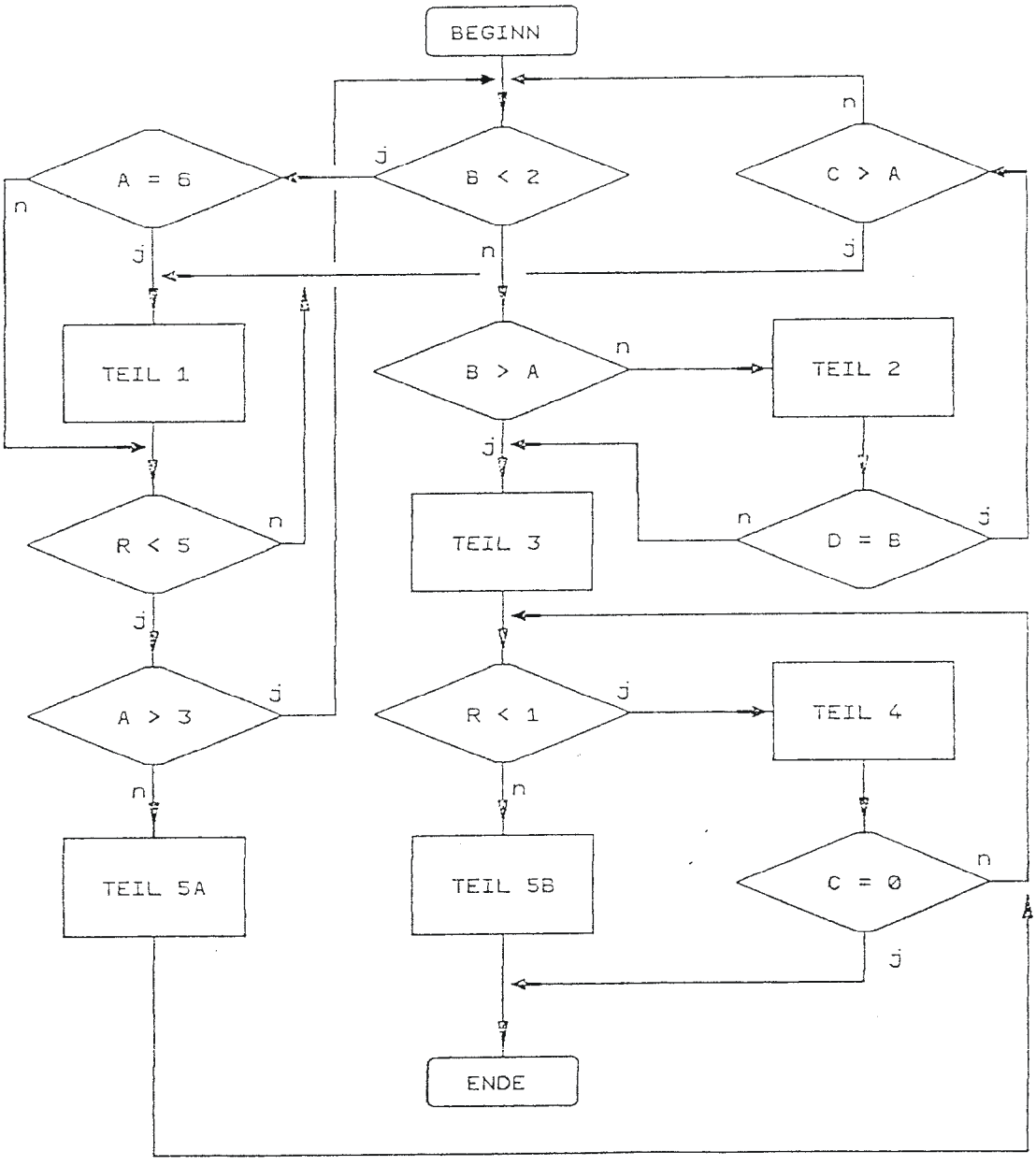

Bild 6.2-1: Beispiel eines unübersichtlichen Programmablaufplans (PAP)

- **Struktogramme**
  Struktogramme wurden 1973 von Nassi und Shneiderman /8/ vorgestellt und heißen aus diesem Grund oft auch N/S-Diagramme. Auch die Struktogrammtechnik ist eine graphische Methode, jedoch ist durch die äußere Form der graphischen Symbole gewährleistet, dass automatisch nur lineare Kontrollstrukturen dargestellt werden können. Übrigens wurde zur gleichen Zeit die Programmiersprache PASCAL und Modula vorgestellt, die in ihrem Sprachvorrat die gleichen linearen Kontrollstrukturen verwendet.

- **Pseudocode**
  Die Idee, Algorithmen mittels Pseudocode zu entwickeln und zu dokumentieren, entstand ca. 1975. Es handelt sich beim Pseudocode um eine textuelle programmiersprachenähnliche Darstellungsform, die sich zusammensetzt aus definierten formalen Worten für die Kontrollstrukturen und normalem umgangssprachlichen Text für die einfachen Anweisungen.
  Die Dokumentation im Pseudocode besitzt den Vorteil der guten maschinellen Speichermöglichkeit und einer ausgezeichneten Änderungsfreundlichkeit. Auf Rechnern installierte automatische Werkzeuge erlauben den Test auf formale Richtigkeit der linearen Kontrollstrukturen und die automatische Umsetzung in die gewählte Programmiersprache durch so genannte Programmgeneratoren.

Böhm und Jacopini haben (unter einer bestimmten Voraussetzung, (s. Kap. 6.4) bewiesen [8], dass alle vorstellbaren sequentiell ablaufenden Algorithmen ausschließlich unter Verwendung, der folgenden Bausteine (d-Strukturen) lösbar sind:

1. Einfache Anweisungen, die nicht den Kontrollfluss beeinflussen

2. Sequenz zweier d-Strukturen der Form
   "d-Struktur 1" ;
   "d-Struktur 2"

3. Bedingt ausgeführte d-Struktur der Form
   IF "boolscher Ausdruck"
   THEN "d-Struktur"

4. Bedingt wiederholte d-Struktur der Form
   WHILE "boolscher Ausdruck"
   DO "d-Struktur"

Man beachte die rekursive Formulierung, die es erlaubt, die Bausteine in beliebige Tiefe zu schachteln. Bemerkenswert ist ebenso das völlige Fehlen von unbedingten Sprungbausteinen und die Umschreibung von bedingten Sprungbausteinen durch die Punkte 3. (der bedingte Vorwärtssprung) und 4. (der bedingte Rückwärtssprung).

