15 Fehlerlokalisierung und -korrektur

15.1 Problemstellung, Lösungsstrategien und Wissensarten

Fehlerbehebung ist ein zweiteiliger Prozeß, der nach der Durchführung erfolgreicher Tests, d. h. nach Hinweisen auf die Existenz von Fehlern, einsetzt:

Teil I: Bestimmung der exakten Natur des Fehlers und Lokalisierung des vermuteten Fehlers im Programmtext (ca. 95% des Aufwands).

Teil II: Korrektur des Fehlers (ca. 5% des Aufwands).

Wegen des geringen Aufwands für die Fehlerkorrektur wird im folgenden fast nur die Fehlerlokalisierung betrachtet.

Betrachtet man den Prozeß der Fehlerbehebung genauer, so sind folgende Schritte notwendig:

1. Erkennen der Fehlersymptome durch einen Soll/Ist-Vergleich von spezifiziertem (bzw. erwartetem) und implementiertem Verhalten.
2. Auffinden der Fehlerursachen, indem Funktion und Struktur des Programms, insbesondere die Programmausführung und -steuerung, betrachtet wird.
3. Effizientes und korrektes Abändern des Programms, wobei die Abänderung wiederum zu überprüfen ist, um sicherzustellen, daß der Fehler korrigiert wurde und daß keine weiteren Fehler erzeugt wurden.

Die Fehlerbehebung wird ungern ausgeführt, da bei Programmierern aus folgenden Gründen eine psychologisch bedingte Abneigung besteht:

1. Durch gefundene Fehler wird das Selbstwertgefühl empfindlich getroffen.
2. Fehler müssen meistens unter Zeitdruck behoben werden, und zwar
   (a) wegen der meist zu knapp kalkulierten Liefertermine,
   (b) durch vom Programmierer selbst erzeugten Druck, da er die „leidigen“ Fehler möglichst schnell (wegen 1.) beheben will.

3. Fehlerbehebung ist intellektuell anstrengend; bei anderen Ingenieurdisziplinen (z. B. Fahrzeugbau) ist sie dagegen meistens einfach. Wenn ein Auto zum Beispiel öfter stehen bleibt (Symptom), kann man Autoradio und Tachometer als Fehlerquelle ausschließen und den Motorbereich als Fehlerquelle eingrenzen, wobei wiederum Wasserpumpe und Ölfilter ausgeschlossen werden können[1]. Bei Computerprogrammen kann dagegen die Fehlerursache für ein Fehlersymptom praktisch in jedem Programmteil liegen, in der Eingabe, in der Verarbeitung oder in der Ausgabe. Die Lokalisierung ist also fast hoffnungslos.

Glücklicherweise gibt es aber auch für die Lokalisierung von Fehlern in Programmen hilfreiche Methoden. Sie beruhen darauf, daß man das Wissen über das Programm und die möglichen Fehler systematisch sammelt, erweitert und strukturiert. Folgende Wissensarten sind dabei relevant: W1) Wissen über das gewünschte (spezifizierte) Programm, W2) Wissen über das vorliegende Programm, W3) Verständnis der Programmiersprache, W4) allgemeine Programmier-Erfahrung, W5) Wissen über den Anwendungsbereich und W6) Kenntnis der (üblichen) Fehler.

Das Wissen kann sich dabei auf vier verschiedene Aspekte bzw. Ebenen eines Programms beziehen:

(a) Struktur des Programms, d. h. Komponenten und ihr Zusammenhang (z. B. der Kontrollfluß- und Datenflußgraph),

(b) Verhalten des Programms, d. h. die Folge der Berechnungen bei bestimmten Eingaben,

(c) Funktion des Programms, d. h. die Relation der Ausgaben (und Zustandsveränderungen) zu den Eingaben,

(d) Klassifikation des Verhaltens nach bestimmten Mustern (pattern matching).

Die Fehlerbehebungsmethoden beziehen sich auf die verschiedenen genannten Wissensarten und -ebenen und richten sich dabei nach einer oder mehrerer der folgenden Strategien:

S1. Bei der **Verifikationsstrategie** wird das vorliegende Programm (W2) mit der Spezifikation des Programms bzw. mit dem gewünschten Programm (W1) bzgl. Berechnungsäquivalenz verglichen [siehe Ebenen (b), (c), (d)].
Diese aufwendige Strategie verlangt viel Wissen über das gewünschte Programm und ist daher sehr aufwendig. Dieser Aufwand lohnt sich i. allg. nur bei Tutoren-Systemen, welche die Lösungen von Programmieranfängern zu vorgegebenen

[1] Generell lassen sich meist **Fehlerbäume** aufstellen, welche die Fehlerpropagierung zwischen strukturellen Komponenten beschreiben. Damit werden Regeln der Art „wenn Teil 1 und Teil 2 und ... Teil n defekt sind, dann ist Teil A defekt“ (bzw. entsprechendes mit „oder“ statt „und“) repräsentiert (genaueres siehe z. B. in [NaV 87] und der dort zitierten Literatur).

Übungsbeispielen überprüfen sollen. Eine Alternative dazu sind vereinfachte Spezifikationen in Form von Zusicherungen.

S2. Bei der **Strukturkontrollstrategie** wird die konsistente Verwendung von Programmkonstrukten [z. B. Bezeichner, Typen; siehe W3 und Ebene (a)] überprüft, evtl. auch die Verträglichkeit mit übergeordneten Programmbauplänen (W5).
Der erste Teil kann nur *potentiell* fehleranfällige Programmstellen aufdecken, der zweite Teil setzt wieder eine aufwendige Wissensbasis und -verarbeitung voraus.

S3. Mit der **Filterungsstrategie** werden diejenigen Programmteile identifiziert, die beobachtetes Fehlverhalten *nicht* verursacht haben können bzw. (als Komplement) die Teile, die einen Fehler enthalten können. Dazu sind die Testabläufe zu analysieren, d. h. die durchlaufenen Kontrollflußwege und die veränderten Variablenwerte [siehe Ebene (b)].
Wird diese Strategie zuerst eingesetzt, können die anderen Strategien auf einer Teilmenge der zu betrachteten Programmkonstrukte operieren und damit Aufwand einsparen.

S4. Mit der **Strategie zur Erkennung stereotyper Fehler** werden (bekannte) Fehler erkannt, bei denen ein klarer Zusammenhang zwischen dem Fehlersymptom und der Fehlerursache besteht.
Dies setzt eine Wissensbasis entsprechender Fehler voraus, die natürlich nur eine Teilmenge der tatsächlich vorkommenden Fehler umfassen kann.

Im folgenden Unterkapitel werden einige Fehlerlokalisierungsmethoden vorgestellt, die sich an obigen Wissensarten, -ebenen und Strategien orientieren.

15.2 Methoden der Fehlerlokalisierung

„Das Gefühl findet, der Scharfsinn weiß die Gründe."
— Jean Paul

Für das Problemlösen allgemein und also auch für das Problem der Fehlerlokalisierung gibt es zwei Methoden, die auf logischem Denken beruhen, sowie weitere Methoden (und Werkzeuge) zur genaueren Lokalisierung der Fehler. Das logische Denken kann induktiv oder deduktiv betrieben werden.

1. Induktion
   Induktives Vorgehen bedeutet das Schließen vom Besonderen auf das Allgemeine. Bei der Fehlerlokalisierung sind Hinweise auf Fehlersymptome in einem oder mehreren Testläufen das „Besondere" und Verbindungen zwischen den Symptomen sowie die Umstände, unter denen ein Fehler auftritt, sind das „Allgemeine".

2. Deduktion
   Deduktives Vorgehen bedeutet das Schließen vom Allgemeinen auf das Besondere. Bei der Fehlerlokalisierung sind allgemeine Hypothesen (Theorien bzw. Prämissen) über die Fehler(ursachen) das „Allgemeine“ und die Eliminierung von Möglichkeiten, die Verfeinerung der Hypothesen (anhand von Beispielen) sowie das Schließen auf den vorliegenden Fall ist das „Besondere“.

Das deduktive und induktive Verfahren der Fehlerlokalisierung läßt sich als Flußdiagramm formulieren (siehe Abbildung 15.1).

15.2.1 Induktionsmethode

Die Induktionsmethode umfaßt folgende Schritte (vgl. Abb. 15.1 links):

Schritt 1: Alle Symptome mit Hilfe der (Test-)Daten zusammenstellen.

Dabei ist folgendes zu beachten:

(a) Günstig sind ähnliche, aber unterschiedliche Testdaten, um den Fehler genauer einzugrenzen.

(b) Bei den Testdaten sollte vorab geklärt werden, ob sie zulässig sind, d. h.:

- Werden nur Testdaten mit Eingabedaten innerhalb des Definitionsbereichs verwendet?
- Werden die richtige Programmversion und die richtige Version der Daten verwendet, d. h., sind Handhabungsfehler (beim „Operating“) ausgeschlossen?

Schritt 2: Die Abweichung zwischen dem Programmverhalten und der Programmspezifikation ist zu beschreiben bzw. zu klassifizieren.

Als Hilfsmittel dient dabei eine Symptom-Tabelle mit acht Feldern, d. h.

- vier Zeilen mit den Fragen „was“, „wann“, „wo“, „welcher Umfang“,
- zwei Spalten mit den Feststellungen „ist vorhanden“, „ist nicht vorhanden“.

Die Einteilung dient dazu, verschiedene Aspekte des Fehlers zu klassifizieren:

- Was ist der Fehler überhaupt (das Symptom)?
- Wann (bei welchen Tests bzw. Situationen) tritt der Fehler auf?
- Wo tritt der Fehler auf, d. h. in welchen Ausgabeteilen?
- In welchem Umfang tritt der Fehler auf (immer, manchmal, „zufällig“)?

Obige Erklärungen beziehen sich auf die „ist vorhanden“-Spalte. Die „ist nicht vorhanden“-Spalte enthält Angaben, wo und wann dieser oder ähnliche Fehler *nicht* auftreten. (Dies entspricht der Filterungsstrategie.)

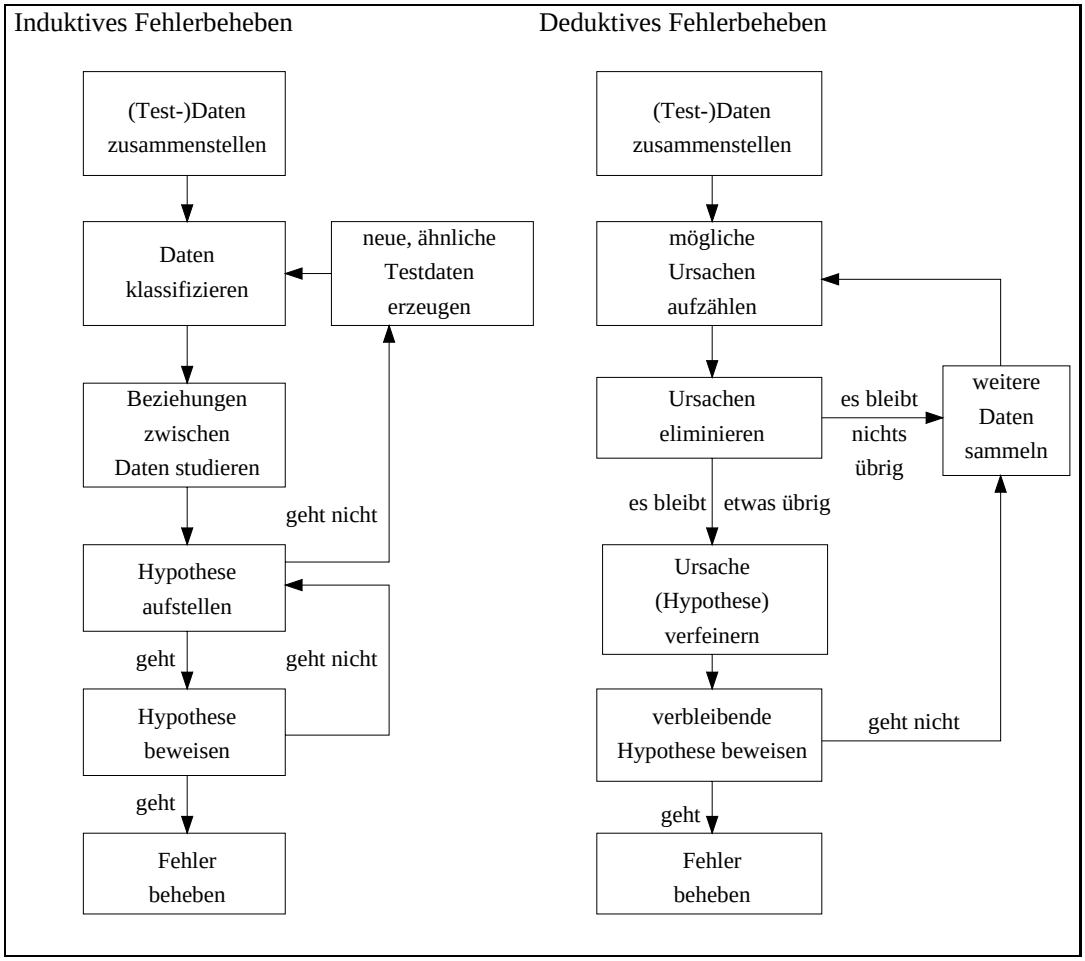

Abb. 15.1: Flußdiagramm zur Darstellung von induktiver und deduktiver Vorgehensweise (nach [Mye 79], Bild 7.1 und 7.4)

Beispiel 15.2.1
Eine Symptomklassifizierung kann etwa folgendermaßen aussehen:

was:	*ist vorhanden:*	*in Sätzen vom Typ A (Fehler)*
	ist nicht vorhanden:	*in Sätzen vom Typ B (kein Fehler)*
wo:	*ist vorhanden:*	*im ganzen zweiten Bericht*
Umfang:	*ist nicht vorhanden:*	*kein vorübergehender zeitweiser Fehler*

Schritt 3: Die Beziehungen zwischen den Daten sind zu studieren.

Dieser Schritt hat das Ziel, Unterschiede zwischen der „ist vorhanden"- und der „ist nicht vorhanden"-Spalte festzustellen.
Dies ist der wichtigste Schritt der Methode. Wenn die Fälle, wo der Fehler auftritt (und wo nicht), klar abgegrenzt werden können, dann kann i. allg. auch die Fehlerursache eingegrenzt werden.

Schritt 4: Eine Hypothese über die Fehlerursachen ist aufzustellen.

Das Aufstellen einer Hypothese ist wichtig, weil damit das weitere Suchen eingegrenzt wird. Dabei ist die Erfahrung des Programmierers bzw. Testers gefordert, um aus den aufbereiteten Testdaten auf eine mögliche Fehlerursache zu schließen. (Dies entspricht der Erkennung stereotyper Fehler.)
Eventuell müssen die in Schritt 3 erkannten Unterschiede verschärft werden durch neue Überlegungen oder neue Testdaten, damit alle acht Felder der Tabelle (gleichmäßig stark) ausgefüllt sind.

Schritt 5: Die Hypothese ist zu beweisen.

Dies umfaßt folgende Teilschritte:

(a) Es ist zu überprüfen, ob an der vermuteten Stelle im Programm wirklich ein Fehler existiert.

(b) Es ist zu verifizieren, daß die entdeckte Fehlerquelle alle beobachteten Fehlersymptome vollständig erklärt (durch manuellen „Walkthrough" mit einfachen Testdaten).

Dabei ist folgendes zu beachten:
Wenn die beobachtete Abweichung bei der Überprüfung tatsächlich die bisher ermittelten Fehlersymptome vollständig erklärt, dann ist der Fehler gefunden worden. Andernfalls hat man nur einen Teil der Fehlerursache gefunden oder gar keine Fehlerquelle. Dann sind neue, geänderte Hypothesen aufzustellen.
In keinem Fall sollte man diesen Schritt auslassen nach der Devise

> „Es sieht so aus, als ob wir den Fehler gefunden haben. Wir probieren 'mal die folgende Korrektur und sehen dann nach, was passiert."

BEISPIEL 15.2.2 (FÜR DIE INDUKTIONSMETHODE)
Prüfungsauswertungsprogramm (nach [Mye 79])

Das Programm druckt den Median, den Mittelwert und die Standardabweichung der Noten, die als Punktwerte angegeben sind.

Schritt 1: *Symptome zusammenstellen*

Für 51 Studierende ergibt sich ein Mittelwert von 73,2 (korrekt), ein Median von 26 (falsch; 82 ist richtig) und eine Standardabweichung von 10,3 (korrekt). Für einen Studierenden ergibt sich ein Mittelwert von 68 (korrekt), ein Median von 1 (falsch) und eine Standardabweichung von 0 (korrekt). Für zwei und 200 Studierende werden Mittelwert, Median und Standardabweichung richtig berechnet. Die fehlerhaften Berechnungen treten (nur) in Report 3 auf[2].

