Teil I

Einführung

1 Beispiele für den Programmtest

„Lang ist der Weg durch das Lehren,
kurz und erfolgreich durch Beispiele.“
— **Seneca**

Dieses Buch beschäftigt sich mit den Problemen und Lösungen, die beim Testen im weiteren Sinne auftreten, d. h. mit statischen und dynamischen Überprüfungsverfahren für Module und ganze Systeme.

Bevor die verschiedenen Verfahren und Überlegungen präsentiert werden, soll Ihnen die Gelegenheit gegeben werden, anhand eines kleinen Beispiels Ihre Testfähigkeit zu erproben. Sie sollten diesen Selbsttest unbedingt durchführen (s. Kapitel 1.1).

Die in diesem Buch vorgestellten Testverfahren sollen — wenn möglich — an *einem* Beispiel demonstriert werden (s. Kapitel 1.2), damit nicht für jedes Testverfahren stets neue Programmspezifikationen oder -implementationen vorgestellt werden müssen und sich die Testverfahren gut vergleichen lassen. Allerdings dürfen keine voreiligen Schlüsse gezogen werden: Testverfahren, die bei einem Beispiel besonders gut (oder schlecht) abschneiden, müssen nicht generell gut (oder schlecht) sein. Daher — und um die Komplexität der Beispiele zu begrenzen — werden oft noch weitere Beispiele vorgestellt, bei denen Stärken oder Schwächen der Testverfahren deutlich werden. Insbesondere für nebenläufige Programme ist ein anderes Standardbeispiel notwendig, da diese Klasse von Programmen andere Eigenschaften als sequentielle Programme hat (s. Kapitel 1.3).

1.1 Selbsttest

Stellen Sie sich vor, daß für das zu testende Programm folgende informelle Spezifikation vorliegt:

> „Das Programm liest drei ganzzahlige Werte ein, die als Längen der Seiten eines Dreiecks interpretiert werden. Das Programm druckt aus, welche Eigenschaft das Dreieck hat: spitzwinklig, stumpfwinklig, rechtwinklig, gleichschenklig oder gleichseitig.“

Der Quelltext des Programms liegt noch nicht vor, aber dennoch sollten Sie sich schon Gedanken über das Testen machen.

Testaufgabe 1:
Gibt es Unklarheiten in der Spezifikation, die noch mit dem Auftraggeber geklärt werden müßten? Wenn ja, welche? Präzisieren Sie die Spezifikation in geeigneter Weise! (Um Sie nicht in Versuchung zu führen, finden Sie eine mögliche Lösung zu dieser und weiteren Testfragen nicht auf der jeweils nächsten Seite, sondern erst im Anhang A.1.)

Nachdem die Programmspezifikation eindeutig ist, können Sie sich nun dem eigentlichen Testproblem zuwenden.

Testaufgabe 2:
Geben Sie eine Reihe von Testdaten an, die Ihrer Meinung nach das Programm ausreichend testen!

Wenn Sie nicht alle im Anhang A.2 aufgeführten Testdaten notiert haben, trösten Sie sich: auch erfahrene, professionelle Programmierer denken i. allg. nur an die Hälfte aller Fälle[1].

Die im Anhang A.2 aufgeführten Testdaten 1 bis 38 bilden einen sehr gründlichen Test für das Programm. Dennoch würde ein fehlerfreier Test nicht unbedingt die Fehlerfreiheit des Programms gewährleisten. Dazu müßte man etwas über die Programmrealisierung wissen, z. B. ob die spezifizierten Fälle und ihre Grenzwerte im Programm entsprechend behandelt werden; falls nicht, sind weitere Tests notwendig.

Man kann vermuten, daß eine vernünftige Programmrealisierung ein einfaches Programm ergibt[2]. Trotzdem werden für einen „ausreichenden“ Test schon sehr viele Testdaten gebraucht. Wie können aber große Programmsysteme mit Zehntausenden, Hunderttausenden oder sogar Millionen von Anweisungen „ausreichend“ getestet werden? Wie kann die Mängel- und Fehlerfreiheit von Programmsystemen überprüft werden, für die keine mathematisch exakte Spezifikation (wie bei dem hier vorgestellten Dreiecksprogramm) vorliegt, sondern nur vage Anforderungen, organisatorische Vorgaben, Erfahrungswissen, betriebswirtschaftliche oder juristische Regeln? Was bedeutet es eigentlich, ein Programmsystem „ausreichend“ zu testen oder zu überprüfen? Was bedeutet „Mängelfreiheit“ und „Fehlerfreiheit“ von Programmsystemen?