Nützlichkeitsüberlegungen der Praxis lassen es sinnvoll erscheinen, die Punkte 1. bis 4. geeignet zu erweitern, ohne dass dabei der Anspruch auf vollständige Realisierbarkeit aller denkbarer sequentieller Algorithmen und derjenige der Einhaltung der Regeln für lineare Kontrollstrukturen verloren geht. So stehen in der Praxis folgende Bausteine (erweiterte d-Strukturen) zur Verfügung:

1. Einfache Anweisungen, die nicht den Kontrollfluss beeinflussen

2. Sequenz zweier d-Strukturen der Form
"d-Struktur 1" ; "d-Struktur 2"

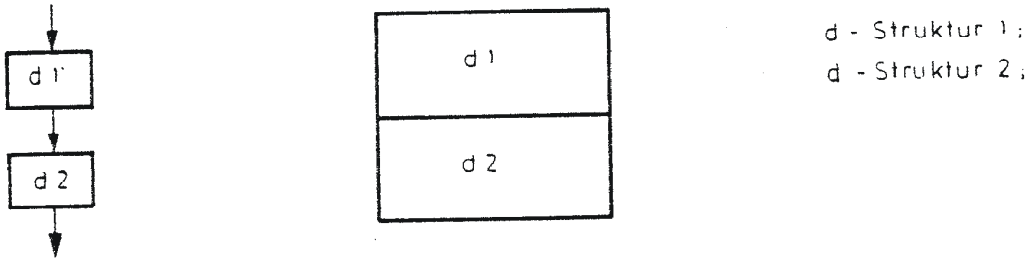

Bild 6.1: Sequenz zweier d-Strukturen

3. Bedingt ausgeführte d-Struktur der Form:
IF "boolscher Ausdruck" THEN "d-Struktur 1
ELSE "d-Struktur 2"

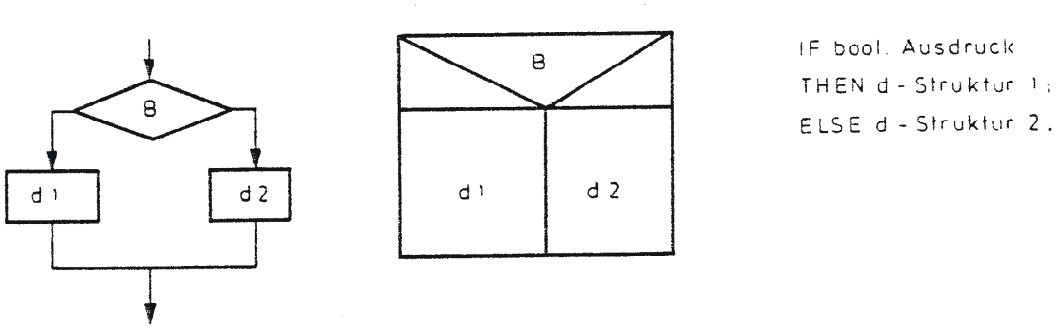

Bild 6.2: Bedingt ausgeführte d-Struktur vom Typ IF

4. Bedingt ausgeführte d-Struktur der Form :
CASE "ausdruck" OF
"konst1": "d-Struktur 1"
"konst2": "d-Struktur 2"
.....
OTHERWISE: "d-Struktur n"