Schritt 2: *Fehlersymptome bzw. Abweichungen klassifizieren*

Die in Schritt 1 gesammelten Symptome ergeben folgende Eintragungen in die Symptom-Tabelle (s. Tabelle 15.1).

Fragestellung	Symptom „ist" vorhanden	Symptom „ist nicht" vorh.
was	Der Median in Report 3 ist falsch.	Mittelwert und Standardabweichung sind korrekt.
wo	nur in Report 3	in den anderen Berichten
wann	für 1 und 51 Studierende	für 2 und 200 Studierende
in welchem Umfang	Median = 26 für 51 Studierende, Median = 1 für einen Studierenden	*Median* scheint *unabhängig* von den Noten (Punktwerten) zu sein

Tab. 15.1 Klassifikation der Fehlersymptome bei induktivem Vorgehen

Die Testdaten von Schritt 1 lassen im Feld „in welchem Umfang; Symptom ist nicht vorhanden" keinen gesicherten *Eintrag zu. Daher muß ein neues Testdatum mit ähnlichen Werten [siehe Schritt 1.(a)], d. h. mit 51 bzw. einem Studierenden, aber anderen Noten (Punktwerten) durchgeführt werden. Wenn sich dann wieder als Median 26 bzw. 1 ergibt, kann folgende Hypothese aufgestellt werden:*

„Median ist nicht abhängig von den Noten (Punktwerten)".

Schritt 3: *Beziehungen zwischen Daten studieren*

Dieser Schritt hat folgendes Ergebnis:

(a) *Nur die Medianberechnung (in Report 3) ist falsch („was"- und „wo"-Zeile).*

(b) *Die Medianberechnung ist nur für ungerade Anzahlen von Studierenden (1, 51) falsch, nicht für gerade Anzahlen („wann"-Zeile).*

[2] Dies ist allerdings nicht verwunderlich, denn nur in Report 3 werden Mittelwert, Median und Standardabweichung ausgegeben.

(c) *Der Median ist (unabhängig von den Noten) gleich der aufgerundeten halben Studierendenanzahl* ($\lceil \frac{51}{2} \rceil = 26$; $\lceil \frac{1}{2} \rceil = 1$; *siehe „Umfang"-Zeile).*

Schritt 4: *Hypothese aufstellen:*
Die Noten der Studierenden sind in einer Liste aufsteigend sortiert abgelegt. Das Programm druckt die (Ordnungs-)Nummer des mittleren Studierenden[3] *und nicht seine Note aus.*

Schritt 5: *Die Hypothese aus Schritt 4 ist anhand des Programmcodes zu beweisen.*

Das Vorgehen beim induktiven Fehlerlokalisieren entspricht dem folgenden Vorgehen beim Lösen eines Mordfalls:

1. sorgfältiges Analysieren der Indizien,
2. Zusammensetzen anscheinend unbedeutender Einzelheiten, um Widersprüche und Ungereimtheiten aufzudecken.

Wenig erfolgversprechend ist dagegen die Kontrolle ganzer Straßenblöcke und die Untersuchung von Eigentumsverhältnissen einer Vielzahl von Personen.

15.2.2 Deduktionsmethode

Die deduktive Fehlerlokalisierung umfaßt folgende Schritte (s. Abb. 15.1 rechts).

Schritt 1: Relevante (Test-)Daten zusammenstellen

Schritt 2: Mögliche Ursachen („Hypothesen") für die fehlerhaften Testdaten aufzählen

Schritt 3: Ursachen (Hypothesen) eliminieren
Bleibt keine Hypothese übrig, sind neue (Test-)Daten zu sammeln. Bleiben mehrere Hypothesen übrig, so ist mit der „wahrscheinlichsten" Hypothese zu beginnen oder zusätzliche Testdaten sind zu untersuchen.

Schritt 4: Hypothese verfeinern

Schritt 5: Verbleibende Hypothese beweisen bzw. bestätigen
Es ist zu zeigen, daß die falschen und richtigen Testergebnisse sich mit Hilfe der Hypothese genau erklären lassen.

[3] Bei einer ungeraden Anzahl gibt es „den mittleren" Studierenden.

Bei der Lösung eines Mordfalls bedeutet deduktives Vorgehen zum Beispiel:

1. Eine Menge von Verdächtigen bestimmen.
   (Hypothese: Einer von ihnen muß der Täter sein.)
2. (a) Verdächtige ausscheiden. (Der Gärtner hat ein Alibi.)
   (b) Die Verdachtsmomente verfeinern. (Der Täter muß schwarze Haare haben.)
3. Schluß ziehen. (Der Butler war es.)

Bei der Induktionsmethode wurde bei Schritt 1(a) angemerkt, daß ähnliche, aber unterschiedliche Testdaten günstig für die Fehlerlokalisierung sind. Entsprechendes gilt für die Deduktionsmethode. Der Zweck solcher Testdaten ist es, möglichst nur eine einzige, aber jedenfalls nur wenige (Ein- oder Ausgabe-)Bedingungen zu testen. Dies steht im Gegensatz zu den Testdaten zum *Aufdecken* von Fehlern, die möglichst viele Bedingungen gleichzeitig abdecken sollen (vgl. Abschnitt 4.2.1, Äquivalenzklassen-Methode). Das Vorgehen besteht darin, die Testdaten zu variieren, die zum Aufdecken eines Fehler(symptom)s führten. Bei der Induktionsmethode dient dies zum Aufstellen oder Bestätigen von Hypothesen. Bei der Deduktionsmethode können damit vermutete Ursachen ausgeschlossen und verbleibende Hypothesen verfeinert oder bestätigt werden.

15.2.3 Fehlerlokalisierung durch Bestimmung der potentiellen Wege mit Fehlern

Falls mit der Induktions- oder Deduktionsmethode die fehlerhafte(n) Anweisung(en) nicht genau zu lokalisieren sind, empfiehlt sich folgendes Vorgehen, welches sich an dem Kontrollfluß des betrachteten Programms orientiert. Es operiert also auf Ebene (a) (Programmstruktur) und ist eine Filterungsstrategie.

Schritt 1:

Ermittle zu den Testdaten $t_1, \ldots, t_n$, bei denen ein Fehlverhalten auftritt, die ausgeführten Wege $w_1, \ldots, w_n$ und jeweils die Menge A_i der Anweisungen, die auf dem Weg w_i liegen, $i = 1, \ldots, n$.

Schritt 2:

(a) Falls genau eine fehlerhafte Anweisung a_f die Ursache für alle fehlerhaften Testergebnisse ist, liegt sie im Durchschnitt aller $A_i : a_f \in \bigcap_{i=1\ldots n} A_i$.

(b) Falls mehrere fehlerhafte Anweisungen die Ursache sind, liegen sie in der Vereinigung aller A_i.

Da man i. allg. die Fehleranzahl *nicht* kennt, muß man — falls man mit Schritt 2(a) keine Fehlerquelle findet — letztlich nach Schritt 2(b) vorgehen.

Das folgende Beispiel demonstriert das Vorgehen gemäß den Schritten 1 und 2.

BEISPIEL 15.2.3
Für das Programm aus Abbildung 7.2, bei dem Anweisung 9 (LOW := MID + 1) durch LOW := MID ersetzt wird, ergibt sich folgendes für $A = (2, 5, 7, 9, 12)$ *und* $N = 5$*, wenn* F *bei den Testdaten* t_1 *bis* t_5 *die angegebenen Werte hat:*

t_1*:* $F = 7$ *ergibt NPOS = 3 (o. k.).*
t_2*:* $F = 5$ *führt zu einer „Endlosschleife".*
t_3*:* $F = 2$ *bewirkt NPOS = 1 (o. k.).*

Aus diesen Angaben läßt sich schlecht eine Hypothese über die Fehlerursache(n) ableiten. Wenn aber noch Angaben über die durchlaufenen Wege vorliegen, ist dies eher möglich.

t_1*: Weg: 0, 1, 2, 3, 5, 6, 7, 11.*
t_2*: Weg: 0, 1, 2, 3, 5, 8, 10,* $(2, 3, 5, 8, 9)^*$ *(endlos)*
t_3*: Weg: 0, 1, 2, 3, 5, 8, 10, 2, 3, 5, 6, 7, 11.*

Da nur bei t_2 *ein Fehler auftritt, muß auf dem entsprechenden Weg eine fehlerhafte Anweisung liegen. Bei einer* **optimistischen Strategie** *könnte man noch die Anweisungen* ausschließen, *die auf den Wegen liegen, die bei den fehlerfreien Testdaten* t_1 *und* t_3 *ausgeführt werden. Damit bleibt nur die Anweisung 9 bei* t_2 *übrig, die in der Tat den Fehler enthält.*
Diese optimistische Strategie ist aber i. allg. falsch, wie die Betrachtung der Testdaten t_4 *und* t_5 *und der zugehörigen durchlaufenen Wege zeigt:*

t_4*:* $F = 9$ *ergibt NPOS = 4 (o. k.).*
t_5*:* $F = 12$ *führt zu einer „Endlosschleife".*
Weg zu t_4*: 0, 1, 2, 3, 5, 8, 9, 2, 3, 5, 6, 7, 11.*
Weg zu t_5*: 0, 1,* $(2, 3, 5, 8, 9)^*$ *(endlos).*

Die Wege, die zu den Testdaten t_1, t_3, t_4 *(mit korrektem Ergebnis) gehören, enthalten die Anweisungen 0, 1, 2, 3, 5, 6, 7, 8, 9, 10, 11, also alle außer 4. Die Wege, die zu den Testdaten* t_2 *und* t_5 *(mit fehlerhaftem Ergebnis) gehören, enthalten nur die Anweisungen 0, 1, 2, 3, 5, 8, 9, 10; d. h. eine Teilmenge der Anweisungen, die in den fehlerfreien Fällen vorkommen. Gemäß der optimistischen Strategie dürfte also keine Anweisung einen Fehler enthalten.*
Der Fehlschluß wird dadurch hervorgerufen, daß bei der optimistischen Strategie angenommen wird, daß ein Testdatum mit korrektem Ergebnis keine fehlerhaften Anweisungen ausgeführt haben kann. Das ist aber nicht der Fall, wie Testdatum t_4 *zeigt. Es führt die fehlerhafte Anweisung 9 aus, ohne einen Ausgabefehler zu erzeugen. Das liegt daran, daß der fehlerhafte Wert von LOW in Anweisung 9 (beispielsweise 3 statt 4) bei der Berechnung von MID in Anweisung 2 trotzdem ein*

richtiges Ergebnis MID $= \lfloor \frac{3+5}{2} \rfloor = \lfloor \frac{4+5}{2} \rfloor = 4$ *berechnet. Der Fehler wird nämlich beim Abrunden mit* $\lfloor\ \rfloor$ *toleriert.*

Man darf also (siehe Schritt 1 und 2 bei dem Vorgehen oben) nur die Wege der Testdaten mit *fehlerhaftem* Ergebnis betrachten.

Die Schritte 1 und 2(b) des obigen Vorgehens bestimmen eine Menge von *vollständigen* Wegen und zugehörigen Anweisungen als potentielle Fehlerquelle. Führt ein Test einen Weg *endlicher* Länge aus und erzeugt er einen fehlerhaften Wert einer Ausgabevariablen, können gewisse Anweisungen als Fehlerquelle ausgeschlossen werden, wenn der Datenfluß genauer betrachtet wird. Durch Rückwärtsverfolgung des Datenflusses[4] kann die minimale Menge von Anweisungen auf dem Weg ermittelt werden, welche für den gegebenen Test dasselbe Berechnungsergebnis für die (fehlerhafte) Ausgabevariable erzeugt. Diese Teilmenge wird (engl.) **slice** genannt (deutsch/wörtlich: „Scheibe, Schnitte“ trifft den Bedeutungsgehalt nicht genau).

Beispiel 15.2.4
Man betrachte folgendes Programm:

```
read(i);
c := i;
read(d);
if d = 37
then begin a := c;
           b := 2;
     end;
else a := 2;
```

Um den Fehler $c := i$ *(statt* $c := i + 1$*) zu finden, der (in einigen Fällen) als Ergebnis* $a = i$ *statt* $a = i + 1$ *liefert, ist mit* $d = 37$ *zu testen.*
Dann ist folgende „slice“ relevant:

```
read(i);
c := i;
read(d);
if d = 37
then a := c;
else ... ;
```

[4] vgl. Kapitel 8.1, Seite 212

15.2.4 Fehlerlokalisierung durch Soll/Ist-Vergleich der Berechnung auf einem Weg

Mit obigen Einschränkungen ist für einen Weg w_i immer noch nicht klar, *wo* sich auf dem Weg die fehlerhafte Anweisung befindet. Daher ist das genaue Verfolgen der Berechnung auf einem Weg notwendig.
Hierbei gibt es zwei Möglichkeiten:

A. Vorwärtsverfolgung des Weges,

B. Rückwärtsverfolgung des Weges.

Bei Tests, bei denen eine Schleife unendlich oft durchlaufen wird, kann natürlich nur die Vorwärtsverfolgung gewählt werden. Rückwärtsverfolgung empfiehlt sich, falls der Fehler in der hinteren Hälfte des Weges vermutet wird. Folgende Schritte 1. bis 3. sind bei beiden Methoden A und B jeweils durchzuführen.

A. Lokalisierung eines Fehlers durch Vorwärtsverfolgung:

1. Beginne mit den Werten der Eingabevariablen.
2. Berechne schrittweise die Werte der (Programm-)Variablen nach jeder Anweisung auf dem Weg (das ist der Programmzustand).
3. Vergleiche den tatsächlichen Programmzustand jeweils mit dem „richtigen“ Programmzustand (wie er sein sollte).
   Bei der ersten *Abweichung* ist eine fehlerhafte Anweisung gefunden (spätestens bei der Ausgabe am Ende des Weges).

B. Lokalisierung eines Fehlers durch Rückwärtsverfolgung:

1. Beginne mit den (fehlerhaften) Werten der Ausgabevariablen.
2. Berechne schrittweise, wie die Werte der Programmvariablen vor jeder Anweisung auf dem Weg gewesen sind (das ist der Programmzustand).

   Beispiel 15.2.5
   Anweisung $x := x + 5$
   Wenn hinterher $x = 8$ *gilt, muß vorher* $x = 3$ *gelten.*

3. Vergleiche jeweils den tatsächlichen Programmzustand mit dem Soll-Programmzustand. Bei der ersten *Übereinstimmung* liegt eine fehlerhafte Anweisung vor.

Beide Methoden operieren also auf Wissensebene (b) (Programmverhalten) und versuchen eine Verifikationsstrategie. Dabei muß als letzte Aufgabe nur noch eine Antwort auf folgende Frage gefunden werden: Wie vergleicht man den tatsächlichen mit dem Soll-Programmzustand? Dabei ist das Hauptproblem die Ermittlung bzw. Spezifikation des Soll-Programmzustands.

Das Vergleichen des erwarteten (Soll-)Programmzustands mit dem tatsächlichen Programmzustand läßt sich durch den Einbau von **ausführbaren Zusicherungen** (executable assertions) in das Programm teilweise automatisieren. Die Zusicherungen werden während der Programmausführung abgeprüft und führen bei Abweichungen zu Fehlermeldungen bzw. zum Programmabbruch.
Die Zusicherungen haben z. B. die Form

ASSERT <Boolescher Ausdruck> Anweisung.

Kompliziertere Zusicherungen können z. B. folgende Form haben:

ASSERT A[I] $\neq$ A[J] FOR ALL (I, J) (1:8) WHERE (I $\neq$ J)
LIMIT 2 VIOLATIONS HALT

Dies bedeutet, daß das Programm beendet wird (HALT), wenn zum zweiten Mal (LIMIT 2) nicht alle ersten acht Elemente des Arrays A bei der Ausführung der ASSERT-Anweisung verschieden sind.

Folgendes Vorgehen ist beim Verwenden von ausführbaren Zusicherungen angebracht:

1. Nach jeder Initialisierung von Variablen, Arrays und sonstigen Datenstrukturen ist eine Zusicherung einzufügen, die den (spezifizierten) Wertebereich der Daten abprüft.

2. Zusicherungen sind an Anfang und Ende von internen Prozeduren einzubauen.

3. Zusicherungen sind vor und nach wichtigen Schleifen oder Prozeduraufrufen einzufügen.

4. Vor möglichen Singularitäten[5] sollten geeignete Zusicherungen plaziert werden, z. B. „ASSERT z $\neq$ 0“ vor einer Division durch z.