Die weiteren Kapitel dieses Buches geben auf diese und weitere Fragen eine Antwort, weisen aber auch auf die ungelösten Probleme hin.

---

[1] Dies ist jedenfalls die Aussage in Kapitel 1 des Buches von Myers (vgl. [Mye 79]). Dort werden 13 Testdaten für ein ähnliches Dreiecksprogramm angegeben, bei dem allerdings die Fälle *rechtwinklig*, *spitzwinklig* und *stumpfwinklig* nicht vorgesehen sind und kein Dialogprogramm vorgeschaltet wird.

[2] Weyuker/Ostrand stellen ein Programm vor, welches die in Anhang A.1 angegebene Spezifikation erfüllt (vgl. [WeO 80] und Übung 16.2).

1.2 Beispiel für ein sequentielles Programm

Für das Beispiel liegt folgende Spezifikation vor:

> „Ein einfacher Textformatierer hat die Aufgabe, einen Text, bestehend aus Wörtern, die durch Blank (BL) oder Newline (NL) getrennt sind, so in Zeilen zu formatieren, daß
>
> (a) keine Zeile mehr als MAXPOS Zeichen hat,
>
> (b) Zeilenumbrüche nur an BL- oder NL-Zeichen auftreten,
>
> (c) die Anzahl der Wörter je Zeile maximal ist.

Obige Spezifikation ist wieder ein „gutes" Beispiel für eine unvollständige und evtl. nicht korrekte Spezifikation — gemessen an den eigentlichen Anforderungen.

Testaufgabe 3:
Geben Sie eine vollständige, präzise Spezifikation für obigen Textformatierer an!

Eine mögliche Lösung dazu finden Sie im Anhang A.3. Sie dient in den folgenden Kapiteln als Vorgabe für Realisierungen. Sie sollten sich die Lösung erst ansehen, wenn Sie selbst zu einem Urteil gekommen sind.
Mit den ursprünglichen Anforderungen (a), (b) und (c) von oben und den zusätzlichen sechs Forderungen 1 bis 6 aus dem Anhang A.3 ist das Programmverhalten präzise und vollständig beschrieben. Allerdings gibt es sicher noch Wünsche und Anforderungen, die damit nicht erfüllt sind.

Testaufgabe 4:
Geben Sie wünschenswerte Änderungen oder Ergänzungen zu der bisherigen Spezifikation des Textformatierers an!

1.3 Beispiel für ein nebenläufiges Programm

Nebenläufige Programme haben besondere Eigenschaften und Fehler, die bei sequentiellen Programmen nicht auftreten können, z. B. die Blockierung von Teilprogrammen. Daher wird hier ein Beispielprogramm angegeben, daß einige dieser besonderen Probleme darstellt.

Bei der „klassischen" Version des Problems der **essenden Philosophen (dining philosophers)** gibt es fünf Philosophen, die einen eintönigen Lebensrhythmus haben, der nur aus einem Wechsel von *denken* und *essen* besteht. Das Essen geschieht an einem runden Tisch mit fünf Plätzen. In der Tischmitte steht eine Schüssel mit Spaghetti, die aber schwierig zu essen sind und daher zwei Gabeln erfordern. Es gibt zwischen zwei Plätzen nur je eine Gabel, also insgesamt nur fünf Gabeln (siehe Abbildung 1.1; Achtung: Fehler im Bild: Die Gabeln müssen *zwischen* den Philosoghen