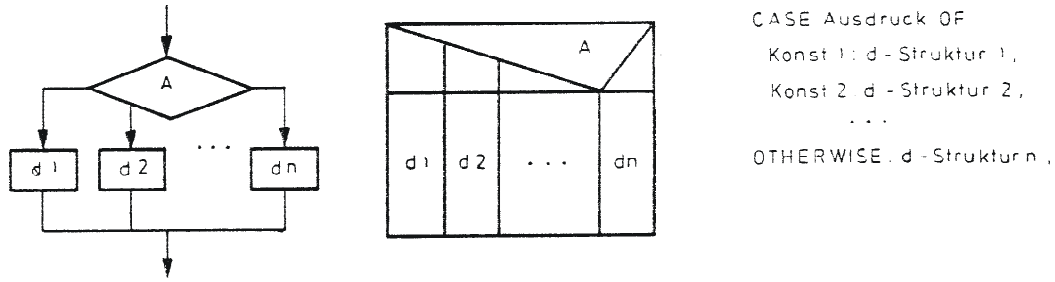

Bild 6.3: Bedingt ausgeführte d-Stuktur vom Typ CASE

5. Bedingt wiederholte d-Struktur der Form
   WHILE "boolscher Ausdruck" DO "d-Struktur"

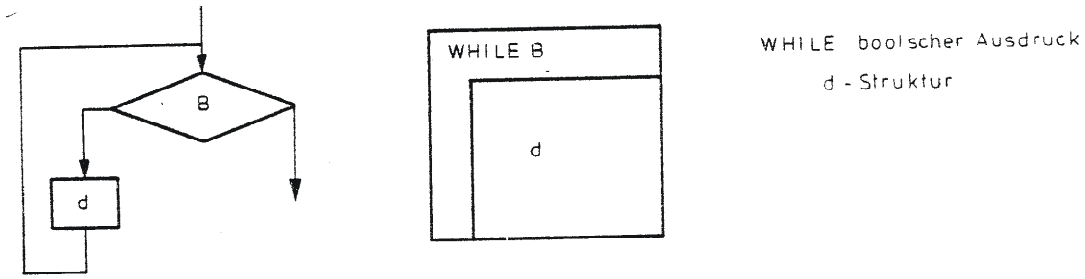

Bild 6.4: Bedingt wiederholte d-Struktur der Form WHILE

6. Bedingt wiederholte d-Struktur der Form
   DO "d-Struktur" UNTIL "boolscher Ausdruck"

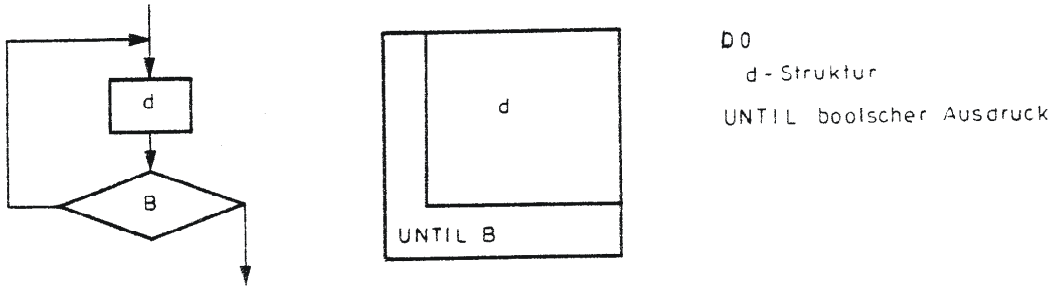

Bild 6.5: Bedingt wiederholte d-Struktur der Form UNTIL

7. Aufruf einer d-Struktur, die an anderer Stelle (in einem anderen Modul) existiert, der Form
   CALL "d-struktur-name"

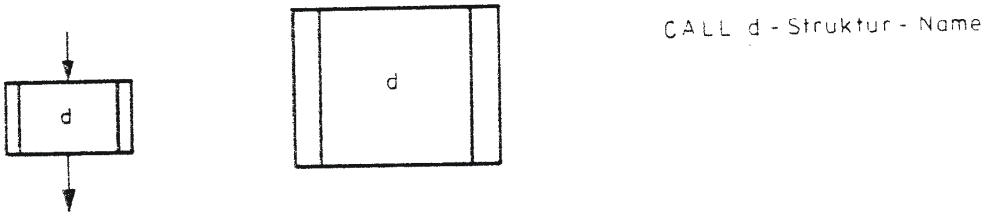

Bild 6.6: Aufruf einer d-Struktur

Das gemeinsame Kennzeichen der so definierten d-Strukturen ist die Existenz genau eines definierten Eingangs und genau eines definierten Ausgangs. Wird also eine d-Struktur durch den Rechner abgearbeitet, so gibt es genau eine definierte Stelle, an der sie betreten und eine ebenso definierte Stelle, an der sie verlassen wird. Innerhalb einer d-Struktur verläuft der Kontrollfluss streng Richtung Ausgang der d-Struktur. Der Verlauf des Kontrollflusses in umgekehrter Richtung gelingt nur mit Hilfe von bedingten Wiederholungen.

Die durch die rekursive Formulierung der d-Strukturen mögliche Schachtelung erfolgt automatisch derart, dass eine d-Struktur immer vollständig in einer anderen liegt oder diese vollständig umfasst. Es gibt keine partiellen Überschneidungen. Mathematisch ausgedrückt: Die möglichen Wege in einer d-Struktur sind immer durch einen planaren Graphen darstellbar. Es ist leicht vorstellbar, das derartige Strukturen besonders leicht überschaubar und testbar sind.

6.3 Der unbedingte Sprung

Es mag schon aufgefallen sein, dass d-Strukturen prinzipiell keine unbedingten Sprünge enthalten. Man bezeichnet deshalb die strukturierte Programmierung auch als die GOTO-lose Programmierung. Über das Für und Wider von GOTOs ist viel geschrieben und gestritten worden; hier sollen einige Fakten aufgezeigt werden, die die Befürworter der GOTO-losen Programmierung anführen.

- Die Semantik einer Sprunganweisung ist nicht selbsterklärend. Das bedeutet, dass die Verwendung einer Sprunganweisung die Information verwischt, ob diese Sprunganweisung Teil eines bedingt auszuführenden oder Teil eines wiederholt auszuführenden Programmteils ist. Bedingt auszuführende und wiederholt auszuführende Programmteile sollten aber wegen ihres semantischen Unterschieds als solche klar erkennbar sein.
- Die Linearität des Programmablaufs ist nicht gewährleistet. Mit einem Sprung ist es möglich, in beliebige Stellen eines Algorithmus oder einer Ablaufstruktur hineinzuspringen oder diese mit beliebigem Ziel zu verlassen. Das Denken in d-Strukturen wird nicht unterstützt.
- Da - vor allem nachträglich - mittels GOTO in jeder Ablaufstruktur ein "emergency entry" oder "emergency exit" eingebaut werden kann, führt die historische Entwicklung des Modulinneren zu unübersichtlichen Lösungen und umgeht den Zwang, den Algorithmus von Grund auf neu zu überdenken und zu strukturieren.

Das Verwendungsverbot des unbedingten Sprungs bezieht sich natürlich nur auf den Algorithmus. Es ist unumgänglichen älteren Programmiersprachen mangels anderer geeigneter Anweisungen das GOTO zu verwenden, um damit einen Teil einer d-Struktur nachzubilden, die anders in dieser Programmiersprache nicht realisierbar wäre.

6.4 Die (n + ½) - Schleife

Im Abschnitt 6.2 wurde eine Einschränkung erwähnt, die die Umwandlung beliebiger Algorithmen in d-Strukturen betraf. Diese Einschränkung soll nun näher betrachtet werden. Es gilt die Aussage: Es gibt (Teil-)Algorithmen, die sich nicht in d-Strukturen zerlegen lassen. Diese Aussage scheint zunächst den Wert der d-Strukturen und der Strukturierten Programmierung an sich in Frage zu stellen.

Das Bild 6.7 zeigt ein konkretes Beispiel einer derartigen Struktur. Es handelt sich um eine DO .. UNTIL Wiederholung, die durch eine Bedingung vorzeitig verlassen werden kann. Sie wird also "(n + ½) mal" ausgeführt, ist nicht nach den Regeln der d-Strukturen umwandelbar in lineare Kontrollstrukturen und nicht in Form eines Struktogramms darstellbar.

Einen Ausweg aus diesem Dilemma fanden Böhm und Jacopini. Sie postulierten und bewiesen:

> Zu jedem beliebigen Algorithmus A gibt es einen funktional äquivalenten Algorithmus A'(der für jede zulässige Eingabe die gleichen Ergebnisse wie A liefert), der lediglich aus d-Strukturen aufgebaut ist. Bei der Konversion von A nach A' müssen eventuell neue Hilfsvariablen eingeführt werden sowie einfache Anweisungen zur Manipulation dieser Hilfsvariablen.

Bild 6.8 zeigt die d-Strukturen-Umwandlung aus Bild 6.7 unter Einfügung der boolschen Hilfsvariablen "break".

Prinzipiell kann man die Zahl der Strukturen, bei deren Umwandlung in d-Strukturen Hilfsvariable eingeführt werden müssen, auf vier Typen beschränken:

- Sprung in eine / aus einer Wiederholung
- Sprung in eine / aus einer Bedingung.

Die formale Lösung der Einführung von Hilfsvariablen ist jedoch nicht unbedingt im Sinne der linearen Kontrollstrukturen, die ja gerade die Übersichtlichkeit des Ablaufs zum Ziel haben. Deshalb wird die Kontrollstruktur der Wiederholung oft so verallgemeinert, dass innerhalb des Wiederholungsteils ein "break" (im Sinne einer Anweisung, nicht einer Kontrollstruktur) erlaubt ist, das den sofortigen definierten Sprung exakt zum Wiederholungsende bewirkt. Manche Programmiersprachen wie C oder ADA unterstützen diese Lösung durch syntaktische Notationen wie break, ESCAPE, EXIT oder LEAVE.