5. In wichtige Schleifen sind *induktive* Zusicherungen einzubauen (vgl. Kapitel 12.4).

Zusicherungen können **lebend** sein, d. h. sie werden tatsächlich ausgeführt, oder sie können als *Kommentar* behandelt werden.
Bei Produktionsläufen wird das Umwandeln von lebenden Zusicherungen in Kommentare aus Effizienzgründen oft empfohlen. Aber dies ist wie das Verhalten eines Matrosen, der seine Schwimmweste zu Trainingszwecken an Land anzieht, aber sie

[5] **Singularitäten** sind einzelne Werte, für die mathematische Funktionen nicht definiert sind.

auszieht, wenn er in See sticht. Die Zusicherungen sollten also lebend bleiben, allerdings nur bei sicherheitsrelevanten Anwendungen zum Programmabbruch (mit Rückfall in einen sicheren Zustand) führen. Die Verletzungen der Zusicherungen sollten (in einer Datei) aufgezeichnet werden, um dem Wartungsdienst als Diagnosehilfe zur Verfügung zu stehen.

15.2.5 Druckanweisungen einstreuen

Bei diesem Vorgehen wird die Dynamik des Programmablaufs durch die Folge der ausgeführten Druckanweisungen protokolliert. Diese Information wird insbesondere dann als Verhaltensbeschreibung [Ebene (b)] benötigt, wenn keine normalen Programmausgaben vorliegen oder das Programm in eine Endlosschleife gerät.

Bei einem unbedachten Einsatz dieser Methode treten folgende Probleme auf:

1. Eventuell werden beträchtliche Datenmengen und Ausgaben erzeugt, insbesondere bei Druckanweisungen in Schleifen.

2. Das Einfügen der Druckanweisungen erzwingt oder verleitet zu Programmänderungen. Dadurch können

   - Fehler verdeckt (maskiert) werden,
   - Ausführungszeiten in Realzeitprogrammen — und damit das Programmverhalten — verändert werden,
   - neue Fehler eingebaut werden.

3. Bei großen Programmen ist dieses Vorgehen ökonomisch nicht zu vertreten (zu hohe Kosten), bei Realzeit- oder Betriebssystem-Programmen können damit nicht alle Fehler eingegrenzt werden.

Damit Problem 1 nicht auftritt und die Vorteile voll zur Entfaltung kommen, sind folgende Anregungen zu beachten:

1. Jede Druckanweisung erzeugt einen **Schnappschuß** des Programmzustands an der entsprechenden Stelle, d. h. folgendes:

   (a) Identifikation des Programmteils,

   (b) Teile des Programmzustands [falls (a) nicht ausreicht].

   Dabei sollte das Drucken an- und abgeschaltet werden können (wegen Problem 1) und die Informationsmenge und -darstellung je nach Bedarf flexibel gewählt werden können.

2. Die Ausgabemenge ist zu minimieren, und zwar mit folgenden Strategien (z. T. alternativ):

   (a) „minimale“ Testdaten verwenden, z. B. Eingabedaten, die nur zu einem höchstens dreimaligen Durchlaufen einer Schleife führen. (Es ist unwahrscheinlich, daß ein Fehler erst bei einem k-maligen Durchlauf mit $k > 3$ gefunden wird, obwohl es natürlich solche Fälle gibt, die extra zu testen sind.) Wenn klar ist, daß die Schleife den Fehler nicht enthält, sollte die Ausgabe dafür völlig abgeschaltet werden bzw. beim Lesen des Protokolls ignoriert werden.

   (b) Als Alternative zu (a) kann die Ausgabe z. B. nach dreimaligem Schleifendurchlauf abgeschaltet werden.

   (c) Für Untermodule eines Moduls bzw. Unteroperationen einer Operation sollten die Schnappschüsse (der Trace) je nach Eingrenzung des Fehlers einzeln an- bzw. abgeschaltet werden.

   BEISPIEL 15.2.6
   Im Modul A werden in einer Schleife jeweils vier Operationen aufgerufen. Falls kein Integrationstest möglich bzw. sinnvoll ist, werden zuerst alle fünf „Schalter“ (für jede Operation und für das Modul je einer) eingeschaltet („trace_on“). Wenn eine Operation offenbar fehlerfrei ist, wird ihr Schalter „abgeschaltet“ („trace_off“). Beim Systemtest z. B. sind schließlich alle Operationsschalter abgeschaltet („trace_off“).

   (d) Werden zu viele Schnappschüsse bei einem Programmablauf erzeugt, sind jeweils nur die letzten k Schnappschüsse zu speichern (mit einstellbarem Wert von k). Alternativ kann man — bei einer sehr großen Anzahl auszuführender Schnappschüsse — diese hierarchisch ordnen und bei der Ausgabe nur Schnappschüsse bis zu einer bestimmten Hierarchie-Ebene berücksichtigen.

3. Inhalt und Plazierung des Schnappschusses
   Für die Beantwortung der Frage, was im einzelnen bei einem Schnappschuß ausgegeben werden soll, und wo Schnappschüsse erforderlich bzw. sinnvoll sind, bieten sich folgende Regeln an:

   (a) Protokollieren des Inhalts von Ein- und Ausgabe-Sätzen.
   Druckausgabe wird natürlich immer schon protokolliert, aber alles andere sollte besonders protokolliert werden. Bei komplizierten Druckausgaben sollten alle Eingaben zur Druckprozedur protokolliert werden und Schnappschüsse in die Druckprozedur selbst eingebaut werden.

   (b) Vor jeder wichtigen Programmverzweigung (z. B. bei Schleifen, insbesondere beim *exit*) sollten die in der entsprechenden Abfrage benutzten Variablen protokolliert werden, damit die Programmlogik kontrolliert werden kann.

(c) Der Verlauf von Berechnungen sollte bei Bedarf protokolliert werden, und zwar
- die Eingaben (bzw. Anfangswerte, falls vom Programm erzeugt),
- die Ergebnisse (Ausgaben),
- bei längeren Berechnungen die Zwischenergebnisse.

15.2.6 Werkzeuge benutzen

Der Aufwand für die Fehlerbehebung (engl.: debugging) hängt natürlich wesentlich von der diesbezüglichen Qualität der Werkzeuge, insbesondere des benutzten Compilers ab. Beim Testen sollte daher immer — falls verfügbar — eine entsprechende „Debugging"-Version des Compilers benutzt werden, die alle (Syntax-)Fehler im Modul-System erkennt und die Fehlerlokalisierung erleichtert.

Durch Werkzeugeinsatz kann folgendes Vorgehen unterstützt werden, das sich an Struktur und Verhalten des Programms [vgl. S. 418, Ebenen (a) und (b)] orientiert:

1. Setzen von Wiederanlaufpunkten („breakpoints")
   An Wiederanlaufpunkten wird der Kontrollfluß angehalten, und der Benutzer kann (im Dialog) den Zustand des Programms abfragen. Damit können also ganz *flexibel* Schnappschüsse angezeigt werden.

2. Die Änderung bestimmter Variablen und vorher bestimmte Befehle und Unterprogrammaufrufe werden stets (automatisch) protokolliert — wie beim Einstreuen von Druckanweisungen — aber man muß die Druckanweisungen nicht selbst einbauen.

Der Compiler bietet meist entsprechende Hilfen an, oder ein besonderer Vorübersetzer erledigt das.

Nachteilig ist bei diesem Vorgehen, daß eventuell wieder ungeheure Daten- bzw. Ausgabe-Mengen erzeugt werden. Der Werkzeugeinsatz ist allerdings effizient und nutzbringend, wenn er es erlaubt, Schnappschüsse nach den Regeln von Abschnitt 15.2.5 zu erzeugen. Wissensbasierte Werkzeuge, die auf den Ebenen (c) und (d) (Funktion und Verhaltensklassifikation) operieren, wären sehr nützlich. Sie sind aber noch im Forschungsstadium oder nur auf bestimmte, enge Problemfelder anwendbar.

15.2.7 Fehlerhafte Situationen

Die bisher genannten Fehlerlokalisierungsverfahren beziehen sich vor allem auf einen normalen Ablauf des Programms, bei dem nur falsche Ausgaben produziert werden bzw. falsche Wege im Kontrollflußgraphen ausgeführt werden. Es gibt aber ein ganzes Spektrum von fehlerhaften Situationen, das zu verschiedenen Arten der Fehlerlokalisierung Anlaß gibt:

1. Übersetzung durch Compiler nicht beendet, keine Fehlermeldung:

   Durch Änderung des Programms kann dieses Problem folgendermaßen gelöst werden (ohne Korrektur des Compilers, der fehlerhafterweise keinen Fehler meldet):

   (a) erfahrenen Kollegen um Hilfe bitten,

   (b) falls keiner verfügbar (als letzter Ausweg):

      i. Programm solange zerlegen (in „Abschnitte"), bis ein Abschnitt kompilierbar ist.

      ii. Die anderen Abschnitte schrittweise „integrieren" bis diese Teilmenge nicht mehr kompilierbar ist. Dann enthält der letzte Abschnitt einen Fehler.

      iii. Falls der Fehler beim Inspizieren dieses Abschnitts nicht gefunden wird, ist der Abschnitt wiederum zu zerlegen (rekursive Fortsetzung bei i, ii, ...) usw.

2. Ausführung des Programms, keine Programmausgabe, evtl. aber Ausgabe von Meldungen des Systems.

   In diesem Fall empfiehlt sich folgendes Vorgehen:

   (a) Bei Fehlern in der Programmlogik (zum Beispiel „*goto* Ende" vor jeder Druckanweisung) ist wie bei der normalen Lokalisierung vorzugehen (siehe vorherige Abschnitte).

   (b) Bei Hardware-, Betriebssystem- oder Programmfehlern (z. B. Division durch 0, Array-Index außerhalb des zulässigen Bereichs), die zu einem Systemfehler führen und einen Systemfehler-Code ausgeben:

      i. Bei Hardware- oder Betriebssystem-Fehlern sind „alternative" Programme zu schreiben, d. h. es sind „Mutanten" des Programms zu bilden (vgl. Kapitel 9.3), so daß die Fehlerursachen umgangen werden bis der Fehler verschwindet oder gefunden wird. (Dies ist leider eine sehr aufwendige und problematische Technik.)

      ii. Bei Programmfehlern sind die normalen Techniken zur Fehlerbehandlung und -lokalisierung zu verwenden (siehe vorherige Abschnitte).

3. Vorzeitige Beendigung des Programmablaufs (abnormal end):

   In diesem Fall ist das Vorgehen aus den vorherigen Abschnitten angebracht. Bei geeigneter Werkzeugunterstützung ist ein Speicherauszug[6] sinnvoll, der vom „Debugger" zur genauen Lokalisierung der Fehlerstelle verwendet werden kann.

[6] Früher (d. h. bei Assembler-Programmen etc.) wurde dazu oft ein kompletter Speicherabzug („Postmortem"-Dump) erstellt und ausgewertet. Diese Technik ist aber selten zu empfehlen, da sie immense Datenmengen und nur ein statisches Abbild der fehlerhaften Situation liefert und der Fehler evtl. schon durch weiter ausgeführte Programmanweisungen überdeckt wird.

4. Unendliche Schleife (sozusagen das Gegenteil von 3):

   In diesem Fall empfiehlt es sich, Druckanweisungen vor und hinter jede Schleife (nicht *in* die Schleife) einzufügen. Die letzte gedruckte Ausgabe „Anfang von Schleife x“ kennzeichnet die unendliche Schleife (siehe Abschnitt 15.2.5).

15.2.8 Voraussetzungen und Prinzipien der Fehlerlokalisierung

Für eine erfolgreiche Fehlerlokalisierung müssen folgende Voraussetzungen gelten und folgende Maßnahmen und Prinzipien sind zu erfüllen:

1. Voraussetzung: konstruktive Maßnahmen und Prinzipien bei der Entwicklung:

   (a) Das Programm ist zu modularisieren, d. h. in relativ kleine Module zu zerlegen.

   (b) Die Codierung sollte den Test unterstützen.
   Fehler sind z. B. in der Hälfte der Zeit zu finden, wenn akzeptierte, verständliche Variablennamen (und nicht V1, V2, ...) verwendet werden.

2. Prinzip der Verwendung von Testhilfen:

   - Eine Liste der Variablen und Konstanten im Modul ist (zum Erkennen von Tippfehlern) zu benutzen.
   - Es sind Diagnosehilfen (Zusicherungen, Druckanweisungen o. ä., siehe Abschnitte 15.2.4 und 15.2.5) in das Programm einzubauen und zwar immer (nicht erst im Fehlerfall).
   - Die eingefügten Diagnose-Anweisungen sollten leicht unterscheidbar sein von anderen Anweisungen, z. B. durch besondere Anfangszeichen, so daß sie von einem Präprozessor in einen Kommentar umgewandelt werden können[7].
   - Die Diagnosehilfen sind frühestens im Produktionslauf zu entfernen[8].

3. Prinzipien der allgemeinen Vorgehensweise:

   - Probleme sind zu isolieren und eins nach dem anderen (nicht mehrere gleichzeitig) zu lösen.
   - Ein guter Diagnostiker sollte in der Lage sein, Fehler zu lokalisieren, *ohne* den Computer zu verwenden, d. h. Werkzeuge sollten nur als zusätzliche Hilfsmittel verwendet werden.

[7] Früher verwendete man dafür verschiedene Farben von Lochkarten.

[8] In kritischen Programmteilen sollten Diagnosehilfen o. ä. als Fehlermeldungen verbleiben (vgl. Abschnitt 15.2.4).

- Wenn man in eine gedankliche Sackgasse gerät, ist folgendes nützlich:
  - Nach einer ca. 30 Minuten dauernden erfolglosen Bearbeitung eines kleinen Programms sollte das Problem „überschlafen“ werden.
    (Dann kann evtl. eine Lösung im Unterbewußtsein oder durch „frisches“ Bewußtsein gefunden werden.)
  - Das Problem sollte einem Kollegen geschildert werden.
    (Dabei kommt man oft auf neue Gedanken und Ideen.)

4. Verwaltungs- und Management-Prinzipien:

   - Die Ausgaben von verschiedenen Testläufen sind nach Datum, Uhrzeit und Version zu ordnen.
   - Unnötige Testausgaben sind wegzuwerfen bzw. in der Testdatenbank zu löschen. Alte Versionen sind aber zu sichern, um eventuell falsche, sogenannte „Korrekturen“ rückgängig machen zu können.

15.3 Prinzipien der Fehlerkorrektur und Fehleranalyse

Folgende Fehlerkorrekturprinzipien sind zu beachten:

1. In der Umgebung eines Fehlers sind noch weitere Fehler zu vermuten, da Fehler gehäuft auftreten.

2. Die Ursache des Fehlverhaltens (der Fehler) — nicht nur ein Symptom davon — ist zu beheben[9].

3. Die Fehler-„Korrekturen“ sind strenger als das Originalprogramm zu testen, da die Wahrscheinlichkeit, daß die „Korrektur“ richtig ist, deutlich weniger als 100% beträgt.

4. Die Fehler-„Korrekturen“ sind bei großen Programmen besonders streng zu testen, da die Wahrscheinlichkeit, daß die Korrektur richtig ist, mit zunehmender Programmgröße sinkt.

5. Nach Fehler-„Korrekturen“ müssen Regressionstests durchgeführt werden, da eine „Korrektur“ nicht nur unvollständig sein kann, sondern auch neue Fehler als Nebeneffekt in anderen Programmteilen erzeugen kann. Daher muß nicht nur die Fehlersituation nach der Korrektur getestet werden, sondern das ganze Umfeld.

[9] Dieses Prinzip wird in anderen Wissenschaften leider auch häufig verletzt, z. B. in der Medizin bei psychosomatischen Leiden oder ernährungsbedingten Krankheiten.

6. Eine Fehlerkorrektur ist wie ein Programmentwurf zu behandeln, d. h. alle Vorgehensweisen, Methoden und Formalismen aus der Entwurfsphase sind auch bei der Fehlerbehebung anzuwenden (z. B. eine Spezifikation der Änderung, eine Inspektion und ein Test). Dies ist notwendig, weil Fehlerkorrektur eine Form des Programmentwurfs ist.

7. Das Programm darf nur in der Quellsprache geändert werden, weil sonst Quellcode und Objektcode nicht mehr zueinander passen, d. h. bei einer erneuten Compilierung kann der Fehler wieder auftreten.