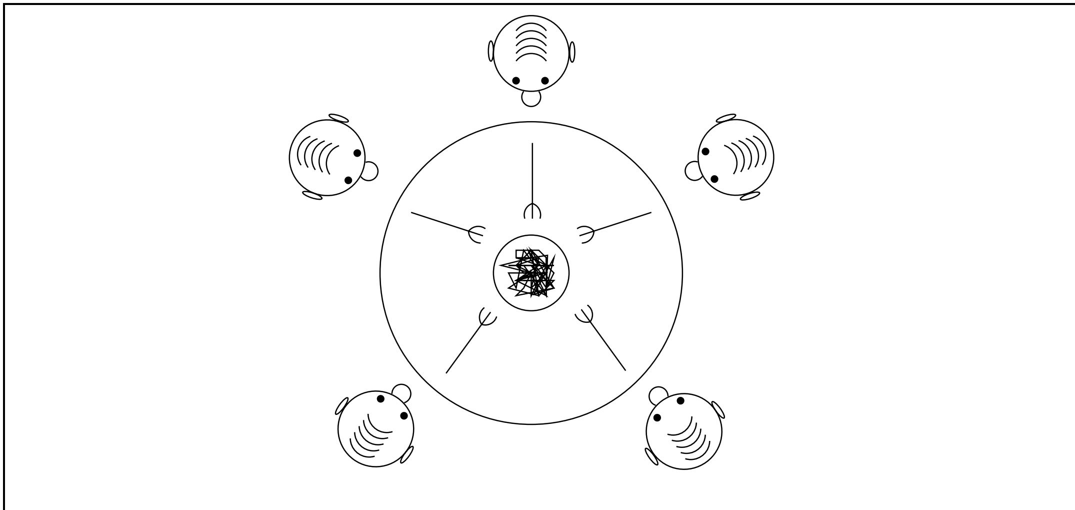

Abb. 1.1: Das „dining philosophers“-Problem

liegen!!). Ein Philosoph kann demnach nur essen, wenn die beiden benachbarten Philosophen keine Gabeln zum Essen aufgenommen haben.

Man könnte nun eine geeignete Verhaltensstrategie ermitteln, bei der kein Philosoph beliebig lange auf das Essen warten muß (und somit verhungert). Stattdessen wird im folgenden von einer gegebenen Strategie ausgegangen. Diese Strategie ist sehr einfach und lautet:

> Jeder Philosoph hat einen festen Platz. Wenn er hungrig ist und essen will, nimmt er zuerst die linke Gabel auf und dann die rechte. Nach dem Essen legt er zuerst die linke Gabel und dann die rechte Gabel zurück auf den Tisch. Falls eine Gabel nicht verfügbar ist, wartet der Philosoph so lange, bis die Gabel verfügbar ist.

In Kapitel 14 (S. 396) wird z. B. überprüft (durch statische Analyse), ob bei der gegebenen Strategie ein Verhungern eines Philosophen möglich ist oder nicht.

2 Grundlegende Problemstellungen und Lösungsansätze

Dieses Kapitel beschäftigt sich damit, welche Ursachen und Auswirkungen fehlerhafte oder unangemessene Software hat, welche Arten von Fehlern entstehen und wie sie vermieden oder beseitigt werden können. Außerdem wird analysiert, welche Rolle dabei das Testen von Software spielt und welche Anforderungen daraus an das Verantwortungsbewußtsein von Testpersonen abgeleitet werden müssen.
Als roter Faden zieht sich dabei das folgende Begriffstripel durch dieses Kapitel: Error — Fault — Failure[1], zu deutsch: Irrtum – Fehler – Fehlverhalten[2].

2.1 Ursachen unangemessener und fehlerhafter Software

„Der Mensch ist für den Irrtum geschaffen.
Nur durch ungeheure Anstrengung
entdeckt er einige Wahrheiten."
— **Friedrich der Große**

Dieses Unterkapitel beschäftigt sich mit Irrtümern und menschlichen Fehlleistungen beim Austauschen, Verarbeiten und Speichern der Informationen, die für die Entwicklung von Software notwendig sind. Zuerst werden die Probleme beim Austauschen und Verarbeiten von Informationen, bei der sogenannten **Kommunikation**, betrachtet. Die Sozialwissenschaften und die Psychologie unterscheiden drei Hauptformen der Kommunikation: intrapersonale, interpersonale und mediengebundene Kommunikation (s. [Mey 75], Bd. 14, S. 91 f.).
Unter **intrapersonaler** Kommunikation wird die Informationsverarbeitung innerhalb eines Menschen verstanden, also das Denken und Wahrnehmen. Der Informationsaustausch zwischen zwei (Gesprächs-)Partnerinnen[3] wird als **interpersonale**

[1] siehe dazu [BaS 87], S. 347 f. und [TLN 78], S. 3: „Error is a causative act producing a fault in a software, which, if evoked, would result in a symptomatic execution failure."

[2] Die Wörter Irrtum und Fehler sind sprachgeschichtlich verwandt: „fehlen" leitet sich ab von englisch „to fail" und (alt)französisch „fa(il)lir" ([mit der Lanze] verfehlen, sich irren). Seit dem 16. Jh. gibt es das Wort Fehler mit der Bedeutung Fehlschuß (Gegensatz: Treffer), erst ab dem 18. Jh. bedeutet es Versehen (Schreib- und Rechenfehler) bzw. bleibender Mangel.