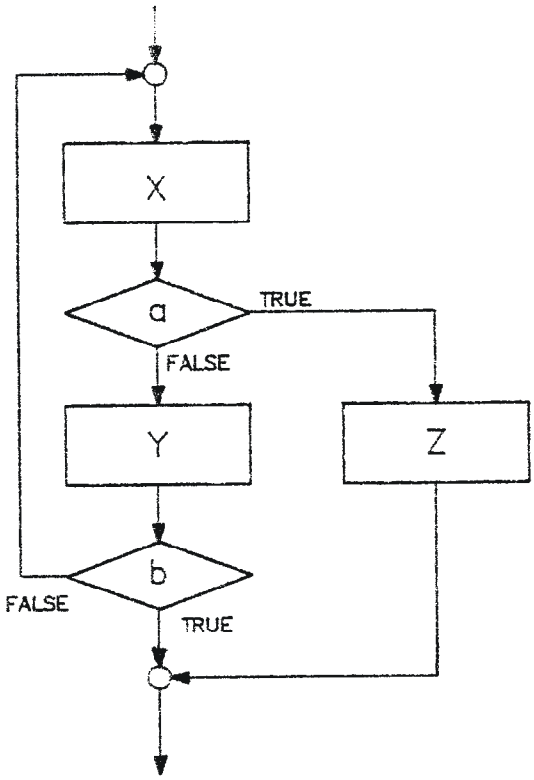

Bild 6.7: Nicht in d-Strukturen zerlegbarer Algorithmus

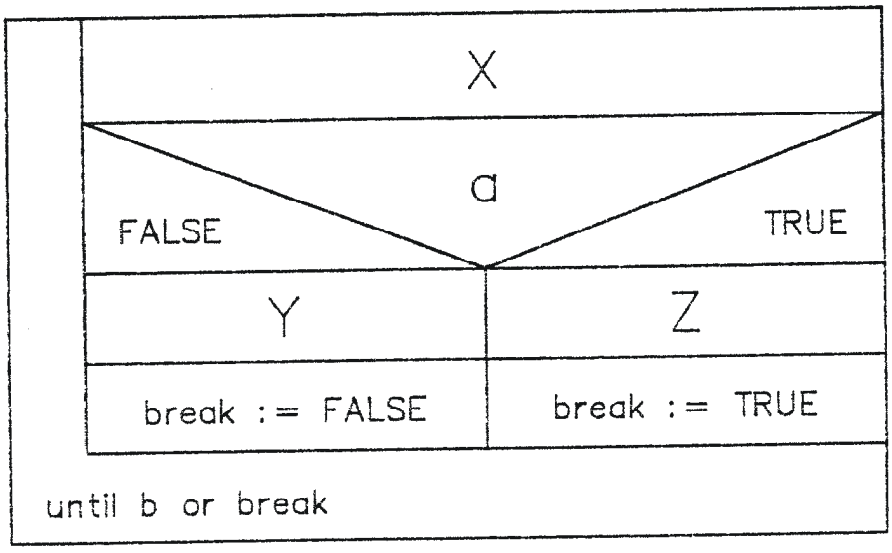

Bild 6.8: zu Bild 6.7 äquivalenter Algorithmus, basierend auf d-Strukturen

6.5 Aufbau von Programmquellen

Der folgende Abschnitt beinhaltet für den Praktiker einige Vorschläge für die Erstellung übersichtlicher und optisch schöner Programmquellen. Folgende formale Punkte sollten Berücksichtigung finden:

Jedes Modul beginne mit einem als Kommentar in der jeweiligen Programmiersprache realisierten Modulkopf. Dieser Modulkopf enthält Verwaltungsinformationen wie den Modulnamen, den Autor, das Erstellungsdatum und die Daten der letzten Änderung, sowie eine kurze verbale Beschreibung des Modulinhalts und eine Kurzinformation über die Exportschnittstelle. Die Daten der letzten Änderung sind sinnlos, wenn sie bei der nächsten Änderung nicht aktualisiert werden. Viele Programmiersprachen erlauben große und kleine Buchstaben und unterscheiden nicht zwischen ihnen (Ausnahme: die Sprache C). Verwendet man Grossbuchstaben für die Schlüsselworte der Sprache und Kleinbuchstaben für jegliche andere Information, so verbessert sich die Übersichtlichkeit erheblich.

Logisch voneinander getrennte Algorithmusteile sollten auch physikalisch durch Einfügung von Leerzeilen voneinander getrennt werden. Ist in der gewählten Programmiersprache keine leere Quellzeile erlaubt, so erfüllt eine leere Kommentarzeile den gleichen Zweck. Außerdem sollte reichlich von der Möglichkeit Gebrauch gemacht werden, das Programm durch Kommentare näher zu erläutern. Sinnvolle Kommentare unterscheiden sich jedoch in ihrem Inhalt vom zu kommentierenden Programm; ein Kommentar der Art

```
motor_aus = TRUE;        // schalte Motor aus
```

trägt zum Ärgernis jedes späteren Lesers bei.

Der optische Eindruck eines Programms ist wesentlich verbesserbar, wenn die Schachtelung der Kontrollstrukturen durch Einrücken hervorgehoben wird. An jeder Stelle des Programms ist so die Übersicht gewährleistet, wo eine d-Struktur beginnt und endet. Dieser Hinweis gilt für alle Programmiersprachen, auch für Assemblersprachen.

Moderne Programmiersprachen kennen die Deklarationspflicht. Jede im Programm verwendete Variable muss vor dem ersten ausführbaren Statement mit ihrem Datentyp und ihrem Namen genannt werden, z. B.

```
int  iZahl;
float  fZaehlerwert1;
```