Es ist unökonomisch, gewisse Fehler immer wieder zu machen und erst nach dem Testen mit viel Aufwand zu lokalisieren und zu korrigieren. Daher sind die eigentlichen Fehlerursachen zu ermitteln und zu beseitigen bzw. es sind Mittel anzugeben, mit denen solche Fehler wenigstens in früheren Entwicklungs- oder Testphasen (z. B. bei der Inspektion) entdeckt werden können. Dies entspricht der Idee, den Entwicklungsprozeß durch ein totales Qualitätsmanagement (total quality management TQM) zu verbessern und damit einen höheren Reifegrad des Prozesses zu erreichen, was mit dem modernen Schlagwort **Reifegradmodell** (capability maturity model, **CMM**) bezeichnet wird (vgl. Kapitel 2.6 und 3.7 zur Zertifizierung und Verbesserung des Reifegrades einer Entwicklungsinstitution).
Für die Prozeßverbesserung durch Fehleranalyse ist folgendes zu ermitteln:

1. In welcher Entwicklungsphase wurde der eigentliche Fehler gemacht? (Eine nicht eindeutige Spezifikation, die später zu einem Entwurfs- oder Codierfehler führt, ist z. B. der eigentliche Fehler.)

2. Durch welche mangelnden Kenntnisse, Fähigkeiten oder organisatorischen Regelungen wurde der Fehler verursacht bzw. durch welche Verbesserungen wäre er zu vermeiden? (Das zeigt Organisations- und Qualifikationsschwächen auf, die durch Umstellungs- und Schulungsmaßnahmen — nicht durch Maßregelungen — zu beheben sind.)

3. Warum wurde der Fehler nicht eher entdeckt bzw. wie hätte er eher entdeckt werden können? (Damit werden Hinweise auf Schwächen und Verbesserungsmöglichkeiten der analytischen Qualitätsmanagementmaßnahmen gegeben[10]).

4. Wie wurde der Fehler entdeckt?
   Die Antwort auf diese Frage bezeichnet erfolgreiche Techniken der statischen und dynamischen Analyse, die unter ähnlichen Bedingungen wiederholt eingesetzt werden sollten. Dies ist also — im Unterschied zu den Punkten 1 bis 3 — eine *positive* Rückmeldung.

[10] vgl. Kapitel 12.1, Inspektion, Schritt 6, Seite 307

15.4 Übungen

Übung 15.1[11]**:**
Das Programm zum binären Suchen (siehe Abbildung 7.2) enthalte Fehler in den Anweisungen 9 und 10, d. h. Anweisung 9 sei LOW := MID und Anweisung 10 sei HIGH := MID.
Überlegen Sie sich die Auswirkungen auf die Testdaten t_1 bis t_5 von Beispiel 15.2.3 (ausgeführte Wege und Ergebnisse für NPOS). Welche Schlüsse könne aus den ausgeführten Wegen gezogen werden, d. h. welche Anweisungen können fehlerhaft sein? Betrachten Sie dazu auch die Testdaten t_6 mit $F = 3$ und t_7 mit $F = 8$.

Übung 15.2:
Ein modifizierter Textformatierer (vgl. Beispiel 7.3.1 auf S. 208) enthalte den Fehler, daß die Reihenfolge der Anweisungen *bufpos := bufpos + 1* und *buffer[bufpos] := c* in Zeile 24 vertauscht ist. Dieser Fehler wird schon beim Test mit einem Eingabetext, der aus einem Buchstaben (z. B. „F“) und *EOF* besteht (und $MAXPOS > 1$), aufgedeckt, da dann die Anweisung *outchar(buffer[k])* in Zeile 16 ein undefiniertes Zeichen (anstelle von „F“) ausgibt.

(a) Verfolgen Sie die Datenfluß- und Kontrollflußkette von der fehlerhaften Ausgabe in Zeile 16 rückwärts, ermitteln Sie die Anweisungen, die dazu einen Beitrag leisten, und geben Sie ein entsprechend reduziertes Programm (die „*slice*“) an.

(b) Geben Sie alternativ dazu ein Programm an, bei dem die verschiedenen Schleifendurchläufe der *slice* „abgewickelt“ sind, d. h. sequentiell nacheinander aufgeschrieben werden.

(c) Geben Sie eine Zusicherung an, die den korrekten Zustand nach der fehlerhaften Programmstelle beschreibt und damit die Abweichung aufdeckt.

[11] Aus Platzgründen werden hier keine umfangreichen Übungsbeispiele für die Induktions- und Deduktionsmethode angegeben (und kleine Beispiele sind zu trivial). Daher wird hier nur auf Beispiele von Myers verwiesen, die sich auch als Übung eignen: Ein Fehler in einem DISPLAY-Kommando, der mit Deduktion zu finden ist (s. [Mye 79], Kap. 7) und ein Fehler in einem Systemprogramm (loader), der mit Induktion zu finden ist (s. [Mye 76], S. 248 ff.).

15.5 Verwendete Quellen und weiterführende Literatur

Die Definition des Begriffs **Fehlerbehebung** und die Einschätzung des Aufwands für die Fehlerlokalisierung und -korrektur stammt von Myers (s. [Mye 79], Kapitel 7). Experimente zur Fehlerbehebung, die den geringen Aufwand für die Fehlerkorrektur bestätigen, führte Gould schon 1975 durch (s. [Gou 75]). Die Schritte einer Fehlerbehebungsmethode sind in [BrS 73], S. 68 ff.; [Mye 76], Kap. 13 und [DuE 88], S. 163 aufgeführt. Die psychologisch bedingte Abneigung bei Programmierern gegenüber der Fehlerbehebung ist in [Mye 79] erwähnt. Die Klassifikation der **Wissensarten**, **-ebenen** und **Fehlerbehebungsstrategien** findet sich in [DuE 88], S. 162; [Duc 93] und [Mil 87], S. 334.

Die **Methoden der Fehlerlokalisierung** werden von Myers in Denkmethoden und Holzhammermethoden eingeteilt. Zu letzteren zählt er die Methoden „Speicherabzug“ (DUMP), „Einstreuen von Druckanweisungen“ und „Werkzeuge benutzen“ (siehe [Mye 79]). Die in [Mye 79] vorgestellte **Induktionsmethode** basiert auf einer Methode von [BrS 73], genannt „The Method“. Diese Methode ist wiederum eine Anpassung einer Problemlösungsstrategie für Manager (aus dem Jahre 1965) an das Problem der Fehlerbehebung in Programmen. Die Vergleiche mit dem Vorgehen beim Lösen eines Mordfalls und das Beispiel 15.2.2 für die Induktionsmethode wurden von [Mye 79], Kap. 7, übernommen. Die **Fehlerlokalisierung durch Rückwärtsverfolgung** wird von Myers für kleine Programme empfohlen, die Vorwärtsverfolgung bzw. -propagierung von Milne, allerdings bei einer funktionsbezogenen Strategie (s. [Mye 79], Kap. 7; [Mil 87], S. 336). Die Verwendung von komplizierten **Zusicherungen** und ein entsprechendes Vorgehen hat Stucki vorgeschlagen (s. [Stu 77]). Der Vergleich mit dem Umgang eines Matrosen mit seiner Schwimmweste stammt von Knuth (zitiert nach [Mye 76], S. 295 u.). Vorschläge zur Verwendung von speziellen Zusicherungen zur Aufdeckung bestimmter Fehlerarten macht Rosenblum (s. [Ros 92]). Bei **Druckanweisungen** wurden die Anregungen von [BrS 73], S. 41 ff., aufgegriffen. Die Aufstellung der fehlerhaften Situationen, die zu verschiedenen Arten der Fehlerlokalisierung Anlaß geben, stammt aus [Tas 74]. **Prinzipien der Fehlerlokalisierung** finden sich in [BrS 73], [Gou 75], [Mye 79] und [Tas 74]. Gould hat durch Experimente nachgewiesen, daß Fehler in der Hälfte der Zeit zu finden sind, wenn akzeptierte, verständliche Variablennamen verwendet werden (s. [Gou 75]). Die **Prinzipien der Fehlerkorrektur und Fehleranalyse** stammen aus [Mye 79], Kap. 7.

Weitergehendes zum Problem der **Mehrfachfehler** und Ansätze mit vorverarbeitetem **(Lokalisierungs-)Wissen** findet man in [Mil 87], S. 335 f. Einen Überblick über **neuere Ansätze** zur Fehlerlokalisierung, insbesondere von logischen und funktionalen Programmen und nebenläufigen und verteilten Systemen gibt der von Fritzen herausgegebene Tagungsband, insbesondere der Überblicksartikel von Ducassé (s. [Fri 93], [Duc 93]).

16 Management des Testens und Prüfens

Ausgehend von den grundlegenden Problemstellungen und Lösungsansätzen des Qualitätsmanagements, Prüfens und Testens (Teil I) wurde in den Teilen II, III und IV dieses Buches eine Fülle von Test- und Prüfmethoden (also auch Analyse- und Verifikationsmethoden) mit ihren Stärken und Schwächen, Anwendungsbereichen und Einschränkungen vorgestellt, verglichen und bewertet. Für Software-Entwicklerinnen und -Entwickler drängen sich daher vermutlich die folgenden Fragen auf:

1. Können Kennzahlen für Softwareprogramme angegeben werden, von denen der Test- und Überprüfungsaufwand, aber auch die Art der anzuwendenden Test- und Prüfungsmethoden abgeleitet werden kann? (Entsprechende Komplexitätsmaße werden in Kapitel 16.1 und 16.2 vorgestellt.)

2. Wenn mehr als eine Softwaretest- oder -prüfmethode anzuwenden ist, wie sind die Methoden zu kombinieren? (Vorschläge dazu werden in Kapitel 16.3 gemacht.)

3. Wie kann die Anzahl der Restfehler im Programm bzw. die Zuverlässigkeit abgeschätzt werden? (Genaueres dazu wird in Kapitel 16.4 beschrieben.)

4. Welche speziellen Managementfragen sind bei der Durchführung des Testens und Prüfens zu beachten? (Empfehlungen dazu gibt Kapitel 16.5.)

16.1 Komplexitätsmaße für die Aufwandsermittlung

Die Komplexitätsmaße beziehen sich auf verschiedene textuelle und graphische Darstellungen von Programmen: Quelltext, Kontrollflußgraph, Datenflußgraph, Datenfluß an Modulschnittstellen, Anweisungen (als Anwendung von Operatoren auf Operanden).

Quelltextlänge

Als Quelltextlänge wird die *Anzahl der Zeilen* im Quelltext (engl.: lines of code = LOC) definiert. Dabei kann differenziert werden, indem Kommentarzeilen mitgezählt oder weggelassen werden.

Vorteile des Maßes:

- Es ist automatisch meßbar.
- Das Maß korreliert mit dem Aufwand zum Erstellen bzw. zum Verstehen des Programms.
- Es ist auf alle Programmiersprachen anwendbar.

Nachteile des Maßes:

- Es ist erst nach der Implementierung (Codierung) einsetzbar.
- Es berücksichtigt keine anderen Faktoren, wie z. B. den Unterschied zwischen sequentiellen und parallelen Programmen, obwohl letztere bei gleicher Länge schwerer zu verstehen sind.

Zyklomatische Zahl $v(G)$

Für einen zusammenhängenden Kontrollflußgraphen G ist definiert:

$$v(G) := e - n + 2,$$

wobei e = Zahl der Kanten (edges), n = Zahl der Knoten (nodes) in G ist.

Es gelten folgende Beziehungen:

1. $v(G) = p + 1$ für strukturierte Programme, wobei p die Anzahl der Verzweigungsknoten ist. (In Abbildung 16.1 gilt: $p(G1) = 1$; $p(G2) = 3$.)
2. $v(G) = r$ = Anzahl der Gebiete, falls der Kontrollflußgraph planar ist.
   (**Gebiete** sind maximale zusammenhängende Punktmengen in der Ebene mit der Eigenschaft, daß je zwei Punkte durch eine Linie verbunden werden können, die keine Kante des Kontrollflußgraphen schneidet. In Abbildung 16.1 sind die Gebiete mit Nummern bezeichnet.
   Ein Graph heißt **planar**, wenn er sich in der Ebene ohne Überschneidungen der Kanten darstellen läßt.)
3. $v(G) = f$ = Anzahl der fundamentalen (linear unabhängigen) Wege im Kontrollflußgraphen (vgl. S. 201).

Vorteile der zyklomatischen Zahl:

- Sie ist automatisch meßbar.
- Sie korreliert mit dem Erstellungs- und Verstehensaufwand, insbesondere mit dem Testaufwand beim Kriterium *alle fundamentalen Wege* (vgl. S. 201).
- Wenn beim Feinentwurf Pseudocode erstellt wird, ist sie schon dann — vor der Codierung — einsetzbar.

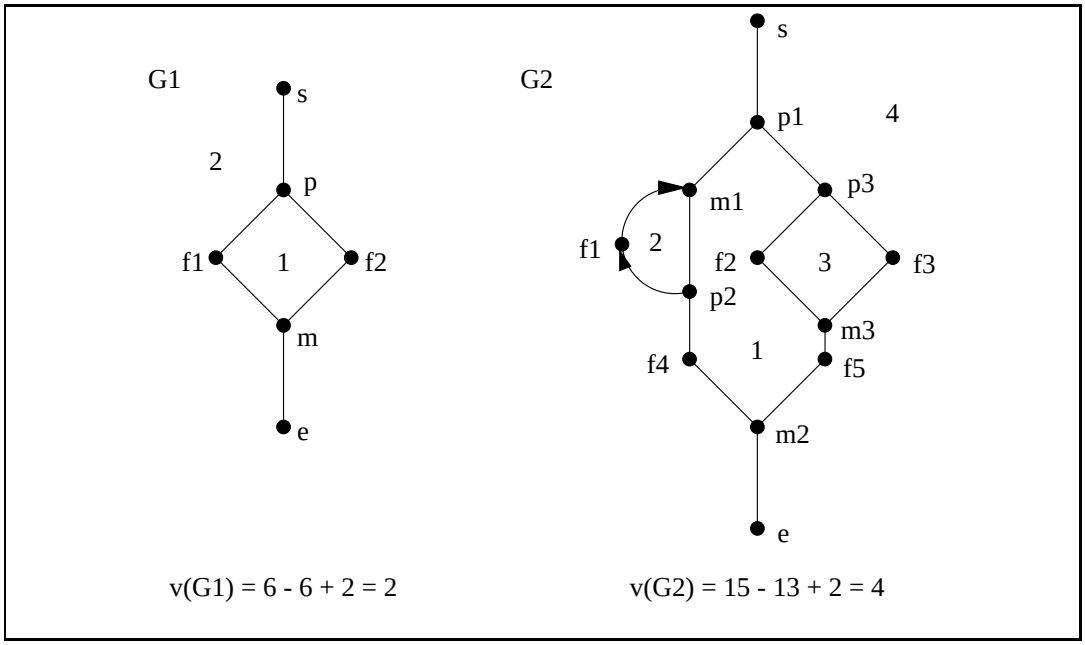

Abb. 16.1: Zusammenhängende Kontrollflußgraphen $G1$ und $G2$

Nachteile der zyklomatischen Zahl:

- Sie ist erst nach dem Algorithmenentwurf (und nur, falls dort Kontrollstruktur spezifiziert wird) einsetzbar.
- Es gibt keine Konstrukte für verteilte bzw. parallele Abläufe.
- Sie differenziert nicht zwischen unterschiedlich komplexen Verzweigungsprädikaten und unterschiedlichen Schachtelungen der Verzweigungen.
- Sie berücksichtigt nur die Anzahl der Verzweigungen im Kontrollfluß.

Um die Nachteile der zyklomatischen Zahl zu vermeiden, muß man Modifikationen vornehmen, welche z. B. *case*-Anweisungen und zusammengesetzte Entscheidungsprädikate höher bewerten als *if-then-else*-Anweisungen mit einfachen Entscheidungsprädikaten.

Datenflußkomplexität innerhalb eines Moduls

Bei diesem Ansatz werden Datenfluß- und Kontrollflußaspekte kombiniert. Grundlage des Maßes sind die Datenflußgraphen für die vorkommenden Variablen. Dabei entsteht ein **Datenflußgraph DFG(v) für eine Variable v** folgendermaßen aus dem Datenflußgraphen des Programms (s. Def. 8.1.1):

1. Er enthält nur die Knoten, in denen v referenziert (gelesen) oder definiert (verändert) wird, und den Anfangs- und Endknoten.
2. Ein Weg zwischen (nach 1) ausgewählten Knoten, auf dem v nicht angesprochen wird, wird durch eine Kante ersetzt.

Die **Datenflußkomplexität** eines Programms P mit Variablenmenge V ist definiert als die Summe aller zyklomatischen Zahlen der Datenflußgraphen für alle Variablen aus V.