[3] Wie im Vorwort angekündigt, wird in diesem Kapitel bei Personen- und Berufsbezeichnungen nur die weibliche (oder eine wirklich neutrale) Form benutzt.

Kommunikation bezeichnet. Unter **mediengebundener** Kommunikation wird der Austausch zwischen der meistens kleinen Gruppe von Kommunikatoren (z. B. Journalistinnen) und der häufig umfangreichen Gruppe von Rezipienten (z. B. Leserinnen einer Zeitung) verstanden. Bei der Entwicklung von Software treten alle drei Hauptformen der Kommunikation auf. Die in letzter Zeit zunehmende elektronische Versendung von Nachrichten mittels Verteilerlisten ist ein Grenzfall zwischen der zweiten und dritten Hauptform der Kommunikation, die Nutzung von Online-Hilfen, Handbüchern und News-Gruppen ein Beispiel für mediengebundene Kommunikation.
Die Information durchläuft bei der menschlichen und technischen Kommunikation jeweils mindestens drei Stufen (s. Abb. 2.1):

1. Verschlüsselung (Kodierung) der Information durch die absendende Person („Sprecherin"),

2. Übermittlung durch das Medium,

3. Entschlüsselung (Dekodierung und Interpretation) durch die empfangende Person („Hörerin").

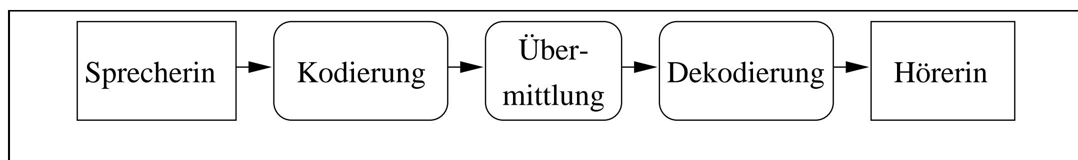

Abb. 2.1: Verarbeitungsstufen für die Information bei menschlicher Kommunikation

Bei jeder Kommunikationsform und auf jeder Stufe können Fehler gemacht werden. Bei der intrapersonalen Kommunikation kann etwa durch Ermüdung, Streß, mangelhafte Motivation oder Information eine Fehlleistung erfolgen.

Beispiel 2.1.1
Eine Programmiererin kann statt der Variablen `BRUTTO` *der Variablen* `NETTO` *einen Wert zuweisen, was als* **Identitätsirrtum** *bezeichnet wird.*

Beim interpersonalen Gespräch oder Schriftwechsel können folgende Fehler bzw. Irrtümer auftreten:

- Beim Verschlüsseln der Information kann ein **Erklärungsirrtum** auftreten, d. h., es wird nicht das gesagt oder geschrieben was eigentlich gemeint ist.

- Durch einen **Übermittlungsirrtum**, z. B. vom Sekretariat, wird die Information inkorrekt übermittelt.

- Durch einen **Entschlüsselungsirrtum** wird entweder (a) die Information vordergründig falsch gehört oder gelesen oder (b) den Signalen eine andere Bedeutung zugeordnet als von der absendenden Person gemeint. Das kann daran liegen, daß die Gesprächspartnerinnen nicht die gleiche (Fach-)Sprache sprechen.

  Beispiel 2.1.2
  Eine EDV-Spezialistin als Systemanalytikerin und eine Sachbearbeiterin in einer Bank können aneinander vorbeireden, weil sie jeweils Fachausdrücke verwenden, die von der Gegenseite falsch oder gar nicht verstanden werden.

Jede an der Kommunikation beteiligte Person braucht also ein gewisses Verständnis und Wissen von den Kenntnissen und Aufgaben der anderen Person[4]. Dazu kommt noch die Kenntnis und Berücksichtigung des situativen Rahmens.

Beispiel 2.1.3
Wenn als Ziel einer Systemanalyse eine Verschärfung der Arbeitsbedingungen oder sogar ein Wegfall der Arbeitsplätze vorgesehen ist, werden die Interviewpartnerinnen einer Systemanalytikerin nur unwillig antworten. Vermutlich werden sie die bestehenden Regelungen am Arbeitsplatz nur positiv beschreiben und keine Mängel zugeben. Die angespannte Situation bei einer Systemanalyse kann also die notwendige Kommunikation blockieren (siehe dazu auch [Mol 85]).