Mit Hilfe dieser Redundanz werden schwer erkennbare Fehler vermieden, die im Programm durch Tippfehler von Variablennamen entstehen können. Statt der stillschweigenden Generierung einer neuen Variablen wird der unbekannte Name zurückgewiesen. Jede Typ- und Variablendeklaration gehört in eine eigene Zeile, jede deklarierte Variable erhalte einen kurzen Kommentar, der ihren Verwendungszweck beschreibt. Der damit verbundene erhöhte Schreibaufwand zahlt sich aus, wenn das Programm nach einer Ruhepause oder in der Wartung überarbeitet wird.

```
/*****************************************************************
*  ganzZahlFunktionen.c    Nützliche Funktionen für ganze Zahlen
*  Erstellungsdatum:         10.10.2003
*  Maier, Huber, Copyright (C) Firma Aaxcon GmbH
*  ---------------------------------------------------------------
*
*  Aufgabe: Berechnung von GGT, KGV, Quersumme, Primfaktorzerlegung
*           Dies Modul soll als Funktionsbibliothek eingesetzt werden
*
*  Bemerkungen und Hinweise:
*           Wurde zu Demonstrationszwecken erstellt
*
*  Bekannte Fehler und Einschränkungen:
*           Beim GGT muss die erste Zahl die größere sein
*---------------------------------------------------------------
*
* Änderungen
* 10.10.2002 GGT, KGV                                    Maier
* 11.06.2003 Qeuersumme, Primfaktorzerlegung             Müller
*****************************************************************/
#include <stdio.h>

/*====== Konstantenvereinbarungen ================================*/
#define MAXZAHL 32768 // Größte Zahl, die bearbeitet werden kann


/*====== Typvereinbarungen =======================================*/
typedef long TGanzeZahl


/*====== Globale Variablen =======================================*/


/*====== Funktionsdeklarationen ==================================*/


/*====== Funktionsdefinitionen ===================================*/


/*----------------------------------------------------------------
* ggT: Berechnet den größten gemeinsamen Teiler von zwei Zahlen.
*      Diese werden in den Eingabeparametern gzZahl1 und gzZahl2
*      übergeben.
* RETURN: Der GGT (Typ TGanzeZahl) wird zurückgegeben
----------------------------------------------------------------*/
TGanzeZahl ggT(TGanzeZahl gzZahl1, TGanzeZahl gzZahl2)
{
  TGanzeZahl gzRest;  // speichert den Rest der Division der beiden Zahlen

  while (1)
    {
      gzRest = gzZahl1 % gzZahl2;
                                // Rest = 0 heißt ggT gefunden -> gzZahl2

     if (gzRest == 0)
             return gzZahl2;   // ggT noch nicht gefunden

     gzZahl1 = gzZahl2;
     gzZahl2 = gzRest;         // gzZahl2 wird immer kleiner, schließlich 1
                               // spätestens dann ist der ggT gefunden !!!
   }
}
```

```
/*-------------------------------------------------------------------
* kgV: Berechnet das größte gemeinsame Vielfache von zwei Zahlen.
*      Diese werden in den Eingabeparametern gzZahl1 und gzZahl2
*      übergeben.
*
* Eingabeparameter
*         gzZahl1   ........
*         gzZahl2   ........
*
* Rückgabeparameter
*         RETURN: Das kgV (Typ TGanzeZahl) wird zurückgegeben
-------------------------------------------------------------------*/
TGanzeZahl kgV(TGanzeZahl gzZahl1, TGanzeZahl gzZahl2)
{
    // Die Implementation ...
    // ....

}
```

Jemand, den man um Hilfe bittet,
wird den Fehler nicht finden.

Jeder, der zufällig einen Blick auf das Listing wirft,
aber nicht um Rat gefragt wurde,
sieht den Fehler sofort.

Gesetz der unterlassenen Hilfeleistung

7. Softwaretest

7.1 Testgründe und Testziele

Nicht nur Softwaretest, sondern Test allgemein ist eine Folge menschlicher Unzulänglichkeit. Nur die Sicherheit, eine Entwicklung von Beginn an richtig durchgeführt zu haben, wäre die Entlastung von allen Dingen, die mit Tests zu tun haben.

Die Praxis sieht (vgl. Kap. 1) anders aus. Bis zu 30 % des Gesamtzeitaufwands für ein Softwareprojekt wird in den Test investiert. Trotzdem sind die abgelieferten Systeme oft unzuverlässig und nicht den Erwartungen entsprechend. Eine empirische Zahl sagt, dass im Durchschnitt in erstausgelieferten Softwaresystemen in 100 Statements etwa 2 unentdeckte Fehler enthalten sind.

Das in diesem Zusammenhang verwendete Synonym "testen **gegen** ..." gibt einen ersten Eindruck von dem Kampf, den der Programmierer gegen die Unwirtlichkeiten der Praxis führt. Findet noch dazu der Test der neu entwickelten Software auf einer neu entwickelten Hardware statt, dann ist Test und Fehlerbeseitigung ungleich schwieriger. So ist der Test ein destruktiver, zersetzender Prozess, Myers [11] nennt ihn sogar einen sadistischen Prozess. Der Vorgang des Testens vermittelt wenig Erfolgserlebnisse. Sind keine Fehler nachzuweisen, so war die Tätigkeit unproduktiv; sind Fehler nachzuweisen, so manifestiert dies einen Misserfolg früherer Arbeiten.

Speziell der Softwaretest wird auch heutzutage noch oft rein empirisch durchgeführt. Nur wenige Programmierer haben konkrete Vorstellungen davon, auf welche Art und Weise ein systematischer Test durchzuführen ist. Das endgültige Erfolgserlebnis bleibt aus, denn nach Dijkstra läßt sich in komplexen Algorithmen zwar die Anwesenheit von Fehlern, nie aber deren Abwesenheit nachweisen; ein endgültiger formaler Korrektheitsbeweis ist also nicht möglich.

Mit dem Entschluss des Programmierers, an einer subjektiv gewählten Stelle im Programm und zu einem subjektiv gewählten Zeitpunkt den Test abzubrechen, wird also eine Toleranzschwelle festgelegt, die das Vertrauen in die eigene Arbeit auf der einen Seite und die Angst vor der Überschreitung von Zeit, Kosten und anderen Faktoren auf der anderen Seite wieder spiegelt. So dient der spätere Systembenutzer als Versuchskaninchen. Bei ihm auftretende, vorher nicht entdeckte Fehler werden in einem bestimmten Zeitraum kostenlos beseitigt.

Einige Definitionen sollen zunächst eine gemeinsame Grundlage schaffen:

Der allgemeine Begriff **Softwaretest** ist ein Prozess, der versucht, durch Programmausführung mit Testdaten Fehler nachzuweisen.

Der **Modultest** bezieht sich auf das Innere einzelner Module.

Der **Integrationstest** bezieht sich auf das gesamte Softwaresystem, also speziell auf die in der Planungsphase festgelegten Schnittstellen zwischen den Modulen.

Der **Installationstest** prüft das Softwaresystem in seiner simulierten oder realen Umgebung.

Der **Systemtest** achtet auch auf die Erfüllung der im Pflichtenheft festgelegten Anfangsziele.

Der **Akzeptanztest** schließlich stellt die Reaktion der Benutzer auf das neue System fest.

Der Begriff des Softwaretests ist grundsätzlich zu unterscheiden vom häufig gebrauchten Begriff des “Debugging“. Während der Softwaretest zum Ziel hat, Fehlfunktionen im Programmsystem nachzuweisen, ist es das Ziel des Debugging, die gefundenen Fehler auf ihren Ursprung zu lokalisieren und zu beseitigen.