Beispiel 16.1.1
Für das Suchprogramm SEARCH aus Abbildung 7.2 (auf Seite 193) erhält man z. B. folgende Datenflußgraphen DFG(v)[1]*:*
DFG(N): 3 Knoten (0, 1, 11), 2 Kanten (s. Abbildung 16.2)
DFG(NPOS): 4 Knoten (0, 1, 6, 11), 4 Kanten (s. Abbildung 16.2)

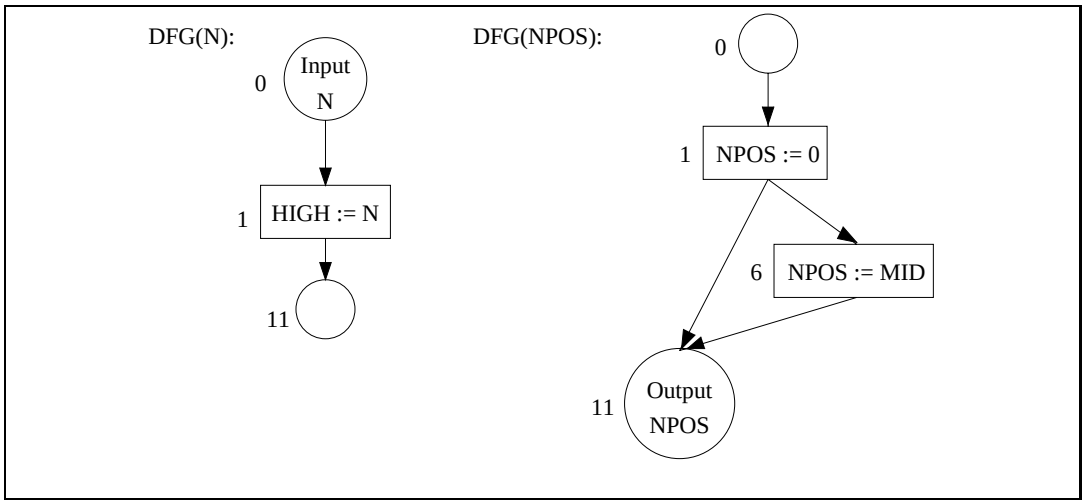

Abb. 16.2: Datenflußgraph für N und NPOS

Datenflußkomplexität zwischen Modulen

Die Komplexität des Datenflusses zwischen Modulen wird aufgrund der Komplexität der Schnittstellen berechnet. Transiente (nur durchgereichte) Daten werden gering bewertet (Gewicht 0,5), Eingabedaten etwas stärker (Gewicht 1), noch stärker die Ausgabedaten (Gewicht 2). Das höchste Gewicht (den Wert 3) erhalten Entscheidungsdaten. Das ergibt pro Modul die gewichtete Summe W.
Für iterative Aufrufe anderer Module werden noch die Entscheidungsdaten, die diese Iterationen steuern, betrachtet, zu einem Wert E gewichtet aufaddiert und als Repetitionsfaktor $R = 1 + \frac{E}{3}$ berücksichtigt (genaueres siehe [Cha 79]). Als **Datenflußkomplexität Q(M)** des Moduls M wird das geometrische Mittel von beiden Werten, d. h. die Wurzel aus ihrem Produkt definiert:

$$Q(M) := \sqrt{W * R} = \sqrt{W * (1 + \frac{E}{3})}$$

[1] zur vollständigen Berechnung der Datenflußkomplexität s. Übung 16.1b.

Die **Datenflußkomplexität Q(P)** eines Programms P mit den n Modulen $M_1, M_2, \ldots, M_n$ wird als arithmetisches Mittel der Datenflußkomplexitäten der einzelnen Module definiert:

$$Q(P) = \frac{1}{n} * \sum_{i=1}^{n} Q(M_i).$$

Operatoren-/Operanden-Komplexität

Der Operatoren-/Operanden-Komplexität liegt folgende Idee zugrunde: Die Berechnung eines Programms besteht aus Operatoren und Operanden und sonst nichts. Dabei werden auch Kontrollkonstrukte wie *if, for, while, goto*, „;“ (für *Sequenz*) und die *Klammerung* als Operator gezählt und Variablen und Konstanten sind Operanden. Von den folgenden vier Größen eines Programms können daher eine Reihe von Maßen abgeleitet werden:

O_1 = Zahl der verschiedenen Operatoren im Programm,
O_2 = Zahl der verschiedenen Operanden im Programm,
N_1 = Anzahl aller Operatorvorkommen im Programm,
N_2 = Anzahl aller Operandenvorkommen im Programm.

Davon abgeleitete Größen sind:

$O := O_1 + O_2$, das **Vokabular** des Programms,
$N := N_1 + N_2$, die **Länge** des Programms,
$D := \frac{N_2}{O_2}$, die **Datenhäufigkeit** im Programm.

In [Els 84] werden durch Korrelationsrechnungen vier Maße (aus 20 Maßen) als unabhängige Maße für verschiedene Programmeigenschaften bestimmt:

- $N = N_1 + N_2$ als Maß für die Größe,
- O_1 als Maß für die Nutzung der Programmiersprache,
- O_2 als Maß für die Datenverwendung,
- $D = \frac{N_2}{O_2}$ als Maß für die Datenflußkomplexität.

Mit diesen vier Maßen läßt sich der Aufwand VA (für das Verstehen) und der Aufwand PA (für das Programmieren) mit einer Konstanten S definieren:

$$VA = \log_2(O_1 + O_2) * O_1 * D * N$$

$$PA = \frac{\log_2(O_1 + O_2) * O_1 * D * (O_1 * \log_2 O_1 + O_2 * \log_2 O_2)}{2 * S}$$

PA ist die Zeit zum Codieren eines Programms, wobei die Zeit für Überlegungen, die mit Entwurfsfragen zu tun haben, nicht mitgezählt wird. In einem

Experiment wurde daher der Programmieraufwand folgendermaßen gemessen: Programmierern wurde ein ALGOL-Programm gegeben; zu schreiben war ein entsprechendes PL/I-, FORTRAN- oder APL-Programm. Frappierenderweise wurde dabei mit einer Abweichung von höchstens 8% der Wert von *PA* (ohne Tippaufwand etc.) in Sekunden erreicht, wobei mit $S = 18$, der (angeblichen) Anzahl der möglichen Unterscheidungen des menschlichen Gehirns (pro Sekunde), gerechnet wurde[2].

Diese Ergebnisse wurden in anderen Literaturstellen nur z. T. bestätigt, denn die zitierten Ergebnisse setzen anscheinend voraus:

- unstrukturierte, unkommentierte Programme, bei denen die Eingabe- und Ausgabeanweisungen und die Deklarationen nicht mitgezählt werden,
- wenig erfahrene Programmierer.

Bei diesen Voraussetzungen orientiert man sich als Programmierer notgedrungen an den Operanden und Operatoren des Programms und nicht an der Struktur des Programms (wie z. B. bei $Q(P)$, der anderen Definition von Datenflußkomplexität auf Seite 443).

Nutzen und Anwendung der Komplexitätsmaße:

- Besonders komplexe Programmteile, d. h. Komponenten, die schwierig zu überprüfen und zu modifizieren sind, können identifiziert werden. Bei diesen Programmteilen ist eine Erhöhung des Test- und Inspektionsaufwands oder eine Codeänderung angebracht. Eine hohe Datenhäufigkeit $D = \frac{N_2}{O_2}$ ist beispielsweise ein Indiz für eine Variablenverwendung mit mehreren Bedeutungen oder für lange Ausdrücke (die aufgeteilt werden sollten).
- Die Modulkomplexität sollte durch Zerlegung von Modulen mit sehr großer Komplexität reduziert werden. Allerdings darf dieses Vorgehen nicht übertrieben werden, da sonst (bei zu vielen Modulen) Probleme bei der interprozeduralen Kommunikation und der Dokumentation entstehen. (Publizierte Algorithmen [in den *CACM* etc.] mit einer Länge N von mehr als 260 haben typischerweise Fehler, kürzere Algorithmen nicht [nach Sullivan, zitiert nach [Cur 81], S. 214].)

Abschließend ist zu bemerken, daß Komplexitätsmaße nicht aussagen können, ob ein Programm besonders gut geschrieben ist, sondern lediglich, ob es besonders schlecht — nämlich zu komplex — ist.

[2] Zur Ableitung der Formel für *PA (= T)* und zu den dabei gemachten Annahmen siehe [Hal 78].

16.2 Auswahl von Softwareprüfmethoden

„Wer sich nicht entscheiden kann, will sich nicht entscheiden."
— Englische Spruchweisheit

Generell sollten solche Prüfmethoden angewandt werden, die im allgemeinen den höchsten Prozentsatz von Fehlern bzw. von Fehlern einer bestimmten Art mit möglichst geringem Aufwand aufdecken. Da es keine Methode gibt, die für alle Fehlerarten die höchste Fehleraufdeckungsrate und den geringsten Aufwand hat (s. Kapitel 6, 11 bis 14), sind generell — unter Berücksichtigung von Kosten, Qualitäts- und Sicherheitsanforderungen — *mehrere* Prüfmethoden anzuwenden. Sollen alle möglichen Fehlerarten aufgedeckt werden, müssen sehr viele Prüfmethoden eingesetzt werden. In einem konkreten Fall liegen aber meist nur gewisse Fehlerarten vor und es reicht eine Teilmenge dieser Testmethoden. Da aber erst nach dem Prüfen die konkret vorliegenden Fehlerarten bekannt sind und sich manche Prüfkriterien (z. B. die Anweisungsüberdeckung aus Kapitel 7.2) nicht an konkreten Fehlerarten orientieren, ist das Fehlerkriterium als Entscheidungsgrundlage für die Auswahl nicht geeignet.
Es müssen also aus den vorliegenden Spezifikationen und Programmen auf möglichst einfache Art Entscheidungskriterien für die Prüfmethodenauswahl abgeleitet werden. Ein Hauptkriterium ist natürlich das Paradigma, nach dem spezifiziert oder programmiert wurde[3]. Nur auf algebraische Spezifikationen lassen sich z. B. die Methoden aus Kapitel 5.2 anwenden, nur auf imperative Programme mit Schleifen lassen sich die meisten Kriterien aus Abschnitt 7.2.4 anwenden. Für eine Spezifikations- oder Programmart lassen sich aber meistens mehrere Prüfmethoden anwenden. Günstig wäre es, wenn **Maßzahlen** bestimmt werden könnten, die sich leicht aus der Spezifikations- bzw. Programmstruktur ableiten lassen und als Entscheidungskriterium geeignet sind.
Für imperative Programmstrukturen wird im folgenden ein Satz von Maßzahlen vorgestellt, welcher eine Auswahl aus 42 bekannten Prüfmethoden erlaubt. (Durch geeignete weitere Maßzahlen läßt sich der Anwendungsbereich auf andere Prüfmethoden erweitern.) Die Grundidee dabei ist, daß überdurchschnittlich komplexe Programmkomponenten die potentiell fehlerhaften Komponenten sind. Also müssen entsprechende Komplexitätsmaße herangezogen werden, die sich aber — im Gegensatz zu Kapitel 16.1 — meist auf spezielle Programmstrukturen beziehen. Es handelt sich um zehn Kontrollfluß-, elf Datenfluß-, drei Datenstruktur- und zwei Arithmetikmaße. Dies orientiert sich an den Testkriterien in Teil III dieses Buches.

[3] vgl. dazu auch Abschnitt 17.2.1

16.2.1 Definition der Kontrollflußmaße

1. Komplexitätstyp *Entscheidungen*:

$$Z_1 = \frac{\text{Anzahl der } \textit{repeat}\text{- und leeren } \textit{else}\text{-Entscheidungen}}{\text{Anzahl aller Entscheidungen}}$$

$$Z_2 = \text{Anzahl der Entscheidungen}$$

Z_1 ist der Anteil der Entscheidungen, denen ein leerer Zweig zugeordnet ist, der bei der Anweisungsüberdeckung nicht unbedingt ausgeführt wird[4].

2. Komplexitätstyp *Entscheidungen/Anweisungen*:

$$Z_3 = \frac{\text{Anzahl der Entscheidungen}}{\text{Anzahl der Anweisungen}}$$

Z_3 beschreibt die Kontrollintensität.

3. Struktur von *Entscheidungen*:

$$P_1 = \frac{\text{Anzahl der atomaren Prädikate}}{\text{Anzahl der Entscheidungen}}$$

$$P_2 = \frac{\text{Anzahl der atomaren und nicht atomaren Prädikate}}{\text{Anzahl der Entscheidungen}}$$

$$P_3 = \frac{\text{Anzahl der atomaren und nicht atomaren Prädikate}}{\text{Anzahl der atomaren Prädikate}}$$

$$P_4 = \frac{\text{Anzahl der arithmetischen Relationen in atomaren Prädikaten}}{\text{Anzahl der atomaren Prädikate}}$$

P_1 ist die mittlere atomare und P_2 die mittlere prädikative Komplexität von Entscheidungen. P_3 ist die mittlere Schachtelungskomplexität der Prädikate und P_4 der Anteil der arithmetischen Relationen an den atomaren Prädikaten.

4. Struktur von *Schleifen*:

$$S_1 = \frac{\text{Anzahl der Schleifenentscheidungen}}{\text{Anzahl der Entscheidungen}}$$

[4] Wenn eine *repeat-until*-Schleife (ohne Verzweigungen im Rumpf) genau einmal durchlaufen wird, ist die Anweisungsüberdeckung erfüllt, nicht aber die Überdeckung des leeren Rücksprungs an den Schleifenanfang.

$$S_2 = \frac{\text{Anzahl der nicht eingeschachtelten Schleifenentscheidungen}}{\text{Anzahl der Schleifenentscheidungen}}$$

$$S_3 = \frac{\text{Anzahl der Zählschleifenentscheidungen}}{\text{Anzahl der Schleifenentscheidungen}}$$

S_1 ist die Schleifenintensität, S_2 die mittlere Schleifenschachtelung und S_3 beschreibt den Anteil der Zählschleifen[5].

Die Maßzahl Z_1 erfaßt die leeren Zweige, die gerade den Unterschied bei der Anweisungs- und Zweigüberdeckung ausmachen (vgl. Kapitel 7.2). Das Maß Z_2 stellt einen Aspekt der Kontrollflußkomplexität dar und korreliert mit der zyklomatischen Zahl (s. Kapitel 16.1). Die Maßzahl Z_3 unterscheidet kontrollintensive von anweisungsintensiven Programmen und kann daher die Eignung von kontrollflußorientierten Testverfahren für die zu prüfenden Programme bewerten.
Die Maße P_1 bis P_4 erfassen die strukturelle Komplexität von Entscheidungen. Bei P_1, P_2, P_3 geht es um das Verhältnis von atomaren und nichtatomaren (also zusammengesetzten) Prädikaten in Entscheidungen. Beispielsweise enthält der Ausdruck $a = 2 \quad or \quad (b < 10 \quad and \quad c \geq 0)$ die drei atomaren Prädikate $a = 2$, $b < 10$ und $c \geq 0$, sowie die nichtatomaren Prädikate $(b < 10 \quad and \quad c \geq 0)$ und $a = 2 \quad or \quad (b < 10 \quad and \quad c \geq 0)$ — das komplette Prädikat. Mit den Maßen wird also beurteilt, ob Testverfahren einzusetzen sind, die besonders auf die Überprüfung zusammengesetzter Entscheidungsprädikate zugeschnitten sind (bei P_1, P_2, P_3) oder (bei P_4) auf das Testen arithmetischer Relationen mit den Vergleichsoperatoren $<$, $>$, $\leq$, $\geq$, $=$, $\neq$ (vgl. Kapitel 9.1, S. 233). Der Einsatz von speziellen Testverfahren für Schleifen (vgl. Abschnitt 7.2.4) ist sinnvoll, falls ein gewisser Anteil der Entscheidungen die Schleifen betrifft (Maß S_1). Die Maße S_2 und S_3 korrelieren mit der Anzahl der Wege im Kontrollflußgraphen des Programms, d. h. bei kleinen Werten von S_2 und S_3 kann es viele Wege (durch häufige Schleifendurchläufe) geben. Bestimmte Testverfahren für Schleifen sind dann aus Komplexitätsgründen nicht praktikabel (vgl. Beispiel 7.2.9 auf Seite 206 und Satz 10.2.1, Nr. 2 auf Seite 257).

16.2.2 Definition der Datenflußmaße

1. Komplexitätstyp *Datenzugriffe/Anweisungen*:

$$F_1 = \frac{\text{Anzahl der Definitionen, B- und E-Referenzen}^6}{\text{Anzahl der Entscheidungen}}$$

F_1 beschreibt die Datenflußintensität.

[5] Nur in Extremfällen — z. B. Anzahl der Entscheidungen = 0 — sind die Maße nicht definiert. In solchen Fällen ist das entsprechende Maß, z. B. Z_1, P_1, S_1, aber sowieso irrelevant.

[6] B-Referenzen sind Berechnungsreferenzen, E-Referenzen sind Entscheidungsreferenzen (vgl. Definition 8.2.4, S. 218).