Dijkstra zeigt die Wichtigkeit von korrekten Softwaremodulen anhand einer empirischen Formel. Sei p (p = (O..1)) ein Maß für die Korrektheit eines Moduls.

p = 0 entspreche "vollkommen falsch" und p = 1 entspreche "vollkommen richtig".

So ist die Korrektheit P des gesamten Softwaresystems aus N Modulen gegeben durch

$$P=p^N .$$

Um bei großen N (z. B. N = 100) die Korrektheit des Gesamtsystems noch signifikant von 0 abweichen zu lassen, muss die Korrektheit der Einzelmodule sehr nahe an 1 liegen.

Bevor die einzelnen Testarten genauer besprochen werden, seien noch einige Praxishinweise in Form von Testaxiomen nebst zugehörigen Bemerkungen vorangestellt. Sie sollen dem Anwender helfen, sich vor dem Test ein paar grundsätzliche Gedanken über eine gezielte Vorgehensweise zu machen und elementare Fehler zu vermeiden.

Testen ist eine Aufgabe für höchst kreative Programmierer.

Zum Softwaretest gehört weit mehr als zur Softwareentwicklung ein großes Maß an Fantasie und ein durch Erfahrung geschultes Gespür für die Wahl besonders empfindlicher Testdaten. Ein guter Softwaretester kann für ein Testobjekt diejenigen Eingabedaten ermitteln, die mit größter Wahrscheinlichkeit einen Fehler hervorrufen.

Testbarkeit sollte schon Ziel der vorherigen Phasen sein.

Die Ergebnisse jeder Softwareentwicklungsphase umfassen auch Überlegungen und Aussagen zu deren Test. Dies gilt sowohl für die Testbarkeit an sich (Anforderungen, die nicht testbar sind, sind wertlos), wie auch für die geeignete Vorgehensweise.

Ein guter Testfall versucht mit hoher Wahrscheinlichkeit bisher unerkannte Fehler zu finden, nicht aber einen Korrektheitsbeweis zu führen.

Da ein Korrektheitsbeweis für komplexe Software nach heutigen Erkenntnissen prinzipiell nicht durchführbar ist, kann er nicht Ziel des Tests sein. Es ist durch gutwillige Wahl von Testdaten möglich, ein Programmpaket auch tagelang zu testen, ohne dass ein Fehler auftritt, jedoch ist dies verschwendete Zeit. Vielmehr sind die böswilligen Testdaten interessant, die die Software an Grenzsituationen bringen, die sie beherrschen müsste.

Es ist unmöglich, das eigene Programm richtig zu testen.

Der Softwareentwickler selbst verliert nach einiger Zeit den Blick für die Fehler im eigenen Programm, da er in gewisser Weise von der Richtigkeit seiner eigenen Gedankenstrukturen überzeugt ist. Testet ein anderer den Algorithmus, so besitzt dieser genügend Skepsis und Abstand, um objektiv zu testen; allerdings dauert der Test wegen der Einarbeitungszeit länger.

Vor dem Testfall die erwarteten Ziele (Ergebnisse, Outputs, Reaktionen, ...) festlegen.

Ein alter Testfehler ist die (oft oberflächliche) Analyse von Testergebnissen nach einem durchgeführten Testfall. Richtig ist vielmehr die Erarbeitung der Testdaten und der erwarteten Ergebnisse vor der Testausführung. Die Arbeit nach dem Testfall reduziert sich auf den Vergleich mit dem Erwarteten und die Klärung eventueller Abweichungen.

Testfälle reproduzierbar wählen.

Diese eigentlich selbstverständliche Regel ist speziell in Echtzeitsystemen manchmal schwer zu realisieren. Jedoch: ein Testfall, der einmal einen Fehler nachweist, jedoch nicht reproduzierbar ist, ist wertlos.

Niemals ein Programm verändern, damit der Test einfacher, wird. Der Test hat immer mit dem Originalprogramm stattzufinden. Die Wahrscheinlichkeit ist hoch, mit der Programmvereinfachung oder -modifikation gerade diejenigen Stellen vorübergehend auszuschalten, die fehlerbehaftet sind.

Nicht "zwischendurch" testen.

Test ist eine höchst ernsthafte Tätigkeit, die nicht "mal eben" zwischen der Erstellung eines Moduls und des nächsten oder zwischen der Kaffeepause und dem Mittagessen durchführbar ist.

7.2 Modultest: Vorschlag zur Vorgehensweise

Eine prinzipielle Betrachtung der systematischen Vorgehensweise zum möglichst vollständigen Modultest liefert zwei grundsätzlich verschiedene Verfahren: das **White-Box-Verfahren** und das **Black-Box-Verfahren.**

Es fällt nicht schwer, die Vorgehensweise dieser Verfahren zu assoziieren. Ein Modul wird im Black-Box-Verfahren allein gegen die Einhaltung aller Spezifikationen der Exportschnittstelle getestet. Das White-Box-Verfahren prüft den Ablauf im Modulinneren. Es scheint zunächst von der Theorie her ausreichend, einen Black-Box-Test durchzuführen, denn das Modul ist ja definitionsgemäß allein durch den Nachweis der vollständigen Funktion der Exportschnittstelle funktionsfähig. Diese Idee scheitert jedoch an der Praxis, denn dieser Nachweis ist mit Hilfe des Black-Box-Verfahrens nur durch den vollständigen Test aller Eingabekombinationsmöglichkeiten der Exportschnittstelle durchführbar, ein Ansatz, zu dem es schon bei schmalen Schnittstellen an Zeit fehlt. Ein einfaches überzogenes Beispiel soll diese Situation verdeutlichen. Ein Modul mit der Schnittstellenspezifikation

```
ENTRY multiply  (IN  multiplikand:   INTEGER,
                 IN  multiplikator:  INTEGER,
                 OUT Produkt:        INTEGER);
```

ist nach dem Black-Box-Verfahren nur nach Ausführung aller Kombinationen von (multiplikand * multiplikator) vollständig getestet. Der nach dem prinzipiellen Verfahren unbekannte, da im Modulinneren versteckte Multiplizieralgorithmus könnte theoretisch bestimmte Kombinationen falsch berechnen. Die konsequente Anwendung des Black-Box-Tests führt schon bei diesem kleinen Beispiel zu Testkombinationen von astronomischer Größe.

Der **White-Box**-Test geht von einer Offenlegung des Modulinneren aus. Mit seiner Hilfe ist es also möglich, von einem gut gewählten und positiven Testfall über den aufgedeckten Algorithmus auf die Funktionsfähigkeit einer bestimmten Gruppe von Eingangskombinationen zu schließen. Jedoch sollte auch der White-Box-Test nicht allein verwendet werden, denn er berücksichtigt zu wenig den Test; gegen die Modulspezifikation, da er sich auf das Modulinnere bezieht. Ein geeigneter, möglichst vollständiger Modultest muss also eine Kombination aus beiden Verfahren verwenden.

Der folgende konkrete Vorschlag für einen möglichst umfassenden, aber nicht zu aufwendigen Modultest lehnt sich an Myers [11] an und sieht vier Schritte vor:

Test gegen die Modulspezifikation

Der Modultest beginnt mit dem Black-Box-Verfahren

Jedoch werden natürlich nicht alle Kombinationen durchgetestet, sondern es erfolgt eine Aufteilung in

- einen typischen Testfall für jede Eingangs- und Ausgangsgröße
- je einen Testfall für die Grenzen von Eingangs- und Ausgangsgrößen

- je einen Testfall für alle logischen Eingangs- und Ausgangsgrößen, speziell für die Fehlerausgänge.

Test aller internen Pfade

Dies ist ein typischer White-Box-Test. Durch geeignete Wahl der Eingangsgrößen soll im Modulinneren jeder Kontrollflusspfad mindestens einmal durchlaufen werden. Dies verhindert unter anderem die Existenz von Codebereichen, die prinzipiell niemals erreichbar sind.

Test aller Wiederholungen

In einem weiteren White-Box-Test werden alle im Modulinneren vorkommenden Wiederholungen durch geeignete Wahl der Eingangsgrößen jeweils auf die minimal und maximal mögliche Zahl der Durchschläge (Minimum und Maximum der Iterationen) überprüft.

Dies verhindert weitestgehend die falsche Formulierung von Wiederholungsbedingungen.

Codeinspektion auf Sensibilität

Dieser vierte Punkt ist ein rein intuitiver Prozess. Hier werden keine formalen Spezifikationen oder Bedingungen überprüft, sondern es erfolgt eine Betrachtung des Codes nach Kriterien, die auf Erfahrung beruhen und stark von den Umgebungsbedingungen wie etwa der verwendeten Programmiersprache abhängen.

Es ist leicht einsehbar, dass die genannten Punkte nur dann nutzbringend einsetzbar sind, wenn schon in den vorherigen Entwicklungsphasen die Prinzipien des Software Engineering konsequent angewandt wurden. So setzt der Black-Box-Test eine Beschreibung der Modulschnittstellen wie in Kapitel 5 beschrieben voraus. Der White-Box-Test ist nur dann verwendbar, wenn ausschließlich lineare Kontrollstrukturen nach dem Prinzip der strukturierten Programmierung (Kapitel 6) verwendet wurden.

Wie viele andere Teile der Softwareproduktion ist heute auch der Modultest durch geeignete Werkzeuge, die auf dem Digitalrechner selbst installiert sind, unterstützbar. Diese oft sehr aufwendigen Werkzeuge lassen sich prinzipiell in die Kategorien **Testfallgenerator** und **Programmanalysator** unterteilen.

Ein Testfallgenerator sorgt anhand der formalen vom Programmierer erstellten Modulspezifikation und der Programmquelle des Moduls für die automatische Generierung von geeigneten Testfällen und Testdaten. Er führt automatisch die Testfälle durch und vergleicht die Resultate mit den Sollvorgaben. Anhand eines Protokolls erhält der Programmierer Auskunft über aufgetretene Fehler und über die Korrektheitswahrscheinlichkeit des Moduls.

Programmanalysatoren, wie z. B. das weit verbreitete Werkzeug LINT unter UNIX) überprüfen das Innere eines zu testenden Moduls. Sie erstellen Protokolle über eine Strukturanalyse, machen auf schlecht formulierte Bedingungen oder niemals erreichbare Statements aufmerksam und vieles mehr.

7.3 Der Integrationstest

Während der Modultest den Nachweis von Fehlern im Inneren einzelner Module zum Ziel hat, ist die Aufgabe des Integrationstests der Nachweis von Fehlern bei der Zusammensetzung des Gesamtsystems auf die Art, wie es im hierarchischen Entwurf vorgesehen war. In der Praxis finden vier verschiedene Verfahren Einsatz.

Das Verfahren 1 ist vom Standpunkt eines schrittweisen und gesicherten Vorgehens abzulehnen, findet jedoch in der Praxis aus Bequemlichkeit oder Unwissenheit häufig Verwendung. Es werden nach diesem Verfahren alle Module auf einmal zusammengesetzt und das so entstandene Gesamtsystem auf Funktion getestet. Aus dem dabei in der Regel entstehenden Ergebnis leitet sich lautmalerisch der Name ab: es heißt **big-bang-Verfahren**.

Es ist also richtiger, die Module Stück für Stück zusammenzusetzen und die entstehenden Teilsysteme auf Funktionsfähigkeit zu testen. Dies kann prinzipiell (vgl. Bild 5.1) bottom-up oder top-down geschehen, woraus sich die beiden gleichnamigen Verfahren ableiten. Es ist vom praktischen Standpunkt sinnvoller, die Module in der umgekehrten Reihenfolge ihrer Entwicklung zusammenzusetzen, also das bottom-up-Verfahren zu verwenden. Man beginnt mit den basismaschinennächsten Modulen und arbeitet sich ebenenweise der Benutzermaschine entgegen. Auf jeder Ebene wird die durchgeführte Teilintegration getestet, indem sog. **driver** (wörtl. Treiber) die Exportschnittstellen der jeweils benutzermaschinennächsten Module simulieren. Diese driver erlauben, Testdaten in das Teilsystem einzuspeisen. So entsteht mit der Zeit ein System, das immer leistungsfähiger wird. Speziell in der Prozessdatenverarbeitung bietet sich diese Vorgehensweise an, da hier oft Teile der Basismaschine durch die "Hardware selbst (Sensoren, Schalter, ...) repräsentiert werden und vorteilhaft integrierbar sind.

Ein schwerwiegender Nachteil des bottom-up-Verfahrens ist jedoch das Zeitproblem der Softwareherstellung. Die konsequente Anwendung des bottom-up-Tests nach einer top-down-Entwicklung setzt voraus, dass die vollständige Modulrealisierung und deren Test abgeschlossen ist, bevor der Integrationstest beginnen kann. Dieser der Parallelarbeit widersprechende und Leerlauf erzeugende Grund führt dann doch häufig zur Anwendung des **top-down-Verfahrens** beim Integrationstest.

Ähnlich dem bottom-up-Verfahren werden beim top-down-Verfahren die soeben realisierten und getesteten Module ebenenweise (jedoch diesmal top-down) zusammengesetzt. Die Importschnittstellen der jeweiligen Test-Enden werden durch sog. **stubs** (wörtl. Baumstümpfe) simuliert, die im einfachsten Fall die übergebenen Daten entgegennehmen und fest voreingestellte oder über eine Konsole eingebbare "Ergebnisdaten" zurückliefern. So entsteht schrittweise ein Gesamtsystem, das im letzten Schritt durch Anbindung an die Basismaschine eigenständig wird.