2. Unterschiedliche Formen des *Datenzugriffs*:

$$F_2 = \frac{\text{Anzahl der Definitionen}}{\text{Anzahl der B-Referenzen} + \text{Anzahl der E-Referenzen}}$$

$$F_3 = \frac{\text{Anzahl der B-Referenzen}}{\text{Anzahl der Definitionen} + \text{Anzahl der E-Referenzen}}$$

$$F_4 = \frac{\text{Anzahl der E-Referenzen}}{\text{Anzahl der B-Referenzen} + \text{Anzahl der Definitionen}}$$

$$F_5 = \frac{\text{Anzahl der B-Referenzen}}{\text{Anzahl der B-Referenzen} + \text{Anzahl der E-Referenzen}}$$

$$F_6 = \frac{\text{Anzahl der E-Referenzen}}{\text{Anzahl der B-Referenzen} + \text{Anzahl der E-Referenzen}}$$

F_2, F_3 bzw. F_4 beschreiben das Verhältnis der Definitionen, B-Referenzen bzw. E-Referenzen zu den jeweils anderen beiden Zugriffsarten. F_5 bzw. F_6 ist die Relation der B-Referenzen bzw. E-Referenzen zu allen Referenzen.

3. Länge von *Datenflußketten*:

$$F_7 = \frac{\text{Gesamtlänge der } k\text{-DR-Interaktionen}}{\text{Anzahl der } k\text{-DR-Interaktionen}}$$

$$F_8 = \text{maximale Länge der } k\text{-DR-Interaktionen}$$

F_7 ist die mittlere Länge der k-DR-Interaktionen.

4. Datenzugriffe auf *bestimmte Datenstrukturen*:

$$F_9 = \frac{\text{Anzahl der Definitionen von Feldelementen, Zeigern und Dateien}}{\text{Anzahl der Definitionen}}$$

$$F_{10} = \frac{\text{Anzahl der E-Ref. von Feldelementen, Zeigern und Dateien}}{\text{Anzahl der E-Referenzen}}$$

$$F_{11} = \frac{\text{Anzahl der B-Ref. von Feldelementen, Zeigern und Dateien}}{\text{Anzahl der B-Referenzen}}$$

F_9, F_{10} bzw. F_{11} ist der Anteil der Definitionen, E-Referenzen bzw. B-Referenzen, die sich auf Strukturen (Feldelemente, Zeiger, Dateien) beziehen.

Die vorgestellten Datenflußmaße messen Eigenschaften, die mit der Eignung datenflußbezogener Testverfahren korrelieren. Sie können also für die Auswahl konkreter datenflußbezogener Testverfahren (s. Kapitel 8) herangezogen werden.
Die Maßzahl F_1 unterstützt z. B. die Entscheidung zwischen dem Einsatz von datenflußbezogenen oder kontrollflußbezogenen Testverfahren (s. Kapitel 7 oder 8). Die Maße F_2 bis F_6 ermitteln, welche der drei Zugriffsformen auf Variablen (Definition, B- oder E-Referenz) hauptsächlich vorkommen, was für den Einsatz eines entsprechenden Testverfahrens (s. Kapitel 8.2) relevant ist. Die Maßzahlen F_7 und F_8 unterstützen die Entscheidung, ob und mit welchem Wert von k mit dem Kriterium *alle k-DR-Interaktionen* getestet werden soll (s. Definition 8.3.3, Seite 222). Die Maße F_9 bis F_{11} messen den Anteil der unterschiedlichen Nutzungsformen von strukturierten Typen. Hohe Werte erschweren den Einsatz datenflußbezogener Testverfahren, da die betroffenen Komponenten der Datenstruktur nicht immer (statisch) ermittelt werden können (vgl. das Array-Problem, Kap. 12.2, S. 317).

16.2.3 Definition der Datenstrukturmaße

1. Komplexitätstyp *Datenstrukturen*:

$$D_1 = \frac{\text{Anzahl der Feld-, Zeiger- und Dateideklarationen}}{\text{Anzahl der Deklarationen}}$$

$$D_2 = \frac{\text{Anzahl der in F/D/Z deklarierten Datenelemente}^7}{\text{Anzahl der Deklarationen von Basistypen}}$$

   D_1 beschreibt die Datenstrukturkomplexität, während D_2 die „Masse“ dieser Datenstrukturdeklarationen mißt.

2. Komplexitätstyp *reelle Datentypen*:

$$D_3 = \frac{\text{Anzahl der Deklarationen reeller Datenelemente}}{\text{Anzahl der Deklarationen von Basistypen}}$$

   D_3 mißt den Anteil der reellen Datentypen an den Deklarationen.

Da die Strukturen Feld, Zeiger und Datei bei der symbolischen Ausführung, der formalen Verifikation und bei datenflußbezogenen Testmethoden Schwierigkeiten bereiten, mißt D_1 den Anteil dieser Datentypen bzw. D_2 den Anteil entsprechender Datenelemente bei den Deklarationen. Entsprechendes mißt D_3 für reelle Daten, da sie bei der symbolischen Ausführung und Verifikation Probleme bereiten, s. Stichwort „Real-Arithmetik“ auf S. 331. Hohe Werte von D_1, D_2 bzw. D_3 geben also die Entscheidungshilfe, die angesprochenen Prüfmethoden nicht einzusetzen.

[7] F/D/Z := Felder, Dateien und Zeiger

16.2.4 Definition der Arithmetikmaße

$$A_1 = \frac{\text{Anzahl der Zuweisungen}}{\text{Anzahl der Entscheidungen}}$$

$$A_2 = \frac{\text{Anzahl der arithmetischen Operatoren}}{\text{Anzahl der Zuweisungen}}$$

A_1 beschreibt die Berechnungsintensität, A_2 die Arithmetikkomplexität.

Bei kleinen Werten von A_1 reichen kontrollfluß- bzw. entscheidungsbezogene Testverfahren (s. Kapitel 7.2 und 9.2) aus, bei großen Werten von A_1 müssen dagegen berechnungsbezogene Verfahren (s. Kapitel 4.3 und 9.1) verwendet werden, insbesondere arithmetikbezogene Verfahren (s. Nr. 3 und 4 in Kapitel 9.1), wenn A_2 einen großen Wert hat.

16.2.5 Fünfstufige Maßwerte anstelle exakter Maßzahlen

Die ermittelten Komplexitätsmaße liefern Werte, die unnötig genau sind. Für die Auswahl der Testverfahren reichen dagegen Bewertungen der Form „die Komplexität ist minimal, gering, mittel, hoch oder maximal“. Diese fünf Stufen ergeben sich durch geeignete Klassenbildung bei den exakten Werten[8]. Was „geeignet“ ist, hängt von dem Einsatzzweck und gewonnenen Erfahrungen mit dieser Auswahlmethode ab. Die Klassenbildung sollte also veränderbar sein.

16.2.6 Beispiel: Anwendung der Komplexitätsmaße

Für drei kleine Module werden in den folgenden Absätzen nur die (deutlich verschiedenen) Maße angegeben. Modul 1 (GGT) berechnet den größten gemeinsamen Teiler zweier positiver ganzer Zahlen. Modul 2 (E/A), ein Modula-2-Bibliotheksmodul, dient der Ein- und Ausgabe natürlicher Zahlen. Modul 3 ist ein Numerikprogramm für die Integration nach der Simpson-Regel.

Modul 1 ist datenflußlastig ($F_1 = 12{,}5$ ist hoch), besitzt nur wenige Entscheidungen ($Z_2 = 2$ und $Z_3 = 0,14 = 14\%$ sind gering), die alle aus atomaren Prädikaten bestehen ($P_1 = P_2 = P_3 = 1$ ist minimal). Z_1, S_1, D_1, D_2, D_3 sind ebenfalls minimal ($= 0$); leere Zweige, Schleifen, strukturierte Daten und reelle Variablen sind also nicht vorhanden. (Die anderen Werte sind nicht relevant). Daher bieten sich einfache

[8] Dies entspricht der **Fuzzyfizierung** in der Theorie unscharfer Mengen und Werte (Fuzzy-Theorie, siehe z. B. [Alt 93], [KGK 95]). Hier reicht allerdings i. allg. eine strikte Klassenbildung aus, z. B. „Maß X hat den Wert *gering*“. Unscharfe Aussagen der Form „Maß X hat den Wert *gering* zu 70% und den Wert *mittel* zu 40%“ werden also nicht gebildet.

datenflußbezogene Testverfahren (z. B. *alle B-/einige E-Referenzen* oder *alle DR-Interaktionen*), formale Verifikation und symbolische Ausführung an.

Modul 2 ist dagegen kontrollflußlastig ($Z_2 = 28$ und $Z_3 = 0{,}27 = 27\%$ sind hoch bzw. mittel). Es besitzt zusammengesetzte (nicht atomare) Entscheidungsprädikate ($P_1 = 2{,}46$, $P_2 = 3{,}57$ und $P_3 = 1{,}45$ sind nicht minimal) und eine geringe Schleifenintensität ($S_1 = 0{,}11 = 11\%$), allerdings einen hohen Anteil leerer Zweige ($Z_1 = 0{,}57 = 57\%$). Es gibt Strukturen als Datentypen ($D_1 = 0{,}06$ und $D_2 = 0{,}48$ sind nicht minimal), sie werden jedoch nicht in Entscheidungs-Referenzen verwendet ($F_{10} = 0$). Die Datenflußintensität hat einen mittleren Wert ($F_1 = 4{,}89$). (Alle anderen Werte sind von untergeordneter Bedeutung.) Also muß ein Testverfahren eingesetzt werden, welches alle Zweige und zusammengesetzten Entscheidungen hinreichend testet (z. B. die [minimale] Mehrfachbedingungsüberdeckung). Als datenflußbezogenes Verfahren bietet sich das Kriterium *alle E-/einige B-Referenzen* an, da es (wegen der Nichtbenutzung strukturierter Daten in E[ntscheidungs]-Referenzen) keine Probleme bereitet.

Modul 3 ist noch stärker datenflußlastig als Modul 1 ($F_1 = 26{,}5$), wobei viele Berechnungs-Referenzen vorkommen ($F_3 = 1,21 > F_2 = 0,61 > F_4 = 0,08$). Es besitzt wenige, einfache Entscheidungen ($Z_2 = 2$, $P_1 = P_2 = P_3 = 1$ ist minimal). Dagegen ist es arithmetik- und berechnungsintensiv ($A_1 = 8{,}5$ ist hoch, $A_2 = 1{,}35$ mittel bis hoch) und verwendet sehr viele reelle Variablen ($D_3 = 0{,}85 = 85\%$). (Die anderen Werte sind in diesem Zusammenhang irrelevant.) Daher ist Modul 3 nicht gut formal zu verifizieren, jedoch gut mit dem Kriterium *alle B-/einige E-Referenzen* und arithmetikbezogenen Verfahren (wie z. B. dem additiven/multiplikativen Fehlerkriterium) zu testen.

16.2.7 Automatisierte Testmethodenauswahl

In obigem Beispiel 16.2.6 wurden die Bewertungen und Entscheidungen durch eine grobe, manuelle Interpretation der Komplexitätsmaße gewonnen. Diese Interpretation kann auch automatisch, durch ein regelbasiertes Expertensystem vorgenommen werden. Dieses geht von dem **Komplexitätsprofil** der zu prüfenden Software aus, welches durch die ermittelten 26 Maßzahlen gegeben ist. (Die Ermittlung der Maßzahlen ist ein Problem der statischen Analyse; vgl. Kapitel 12.2, S. 315). Durch die Anwendung von Regeln kann daraus einerseits eine Fehlerprognose für die Software abgeleitet werden; andererseits kann die Eignung der 42 Prüfverfahren für die vorliegende Prüfaufgabe ermittelt werden, indem Wissen über die Eigenschaften der Prüfverfahren und die Softwarekomplexität verwendet wird. In einem letzten automatischen Schritt werden Kombinationen von Prüfverfahren generiert, die alle prognostizierten Fehler abdecken, möglichst redundanzfrei sind und allgemein akzeptierte Anforderungen an Prüfungen (z. B. C_1-Überdeckung) erfüllen. (Genaueres siehe in [Lig 93a] bzw. [Lig 93b].)

16.3 Kombination von Softwareprüfmethoden

Die Kombination von Prüfmethoden kann relativ lose oder ziemlich streng gekoppelt sein.
Eine **lose** Kombination ist eine Anwendung von Prüfverfahren in bestimmter sequentieller Reihenfolge nacheinander oder unabhängig voneinander (nebenläufig bzw. parallel). Als sequentielle Reihenfolge empfiehlt sich (für jede Entwicklungsphase, falls möglich):

1. informelle Prüfung (s. Kapitel 12.1),
2. formale statische Analyse und symbolische Ausführung (s. Kapitel 12.2, 12.3),
3. spezifikationsbezogenes Testen (s. Kapitel 4 und 5),
4. implementationsbezogenes Testen (s. Kapitel 7 bis 9),
5. Überprüfung von Zusicherungen (dynamisch — s. Abschnitt 15.2.4 — oder statisch als formale Programmverifikation — s. Kapitel 12.4).

Auf alle Fälle sollte ein gutes Werkzeug zur statischen Analyse eingesetzt werden. Damit werden viele Fehler ohne die Mühe des Erstellens von Testfällen und Testdaten gefunden. Falls Spezifikation und Entwurf nicht formal sind, sollten sie sorgfältig informell geprüft werden — aber auch für Programme ist dies am Anfang günstig. Beim dynamischen Testen sollten Testfälle und Testdaten spezifikationsorientiert erstellt und später weitere Testdaten erzeugt werden, um hohe Werte für die ausgesuchten implementationsorientierten Überdeckungskriterien zu erreichen.
Zumindest bei kritischer Software, d. h. Software mit hohen Sicherheitsanforderungen, sollte mit Zusicherungen gearbeitet werden: Abprüfung von dynamischen Zusicherungen während des Programmlaufs und/oder manueller Induktionsbeweis bei (kritischen) Schleifen — eventuell unterstützt von geeigneten Werkzeugen. Umfangreiche Software muß natürlich modular und inkrementell getestet werden und zum Schluß müssen System- und Abnahmetests erfolgen (s. Kapitel 13).

Eine **strenge** Kopplung ist für Prüfmethoden möglich, die sich vordergründig auf verschiedene Aspekte beziehen, die aber bei geeigneter Abstraktion vergleichbar sind.
Ein Beispiel dafür ist eine spezifikationsorientierte Methode, die sich an Datenbereichen orientiert (Äquivalenzklassenbildung, s. Abschnitt 4.2.1) und eine implementationsorientierte Methode, die sich an Wegen durch den Kontrollflußgraphen orientiert, z. B. die Pfadüberdeckung (s. Abschnitt 7.2.4). Betrachtet man die Menge der Eingabedaten, die einen bestimmten Weg[9] durch den Kontrollflußgraphen ausführen, so erhält man eine **Weg-Äquivalenzklasse**: Es ist die Menge der Eingabedaten, die

[9] Bei *while*- und *repeat*-Schleifen sind wieder endlich viele Klassen von Wegen zu bilden (vgl. Abschnitt 7.2.4)

vom Programm äquivalent behandelt werden, während die spezifikationsorientierten Äquivalenzklassen Mengen von Eingabedaten sind, die äquivalent behandelt werden sollen. Seien $S_1, \ldots, S_n$ die n spezifikationsorientierten Äquivalenzklassen und $P_1, \ldots, P_m$ die sich (bei geeigneter Wegeklassenbildung bei Schleifen) ergebenden m Weg-Äquivalenzklassen. Dann gibt es mehrere Fälle für eine Weg-Äquivalenzklasse P_i:

1. $P_i = S_j$ für ein j $(1 \leq j \leq n)$, d. h. $P_i \cap S_j = P_i = S_j$:
   Der zu P_i gehörige Weg realisiert genau den durch S_j beschriebenen Fall. Der Datenbereich S_j ist also korrekt implementiert.

2. $P_i \subset S_j$ für ein j $(1 \leq j \leq n)$, d. h. $P_i \cap S_j = P_i \neq S_j$:
   Der zu P_i gehörige Weg realisiert einen „Unterfall“ des durch S_j beschriebenen Falles. Das kann eine erlaubte oder sogar (aus Datenstrukturgründen) notwendige Verfeinerung sein. Ob das der Fall ist, sollte aber überprüft werden.

3. $P_i \cap S_j \neq \emptyset$ und $P_i \neq P_i \cap S_j \neq S_j$ für ein j $(1 \leq j \leq n)$:
   Der zu P_i gehörige Weg realisiert (nur) einen Unterfall des durch S_j beschriebenen Falles und auch noch Fälle, die durch andere spezifikationsorientierte Äquivalenzklassen ganz oder zum Teil beschrieben sind. Die entsprechenden Datenbereiche überlappen sich also nur teilweise. Das dürfte fast immer ein Fehler sein. Ob dies der Fall ist, kann durch Inspektion, symbolische Ausführung, (in-)formelle Verifikation oder durch Ausführung von Testdaten aus $P_i \cap S_j$, dem sogenannten **fehleraufdeckenden Teilbereich**, festgestellt werden.