Das **sandwich-Verfahren** versucht bei Bedarf die Vorteile des top-down- und des bottom-up-Verfahrens zu vereinen. Die generelle Testrichtung ist die des top-down, jedoch wird dieser Richtung an bestimmten Stellen, die z. B. von allen Systemteilen intensiv genutzt werden oder bei denen es aus anderen Gründen ratsam erscheint, bottom-up entgegengearbeitet. In Teilen braucht man bei diesem Verfahren sowohl

driver wie auch stubs; sie können eventuell beim Erreichen einer entsprechenden Annäherung zu Verbindern verschmelzen. Schon dieser Kurzüberblick zeugt von der Notwendigkeit, sich bereits während der Planungsphase intensive Gedanken über den Integrationstest zu machen. Neben fachlichen Einwirkungen haben diese Überlegungen Einfluss auf die Arbeitseinteilung, d. h. auf die Priorität, in der die einzelnen Module bearbeitet und fertig gestellt werden müssen, um rechtzeitig für den Test verwendbar zu sein.

7.4 Der Systemtest

Ein vom Datenverarbeitungsstandpunkt betrachtet lauffähiges und mit hoher Wahrscheinlichkeit in weiten Teilen korrektes Softwaresystem besteht seine schwerste Prüfung im Systemtest. Das eigentlich fertige Produkt wird in seine reale oder zunächst simulierte Umgebung eingesetzt. Getestet wird gegen die Spezifikationen, die vor längerer Zeit im Pflichtenheft festgelegt wurden. Hier zeigt sich, ob das neue System die technischen, kaufmännischen und kommerziellen Ziele erfüllt, wegen derer man sich überhaupt zu seinem Bau entschlossen hat.

Je nach Anwendung des neuen Softwaresystems werden weitere Tests durchgeführt, die hier nur in Stichworten genannt werden. Hat das System eine Schnittstelle zum menschlichen Benutzer, so beobachtet ein Akzeptanztest dessen zustimmende oder ablehnende Reaktionen. Ein Test des Systems unter größtmöglicher Belastung kann dessen Verhalten zeigen, wenn man alle möglichen Situationen, deren gleichzeitiges Auftreten zwar unwahrscheinlich, aber nicht unmöglich ist, gesteuert gleichzeitig auftreten lässt. Ein Test gegen die vom System entwickelte Datensicherheit ergibt ein Maß für seine Fähigkeit, unbefugte Zugriffe zu internen Daten privater Natur zu verhindern. Zuverlässigkeitstests, vor allem bei hochverfügbaren Softwaresystemen (oft in Verbindung mit hochverfügbarer Hardware) ergeben ein statistisches Maß für die Ausfallwahrscheinlichkeit im unrechten Moment.

7.5 Fehlerbeseitigung

Auch das Debugging, also die Suche, Lokalisierung und Beseitigung erkannter Fehler, wird oft vom Rechner selbst unterstützt. Beim Fehlen jeglicher anderer Hilfsmittel dienen - zumindest bei problemorientierter Programmiersprache - an zentralen Stellen im Programm zu Testzwecken eingebaute “printf“-statements als bewährtes Mittel, die Inhalte von Variablen und Parametern sowie den Kontrollfluss anzuzeigen. Man sollte jedoch auf diese Methode nur im Notfall zurückgreifen, denn sie ist mühsam durchführbar und verstößt noch dazu gegen den Grundsatz, ein zu testendes Programm zum Zwecke des Tests nicht zu verändern.

Sehr viel komfortabler ist die Verwendung eines so genannten Debug-Systems, das die Möglichkeiten der Haltepunkte und des Trace unterstützt. Arbeitet ein derartiges System in einer Entwicklungsumgebung mit einem Compiler zusammen, so kann es oft auf die symbolischen Namen und die Zeilennummern des Quellprogramms zurückgreifen, denn der Compiler erzeugt dann spezielle Information für das Debug--

System. Unter Benutzerkontrolle sind dann definierte Programmstücke des Testmoduls abarbeitbar, die Abarbeitung ist beim Auftreten bestimmter Bedingungen unterbrechbar und die Inhalte von Variablen sind darstellbar.

Zwei Probleme sind mit dieser Vorgehensweise verbunden. Zum einen ist ein solches Debug-System wesentliche wertloser, wenn man nicht auf der gleichen Sprachebene die Fehler suchen kann, in der das Programm entwickelt wurde. Speziell für Mikrorechner werden oft Compiler für problemorientierte Programmiersprachen angeboten; jedoch die Fehlersuche in den so erstellten Programmen ist auf Assemblerebene durchzuführen. Dies ist um so schlimmer, als dass der Programmierer im Zweifelsfall gezwungen ist, sich neben seinem eigentlichen Problem der Fehlerbeseitigung in die Codegenerierung des benutzten Compilers einzudenken, eine Arbeit, die er gerade durch Wahl der Programmiersprache vermeiden wollte.

Das zweite Problem betrifft besonders die Prozessdatenverarbeitung. Die unter Debug-Kontrolle ablaufenden Programme laufen oft langsamer, d. h. sie laufen nicht "in Echtzeit", denn das Debug-System verbraucht selbst Rechenzeit z. B. zur Abprüfung von Haltepunktbedingungen. Bestimmte hardwarenahe Programme sind jedoch darauf angewiesen, mit voller Rechnergeschwindigkeit abzulaufen. Unter diesen Umständen ist ein derartiges Debug-System nicht einsetzbar. Sehr komfortabel, aber auch recht teuer sind so genannte **Echtzeit-Emulatoren**.

Ihr Prinzip ist es, für die ablaufende zu testende Software einen eigenen Prozessor zur Verfügung zu stellen, auf dem das Programm unter allen Umständen in Echtzeit ablaufen kann. Große Teile des Debug-Systems (z. B. die Haltepunkterkennung) sind dann in zusätzlicher Hardware realisiert, um das Programm unter keinen Umständen vor Zutreffen einer Haltepunktbedingung zu unterbrechen oder zu verlangsamen. Weiterhin werden oft die letzten n ausgeführten Programmschritte archiviert und bei Bedarf aus einem speziellen History-Speicher zurückgeholt. Diese Art des kombinierten Software- und Hardwaretests hat sich vor allem bei der Entwicklung von schlüsselfertigen Geräten, die eine Mikrorechnersteuerung besitzen, durchgesetzt.