16.4 Abschätzung von Fehleranzahl und Zuverlässigkeit

Das Testen kann beendet werden, wenn die (Rest-)Fehleranzahl im Programm bzw. die Wahrscheinlichkeit für ein Fehlverhalten — die Zuverlässigkeit — klein genug (idealerweise gleich Null) ist. Beide Werte sollten also ermittelt werden.

16.4.1 Abschätzung der Fehleranzahl

Es gibt zwei Methoden zur Abschätzung der Fehleranzahl:

- Fehlereinpflanzung (error seeding),
- Testen durch zwei unabhängige Gruppen.

Fehlereinpflanzung[10]
Die Methode der Fehlereinpflanzung umfaßt folgende Schritte:

1. Von Personen, welche die bisher gemachten Tests T nicht kennen, oder von Werkzeugen, die unabhängig von der Testdatengenerierung und Testdatenbewertung arbeiten, werden zusätzliche Fehler in das Programm P eingepflanzt, wodurch ein Programm P' entsteht.

2. Die Menge aller bisher für P durchgeführten Tests T wird mit P' durchgeführt.

3. Die beim Test von P' mit T gefundenen eingepflanzten Fehler werden bestimmt. Außerdem sind aus den bisherigen Tests die echten Fehler von P bekannt. Also kann notiert werden:

   GE = Anzahl der gefundenen eingepflanzten Fehler,
   E = Anzahl der eingepflanzten Fehler,
   GF = Anzahl der gefundenen echten Fehler.

4. Es wird unterstellt, daß echte und eingebaute Fehler mit derselben Rate gefunden werden. Dann gilt:

   $$\frac{GF}{F} = \frac{GE}{E} \tag{16.1}$$

   wobei F die Anzahl aller echten Fehler ist.

5. Die Anzahl F der echten Fehler ergibt sich dann aus 16.1:

   $$F = E * \frac{GF}{GE} \tag{16.2}$$

Vorteile der Methode der Fehlereinpflanzung:

- Der Testaufwand ist gering, da nur *ein* Programm zu testen ist. (Dies ist ein großer Vorteil gegenüber der Mutationsanalyse [vgl. Kapitel 9.3].)
- Man erhält eine Abschätzung der Anzahl der echten Fehler.

Nachteile der Methode der Fehlereinpflanzung:
Bei Schritt 3 ist (bei einem Fehlverhalten von P') die Ermittlung des verursachenden eingepflanzten Fehlers nicht immer eindeutig möglich; evtl. können mehrere Fehler das Fehlverhalten bewirken[11].
Bei Schritt 4 ist die Annahme der Gültigkeit von Gleichung 16.1 problematisch[12]. Sie setzt folgendes voraus, was bei Softwareprodukten meist nicht erfüllt ist:

[10] engl. „bebugging“ — statt „debugging“ — genannt

[11] Dieser Nachteil tritt bei der Mutationsanalyse (s. Kap. 9.3) nicht auf, da dort beim Erzeugen der Mutanten nur jeweils *ein* Fehler eingepflanzt wird.

[12] Eine Abschätzung der Wahrscheinlichkeit, daß Gleichung 16.1 gültig ist, selbst wenn die folgenden Voraussetzungen erfüllt sind, findet man bei Myers (s. [Mye 76], S. 337).

- eine sehr große Anzahl von Fehlern im Programm,
- eine gleichmäßige Verteilung der Fehler im Programm,
- eine Verteilung der eingepflanzten Fehler, die der Verteilung der echten Fehler entspricht,
- eine gleiche Wahrscheinlichkeit, echte und eingepflanzte Fehler mit Tests zu entdecken,
- keine Wechselwirkung zwischen den echten und den eingepflanzten Fehlern.

Testen durch zwei unabhängige Gruppen
Bei dieser Methode sind folgende Schritte auszuführen:

1. Zwei unabhängige Gruppen G_1 und G_2 entwickeln jeweils Testdatenmengen T_1 und T_2 für das Programm P.
2. Die Anzahl F_i der von der Gruppe G_i entdeckten Programmfehler (für $i = 1, 2$) und die Anzahl $F_{1\cap 2}$ der Fehler, die sowohl von Gruppe G_1 als auch von Gruppe G_2 gefunden wurden, ist zu bestimmen.
3. Sei F die Anzahl aller in P vorhandenen Fehler. Dann dürfen unter der Annahme, daß beide Testgruppen bei allen Fehlern und beliebigen Teilmengen eine konstante **Testeffizienz** (Wahrscheinlichkeit der Aufdeckung von Fehlern) haben, folgende Gleichungen aufgestellt werden:

$$\frac{F_1}{F} = \frac{F_{1\cap 2}}{F_2} = \text{Testeffizienz von Gruppe } G_1; \tag{16.3}$$

$$\frac{F_2}{F} = \frac{F_{1\cap 2}}{F_1} = \text{Testeffizienz von Gruppe } G_2. \tag{16.4}$$

4. Aus der Gleichung 16.3 (und 16.4) ergibt sich für die in P vorhandenen Fehler:

$$F = \frac{F_1 * F_2}{F_{1\cap 2}} \tag{16.5}$$

5. Die Zahl der noch nicht gefundenen Fehler ergibt sich folgendermaßen:

$$NF = F - F_{1\cup 2} = F - (F_1 + F_2 - F_{1\cap 2}), \tag{16.6}$$

wobei $F_{1\cup 2}$ die Anzahl der von beiden Gruppen insgesamt gefundenen Fehler ist. Aus 16.5 und 16.6 erhält man:

$$NF = \frac{(F_1 - F_{1\cap 2}) * (F_2 - F_{1\cap 2})}{F_{1\cap 2}} \tag{16.7}$$

Die bei den Gleichungen 16.3 und 16.4 unterstellte Konstanz der Testeffizienz ist problematisch. Vielmehr wird die Testeffizienz auf den von der jeweils anderen Gruppe *gefundenen* Fehlern i. allg. besser sein als auf den nicht gefundenen Fehlern, da sie leichter zu finden sind. Daher gilt:

$$\frac{F_{1\cap 2}}{F_2} > \frac{F_1}{F} \quad \text{und} \quad \frac{F_{1\cap 2}}{F_1} > \frac{F_2}{F}. \tag{16.8}$$

In jedem Fall also

$$F > \frac{F_1 * F_2}{F_{1\cap 2}}. \tag{16.9}$$

Damit gibt es also nur eine *untere* Abschätzung für die Zahl der Fehler.

Die Methoden des Fehlereinstreuens und des Testens durch zwei unabhängige Gruppen liefern also nur den Hinweis, wann der Testvorgang weitergehen muß: Wenn noch nicht die geschätzte Anzahl der Fehler gefunden wurde. Dies ist ein wichtiger psychologischer Faktor beim Testen. Das Erreichen dieser Anzahl gibt aber keinen verläßlichen Hinweis darauf, ob aufgehört werden kann. Daher sind zusätzliche Verfahren anzuwenden (s. Kap. 9.3 und 16.5).

16.4.2 Zuverlässigkeit

Das stochastische Auftreten von Softwarefehlern im Ablauf der Zeit kann als Zählprozeß N beschrieben werden, wobei $N(t)$ die Zahl der zwischen dem Zeitpunkt $t_0 = 0$ und dem Zeitpunkt t aufgetretenen Fehler angibt (für $t \geq 0$).

Als **Zuverlässigkeit R(t)** wird dann die Wahrscheinlichkeit für ein fehlerfreies Arbeiten bis zum Zeitpunkt t, d. h. $N(t) = 0$, definiert.

Die verschiedenen Modelle für die Zuverlässigkeitsberechnung machen leider Annahmen, die in der Realität nicht stimmen oder nur schwer zu erfüllen sind, z. B.:

1. Die Zahl der anfänglichen Programmfehler kann zuverlässig geschätzt werden. (Dies gilt nur bei speziellen Voraussetzungen; siehe vorherigen Abschnitt 16.4.1.)
2. Die Fehlerrate ist proportional zur Zahl der verbliebenen Fehler. (Wenn die Fehlerrate z. B. auf die Hälfte sinkt, sind i. allg. aber noch mehr als 50% der anfänglichen Fehler vorhanden. Die restlichen Fehler sind nämlich schwerer zu entdecken, da sie meist unter Bedingungen auftreten, die seltener vorkommen.)
3. Die versuchte Korrektur eines Fehlers vermindert die Zahl der verbleibenden Fehler um 1. (Leider werden oft neue Fehler gemacht bzw. Fehlerursachen nur z. T. beseitigt.)
4. Die Programmgröße bleibt im Softwarelebenszyklus konstant. (Weil es Ergänzungen und Erweiterungen gibt, stimmt dies i. allg. nicht.)

Ein Hauptkritikpunkt an diesen Modellen ist der Ansatz, die Zuverlässigkeit eines *Produkts* (d. h. hier: eines Programms) bestimmen zu wollen. Die entsprechenden Modelle aus der Produktion von materiellen Gütern setzen aber alle eine große Stückzahl voraus. Durch Stichproben kann die Zuverlässigkeit des Produktionsprozesses und damit die Qualität der in Zukunft produzierten Güter abgeschätzt werden. (Beispielsweise wird jeder 100. Feuerwerkskörper testweise gezündet.) Ein Softwareprodukt wird aber nur einmal entwickelt. Es könnte höchstens versucht werden, die Zuverlässigkeit eines Programmierers oder eines Programmierteams zu bestimmen[13]. Dafür ist aber i. allg. die Zahl der geschriebenen Programme zu klein und die einzelnen Programmentwicklungen sind i. allg. nicht vergleichbar.
Man kann die **Zuverlässigkeit** von Software sinnvollerweise nur als das eher subjektive Maß des Vertrauens in das korrekte Operieren der Software definieren. Dies kann allerdings objektiviert werden, wenn man die Güte der Tests, mit der die Software getestet wurde, oder die Restfehleranzahl bestimmt. Ansätze dazu sind die Mutationsanalyse (s. Kapitel 9.3) und die Abschätzmethoden des vorherigen Abschnitts. Zuverlässigkeitsmodelle können noch unter folgendem Aspekt kritisiert werden: Es ist ziemlich egal, wie viele Fehler im Programm verblieben sind, von Interesse ist eigentlich nur folgendes (vgl. Kap. 2.2 und 2.3):

- wie und wann sich die Fehler auswirken,
- welche Kosten bzw. welchen Schaden die Fehler verursachen.

Beispiel 16.4.1
Eine Anekdote besagt, daß im Steuerprogramm der Venussonde „Mariner 1“ die folgende Zählschleife eines Fortran-Programmteils einen Fehler enthielt:

```
DO 3 I = 1, 3
⋮
3 CONTINUE
```

Das Komma war durch einen Punkt ersetzt worden, was eine Zuweisung des Wertes 1.3 an die Variable DO3I ergab. Als Folge davon kam die Venussonde vom Kurs ab und mußte am 22. Juli 1962 nach 290 Sekunden Flugzeit gesprengt werden. Damit waren 18,5 Millionen Dollar „verpulvert“ worden.

[13] vgl. CMM-Ansatz in Kapitel 3.7

16.5 Spezielle Managementfragen

Beim Testen großer Systeme ist das Projektmanagement gefordert:

- Tausende von Testdaten müssen definiert und ausgeführt und die entsprechenden Ergebnisse müssen überprüft werden.
- Tausende von Fehlern müssen korrigiert werden.
- Eine Vielzahl von Mitarbeitern ist zur gleichen Zeit über einen Zeitraum von einem halben Jahr oder mehr beschäftigt.

Aber auch bei kleineren Systemen ist die Planung des Testens notwendig. Um die Steuerung dieser Vorgänge in den Griff zu bekommen ist ein **Testplan** erforderlich, der folgende Inhalte haben sollte:

1. Definition der Zielvorstellung einer jeden Testphase. (Beispiel: Beim Testen von Modul D soll eine hundertprozentige Zweigüberdeckung erreicht werden.)
2. Auswahl von Komplexitätsmaßen für die Steuerung des Test- und Modifikationsprozesses (vgl. Kapitel 16.1, 16.2).
3. Kriterien für die Beendigung des Testens (vgl. Kapitel 9.3, 16.4).
4. Einen Zeitplan, der festlegt, wann die Testdaten definiert, ausgeführt bzw. überprüft werden sollen. (Beispiel: Modultestdaten sollen schon am Ende der Entwurfsphase definiert werden).
5. Verantwortlichkeiten
   Dabei ist festzulegen, welche Mitarbeiter die Testdaten definieren, ausführen und überprüfen, wer die entdeckten Fehler behebt, wer als Schlichter fungiert, wenn unklar ist, ob ein Ergebnis der (zweideutigen) Spezifikation widerspricht.
6. Standards für das Entwerfen, Beschreiben und Speichern von Testdaten in Testdatenbibliotheken.
7. Angaben darüber, wer Testwerkzeuge entwickelt oder benutzt und wann und wie sie angewendet werden.
8. Planung der für jede Testphase benötigten Rechenzeit.
9. Planung der eventuell für das Testen zusätzlich benötigten Hardware.
10. Planung der Zusammensetzung der Programmteile beim Integrationstest (siehe Kapitel 13.3).

11. Vorgaben in der Form von Zeitplänen, Ressourcen und Beendigungsmerkmalen für die verschiedenen Aspekte des Testfortschritts und die Lokalisierung fehleranfälliger Module.

12. Mechanismen für die Fehlermeldung, Fehlerverfolgung und Fehlerkorrektur bei der Fehlerbehebung (siehe Kapitel 15).

13. Planung der Testdaten, die nach einer Programmänderung noch einmal — bei einem Regressionstest — ausgeführt werden sollen.

Kriterien für die Beendigung des Testens anzugeben, ist eines der schwierigsten Probleme. In der Praxis wird meistens die folgende bedeutungslose Regel angewandt, welche die Qualität der Testdatenmenge außer acht läßt:

Der Test wird beendet, wenn

(a) bei keinem Testlauf ein Fehler aufgetreten ist und

(b) die geplante Testzeit abgelaufen ist.

Das Testziel sollte jedoch positiv formuliert werden:

1. Festlegung der Anzahl der Fehler, die entdeckt werden sollen (z. B. bei der Inspektion vier bis acht Fehler pro 100 Anweisungen).

2. Festlegung des Testzeitraums.

Wenn die Zahl der Fehler bei obiger Festlegung überschätzt wurde (also nicht genügend viele gefunden wurden), ist von einem möglichst unabhängigen Prüfer zu beurteilen, ob die Testdaten eine genügend große potentielle Fehleraufdeckungsqualität hatten. (Dazu sind die Kriterien aus Teil II und III heranzuziehen.) Falls ja, kann der Test dennoch beendet werden, sonst ist die Testdatenmenge zu erweitern bzw. zu verbessern.

Zur Festlegung des Testzeitraums empfiehlt sich folgendes Vorgehen:

1. Festlegung eines Zeitraums aufgrund von Erfahrungswerten, z. B. 35% der Gesamtentwicklungszeit (vgl. [Boe 77], Figure 3).

2. Beobachtung des Testerfolgs über die Testzeit und gegebenenfalls Entscheidung, die Testzeit zu verlängern bzw. abzukürzen. Dazu ist die Fehleraufdeckungsrate (gefundene Fehler pro Tag bzw. Woche) zu ermitteln und über die Zeit aufzutragen.

Das Testen kann demnach beendet werden,

- wenn die geschätzte Zahl der Fehler gefunden wurde *oder*
- wenn die Testzeit abgelaufen ist und die Fehleraufdeckungsrate schon sehr klein (idealerweise gleich 0) ist.

16.6 Übungen

Übung 16.1:

(a) Ermitteln Sie für die Programme aus Abbildung 7.1 und 7.2 sowie aus Beispiel 7.3.1 (Kapitel 7.3) die zyklomatische Zahl $v(G)$.

(b) Berechnen Sie die Datenflußkomplexität des Programms SEARCH aus Abbildung 7.2 (vgl. Beispiel 16.1.1).

Übung 16.2:

Bestimmen Sie die Weg-Äquivalenzklassen (s. Kapitel 16.3) des folgenden Programms zur Dreiecksklassifikation und vergleichen Sie diese Klassen mit den Äquivalenzklassen, die zur Spezifikation gemäß Tabelle A.1 im Anhang A.1 gehören. Geben Sie alle fehleraufdeckenden Teilbereiche an.

```
procedure dreiecksklasse(A, B, C: integer);
    begin
        if A ≥ B and B ≥ C
        then if A = B or B = C
            then if A = B and B = C
                then writeln(„gleichseitig“);
                else writeln(„gleichschenklig“);
            else if A * A = B * B + C * C
                then writeln(„rechtwinklig“);
                else if A * A > B * B + C * C
                    then writeln(„stumpfwinklig“);
                    else writeln(„spitzwinklig“);
        else writeln(„illegal“);
    end;
```

16.7 Verwendete Quellen und weiterführende Literatur

Eine Übersicht über **Komplexitätsmaße** findet man z. B. bei Pocsay/Rombach sowie Curtis (s. [PoR 84], S. 37 ff.; [Cur 81]). Neuere Ansätze werden von Basili und Dumke/Zuse vorgestellt (s. [Bas 90] und [DuZ 94]), von Dumke insbesondere für Spezifikationsmaße, Maße für funktionale, logische und objektorientierte Programme, Maße zur Bestimmung des Testniveaus und der Zuverlässigkeit und Wartungsmaße (s. [Dum 92]). Die **zyklomatische Zahl** $v(G)$ stammt von McCabe (s. [McC 76]). Die Modifikationen dieses Maßes bzgl. der *case*-Anweisungen bzw. Schachtelung von Verzweigungen haben Myers und Basili/Reiter bzw. Roth/Lesshafft vorgeschlagen (s. [BaR 80], [Mye 77], [RoL 87]). Eine Modifikation, die auch die Reihenfolge im Programmtext berücksichtigt, beschreibt Woodward (s. [WHH 79]). Das Konzept der **Datenflußkomplexität** eines Programms findet sich bei Chapin und Sunokara et al. (s. [Cha 79], [Su& 81]). Ähnliche Konzepte präsentieren Rapps/Weyuker und Tai (s. [RaW 82], [Tai 84]). Chapin hat die Datenflußkomplexität zwischen Modulen definiert (s. [Cha 79]). Die **Operatoren-/Operanden-Komplexität** wurde von Halstead formuliert (siehe [Hal 78]). Die Voraussetzungen für die Anwendbarkeit des Maßes wurden experimentell von Curtis untersucht (s. [Cu& 79]). Experimente zur Aussagekraft der Komplexitätsmaße wurden u. a. von Elshoff und Potier et al. durchgeführt (s. [Els 84], [Po& 82]). Eine mathematische Formulierung der zu fordernden Eigenschaften der Maße für Größe, Länge, Komplexität, Kohäsion und Kopplung von Modulen wird von Briand/Morasca/Basili vorgestellt (s. [BMB 96]).

Die **Komplexitätsmaße für die Auswahl von Softwareprüfmethoden** stammen von Liggesmeyer (s. [Lig 93a], [Lig 93b]). Eine verständliche Einführung in die **Fuzzy-Logik** mit einer ausführlichen Beschreibung von Anwendungsbeispielen findet man in [Alt 93], eine formale, präzise Darstellung dieser unscharfen Logik dagegen in [KGK 95].

Weyuker/Ostrand haben vorgeschlagen, die spezifikationsorientierten Äquivalenzklassen mit den Weg-Äquivalenzklassen zu kombinieren, um fehleraufdeckende Teilbereiche zu erhalten (s. [WeO 80]). Die vorteilhaften Effekte der **Kombination** verschiedener Prüfmethoden hat Selby empirisch bestimmt (s. [Sel 86]).

Das Originalmodell zur **Fehlereinpflanzungsmethode** stammt von Mills (siehe [Mil 72]). Die hier verwendete Darstellung und die Methode des Testens durch zwei unabhängige Gruppen findet man bei Myers (s. [Mye 76]). Von Knight/Ammann wurden die Fehlereinpflanzungsmethoden experimentell bewertet (s. [KnA 85]).

Moawad hat das Auftreten von Softwarefehlern als Zählprozeß $N(t)$ beschrieben (s. [Moa 84]). Die Kritik bzgl. der gemachten Annahmen bei den **Zuverlässigkeitsmodellen** bezieht sich auf die Modelle von Musa, Goel/Okumoto und Littlewood/Verral (s. Referenzen in [TLP 95], S. 407). Ein kompliziertes Modell, welches diese Annahmen fast alle fallen läßt, stammt von Littlewood (s. [Lit 80]). Einen

Ansatz, der die Kosten von Fehlern berücksichtigt und gut bei Zustandsmodellen anwendbar ist, stellt Weyuker vor (s. [Wey 95]). Ein Konzept zur Zuverlässigkeitsmessung für umfangreiche kommerzielle Software und Hinweise auf neuere Literatur enthält [TLP 95].

Die Empfehlungen zum **Testmanagement** und Beenden des Testens orientieren sich an Myers und Daly (s. [Mye 76], Kap. 14; [Mye 79], Kap. 6; [Dal 77]). Dalal/McIntosh geben eine ökonomisch motivierte Regel für die Beendigung des Tests großer Systeme an, die von folgenden Parametern abhängt: der Zahl der gefundenen Fehler, den Testkosten pro Zeiteinheit, der exponentiell verteilten Fehlerfindungsrate μ und der Differenz der Kosten für die Fehlerbeseitigung beim Testen und „im Feld" [beim Kunden] (s. [DaM 94]). Weitere zu beachtende Aspekte — wie z. B. **Projektaudits** (zur Überprüfung der Wirksamkeit eines Qualitätsmanagementplans in einem Projekt), Erfassung der Fehlerkosten, Überwachung von Softwareänderungen, Konzepte der Personalführung und -schulung und des Werkzeugeinsatzes und, nicht zuletzt, die Erziehung zur Qualität — findet man in [FLS 91b], Kapitel 5 bis 7.

Eine **Übersicht** über das Messen (aber auch Prüfen und Bewerten) von Software in allen Phasen der Entwicklung geben Hausen et al. (s. [HMS 87]).

Im ESPRIT-Projekt **AMI** (application of metrics in industry) wurde eine Methode entwickelt, mit der Maße definiert und installiert werden können, die nicht nur testbezogen sind, sondern sich auch auf andere Aspekte — wie z. B. Produktivität, Zuverlässigkeit, Kosten — beziehen (s. [Sch 92]).

17 Zusammenfassung und Ausblick

17.1 Zusammenfassung

Die in diesem Buch vorgestellten spezifikationsorientierten Testmethoden haben sich an verschiedenen Spezifikationsarten orientiert: an Ein- und Ausgabe-Datenbereichen, funktionalen Formen, Datenflußdiagrammen, Petri-Netzen, endlichen Automaten, Pfadausdrücken und algebraischen Spezifikationen (siehe Kapitel bzw. Abschnitt 4, 5, 14.4.1 und 16.3).
Beim implementationsorientierten Testen wurde dagegen nur eine Implementationsart betrachtet: die Implementation als **imperatives** Programm. Ein solches Programm verwendet Variablen, die als Speicher für Werte dienen, und Anweisungen, die diese Werte lesen (referenzieren) und schreiben (definieren). Damit ist ein Datenfluß gegeben (s. Kapitel 8). Die Reihenfolge, in der die Anweisungen ausgeführt werden (können), wird als Kontrollfluß bezeichnet. Dabei wird meistens ein sequentieller, deterministischer Kontrollfluß verwendet (s. Kapitel 7). Ein nichtsequentieller, nebenläufiger Kontrollfluß ist zwangsläufig nichtdeterministisch (s. Kapitel 14.1). Die vorgestellten Prüfmethoden (insbesondere die Testmethoden von Kapitel 7 bis 9 und 13 sowie Abschnitt 14.4.2, aber auch die statischen Prüfmethoden von Kapitel 12 und 14.3, die Maße von Kapitel 16.1 und 16.2 sowie die Fehlerlokalisierungsmethoden der Abschnitte 15.2.3 bis 15.2.8) haben sich genau an den entsprechenden Programmstrukturen (Kontrollfluß, Datenfluß, Ausdrücke, Anweisungen und Datenstrukturen) orientiert.

17.2 Ausblick

17.2.1 Anpassung an neue Spezifikations- und Programmierparadigmen

Als Ausblick ist zu diskutieren, für welche Spezifikationsverfahren und Implementierungsverfahren keine angemessenen Prüfverfahren vorgestellt wurden bzw. wie die Prüfverfahren geeignet erweitert oder modifiziert werden können, um neuen Spezifikationsmethoden und Programmierparadigmen gerecht zu werden.

Beim **objektorientierten** Entwerfen und Programmieren (mit Klassen bzw. Objekten mit sogenannten Methoden und Vererbung) wirkt sich vor allem der Vererbungsmechanismus auf die Prüfverfahren aus. Die geeignete Reihenfolge und der

Zusatzaufwand beim Integrationstest von Modulen hängt jetzt nicht nur von der Benutztbeziehung zwischen Modulen, sondern auch von der Vererbungsbeziehung ab. Außerdem gewinnt der Integrationstest an Bedeutung, da die einzelnen Methoden/Operationen meist eine einfache Struktur haben und schlecht isoliert von der Aufrufumgebung anderer Methoden/Module getestet werden können (vgl. Kapitel 13.2 und 13.3).
Beim strikten **funktionalen** Programmieren entfällt das Konzept der Variablen und der Zuweisungen, also auch der Kontroll- und Datenfluß. Stattdessen gibt es

1. Argumente und Ergebnisse von Funktionen (anstelle von Eingabe- und Ausgabevariablen),
2. Schachtelung (Einsetzen) von Funktionen (anstelle von sequentieller Reihenfolge);
3. Fallunterscheidungen (anstelle von Verzweigungen des Kontrollflusses),
4. rekursive Funktionseinsetzungen (anstelle von Schleifen).

Die kontrollflußbezogenen Kriterien von Kapitel 7 sind also entsprechend anzupassen, d. h. die Entsprechungen *Funktionsschachtelung = Sequenz*, *Fallunterscheidung = Verzweigung*, *Rekursion = Schleife* sind zu verwenden. Das Kriterium *Zweigüberdeckung* wird also zum Kriterium *Fallunterscheidungsüberdeckung*, d. h. es muß soviel verschiedene Testdaten (Funktionsargumente) geben, daß jede Fallunterscheidung bei der Auswertung der Funktionsausdrücke[1] mindestens einmal vorkommt. Das Kriterium „jede Schleife genau 0-mal und mehr als einmal durchlaufen“ wird beispielsweise zum Kriterium „jede Rekursion 0-mal und mehr als einmal ausführen.“

Bei der **logischen** Programmierung werden **deklarative** Programme geschrieben: Es wird nicht mehr eine determinierte Folge von Zuweisungen (wie bei imperativen Programmen) oder von Termersetzungen (Funktionsschachtelung, Fallunterscheidung, Rekursion) vorgenommen. Stattdessen wird für die zu berechnenden Funktionen mit logischen Ausdrücken bzw. Prädikaten spezifiziert, welche Gleichungen sie erfüllen müssen bzw. welchen Regeln sie genügen müssen. Durch sogenannte **Unifikation** können solche Gleichungen gelöst werden. Auch in diesem Fall können einige Prüfkriterien entsprechend übertragen werden. Das Kriterium *Anweisungsüberdeckung* wird z. B. zum Kriterium *Gleichungsüberdeckung*, d. h. jede Gleichung sollte mindestens bei einer Unifikation verwendet werden.

Nicht zu unterschätzen ist auch der Einfluß der Datenstrukturen. Die Berechnung von Zahlwerten läßt sich einfach modellieren und daher auch testen, verifizieren und symbolisch ausführen; allerdings gibt es beim Typ „Real“ schon Probleme mit der Rechengenauigkeit. Die Berechnung **nichtnumerischer** Werte (Texte, Listen, Bäume oder Verbünde von Werten) ist dagegen komplexer zu modellieren und entsprechend schwieriger zu überprüfen.

[1] Auswertung durch „Termersetzung“

Bei allen diesen Varianten des Programmierparadigmas oder der verwendeten Datenstrukturen ist natürlich wieder zu untersuchen, welche neuen, anderen Fehlerarten dabei auftreten können und üblicherweise auftreten und mit welchen Prüf- und Testmethoden diese Fehler zu finden sind.

17.2.2 Erforschung, Entwicklung und Erprobung von neuen Prüf- und Testmethoden

Das spezifikations- und entwurfsorientierte Testen leidet darunter, daß meist eine formale Basis fehlt. Daher sind formale Spezifikations- und Entwurfsverfahren zu entwickeln bzw. für das Testen nutzbar zu machen, z. B. zum (möglichst) automatischen Generieren der Testfälle und Testdaten.

Für die implementationsorientierten Testmethoden sind ebenfalls praktikable Verfahren zur (möglichst) automatischen Testdatenerzeugung zu entwickeln, da das Hilfsmittel „symbolische Ausführung“ theoretische und praktische Probleme aufwirft (vgl. Kapitel 12.3). Denkbar wäre z. B. ein zufälliger Erzeugungsprozeß, der von „intelligenten“ Suchstrategien (z. B. genetischen Algorithmen) gesteuert wird.

Außerdem ist der Soll-Ist-Vergleich der Testergebnisse zu automatisieren oder zu unterstützen:

1. durch Ableitung der Solldaten aus der (formalen) Spezifikation oder
2. durch Einsetzen der Ein- und (Ist-)Ausgabedaten in die Gleichungen oder Relationen einer entsprechenden Spezifikation (vgl. Kapitel 5).

Das Kosten/Nutzen-Verhältnis der bisher vorliegenden Testmethoden und Prüfmethoden („Aufwand für die Durchführung“ zu „gefundene Fehler“) sollte durch empirische oder analytische Untersuchungen ermittelt werden (s. Teilergebnisse in den Kapiteln 6, 10, 11 und an verschiedenen Stellen in den Kapiteln 12 bis 14). Dies ist als Grundlage für eine geeignete Auswahl von Test- und Prüfmethoden zu verwenden. Der Aufwand für die Durchführung eines Tests hängt natürlich von der Anzahl der zu generierenden Testdaten ab, aber auch von dem Aufwand für die Generierung, Ausführung und Auswertung eines einzelnen Testdatums und des zugehörigen Testlaufs. Dabei spielt die Verfügbarkeit geeigneter Software-Werkzeuge, Hardware und Methoden eine entsprechende Rolle.
Die Software-Werkzeuge wurden in diesem Buch nicht vorgestellt, da dies den Rahmen sprengen würde, Werkzeuge zudem schneller veralten als unterliegende Methoden und meist programmiersprachenabhängig sind. Die Testwerkzeugentwicklung wird also zwangsweise immer der Programmiersprachenentwicklung „hinterherhinken“. Allerdings sollten Teile des Compilers (z. B. der Syntax-Parser) sofort für die notwendige statische Analyse zur Verfügung stehen.

Der Aufwand für die Testdurchführung läßt sich spürbar senken, wenn geeignete Hardware zur Verfügung steht. Der immense Zeitaufwand für die Mutationsanalyse läßt sich z. B. stark reduzieren, wenn ein Parallelrechner zur Verfügung steht, auf dem die einzelnen Mutanten parallel getestet werden können (vgl. Kapitel 9.6).
Mit der Mutationsanalyse und dem Konzept der fehleraufdeckenden Teilbereiche (s. Kapitel 9.3 und 16.3) wurden Methoden vorgestellt, die sich direkt an den Fehlern bzw. Abweichungen im Programm orientieren. Da diese Methoden (noch) nicht praktikabel sind, andere Methoden (aus diesem Buch) aber nicht genügend fehlerorientiert sind, muß weiterer Aufwand für die Erforschung und Entwicklung praktikabler und fehleraufdeckender Methoden und Werkzeuge getrieben werden. Es bleibt viel zu tun ...

17.3 Verwendete Quellen und weiterführende Literatur

Sneed gibt in [Sne 95] einen Überblick über das **objektorientierte Testen**. Genaueres zum Vorgehen und zu den Problemen bei *nicht strikter* oder *Mehrfach*-Vererbung findet man in [FLS 91a], Kap. 4.1. Eine detaillierte Fehlertaxonomie und daran orientierte Methoden für den Modul- und Integrationstest objektorientierter Systeme präsentiert Overbeck in [Ove 93]. Ein Kapitel in [BeG 96] behandelt den Test für logische Programmiersprachen. Genaueres zum Prüfen **logischer Programme**, insbesondere von Prolog-Programmen, findet man in [FLS 91a], Kap. 4.2. Die Erzeugung von Testdaten durch **genetische Algorithmen** behandelt Jones (s. [Jon 95]). Die Idee der Automatisierung des **Soll-Ist-Vergleichs** durch Einsetzen der Testdaten in die Spezifikation wird von Hörcher beschrieben (s. [Hör 95]).

Anstelle eines Nachwortes:

Ein Buch hat oft auf eine ganze Lebenszeit
einen Menschen gebildet oder verdorben.

— **Herder**