Durch fehlerhafte, aber auch durch fehlende Kommunikation können Mißverständnisse zwischen Absender und Empfänger auftreten. Das kann einen Irrtum über qualitative oder quantitative Eigenschaften der Software bewirken, was als **Inhaltsirrtum**[5] bezeichnet wird.

Beispiel 2.1.4
Der Versuch, den 150 Millionen Dollar teuren Satelliten „Intelsat 6" von einer Rakete vom Typ Titan 3 auszusetzen, ist wegen einer falschen Absprache bei der Entwicklung gescheitert. Die Rakete kann zwei Satelliten transportieren, hatte aber nur einen in der oberen Position geladen. Der Befehl, den ersten Satelliten auszusetzen, wurde nicht ausgeführt, weil die Computerfachleute damit den oberen Satelliten, die Verkabelungsfachleute aber den unteren (nicht vorhandenen) Satelliten meinten (s. SEN, Vol. 15, No. 3, Juli 1990, S. 14 f.).

Die aufgezeigten Kommunikationsprobleme können nicht nur zwischen den Personen auftreten, die die Software entwickeln und anwenden, sondern auch zwischen

[4] „Denn immer dann, wenn Software nicht von ein und denselben Personen entwickelt und genutzt wird, sind Interpretationsakte notwendig, die sich einerseits auf die erstellte Software-Dokumentation und andererseits auf die im Zuge der Programmausführung gewonnenen Daten beziehen." (s. [Luf 88], S. 145)

[5] Identitäts-, Erklärungs- und Inhaltsirrtum berechtigen übrigens nach §§118 und 119 BGB zur Anfechtung von Willenserklärungen — wie etwa dem Inhalt des Pflichtenhefts.

den einzelnen Personen eines Entwicklungsteams. Diese Kommunikationsprobleme verhindern die angemessene softwaretechnische Umsetzung von ursprünglichen Anforderungen und von späteren Anforderungen während der Wartungsphase.

Im folgenden werden die Probleme beim Speichern von Informationen, die **Gedächtnisprobleme**, die zu Fehlern in Softwaresystemen führen, betrachtet.

Beim konventionellen Programmieren wird versucht, „wie ein Computer zu denken". Bei komplexen Problemen erfordert dies nach David L. Parnas folgendes Vorgehen, das sich am Kontrollfluß und Datenfluß des Programms orientiert:

> „ ... Wenn die Reaktion des Computers von Bedingungen abhängt, die erst zur Laufzeit bekannt sind, ... müssen wir eine oder mehrere Aktionen markieren und uns merken, wie man dorthin gelangt. ... Wenn wir Schleifen in das Programm einführen, gibt es viele Wege, um zu einigen dieser Stellen zu gelangen, und wir müssen uns alle diese Wege merken. Wenn wir weiter durch den Algorithmus vordringen, erkennen wir den Bedarf an Information über frühere Ereignisse und erweitern (daher) unsere Datenstruktur um weitere Variablen. Wir müssen uns nun merken, was die Daten bedeuten und unter welchen Bedingungen die Daten relevant sind. ... Die Menge dessen, was wir uns merken müssen, wächst und wächst. Die einfachen Regeln, die definieren, wie wir zu bestimmten Stellen gelangen, werden komplexer. ... Die einfachen Regeln, die definieren, was die Daten bedeuten, werden komplexer, wenn wir andere Verwendungen für existierende Daten finden und neue Variablen hinzufügen. Schließlich machen wir einen Fehler." (Siehe [Par 85], zitiert nach [Mye 86], S. 63.)

Eine weitere Ursache für Komplikationen bei der Entwicklung von Software ist die **Komplexität** von Systemen mit Nebenläufigkeit (concurrency) und vielen unabhängigen Prozessen (multiprocessing). „Das Schreiben und Verstehen von sehr großen Echtzeitprogrammen", aber auch von komplexen integrierten oder verteilten Programmsystemen „durch *Denken wie ein Computer* ist daher außerhalb unserer intellektuellen Möglichkeiten." (Siehe [Par 85], zitiert nach [Mye 86], S. 64.)

Gestörte Informationsverarbeitung ist nicht die einzige Fehlerursache bei der Entwicklung von Software. Fehler in Programmen werden auch durch mangelndes **Problemverständnis** erzeugt. Gemeint sind hier Mängel oder „Unklarheit(en) in den Anwenderwissenschaften, die beim Versuch der Software-Entwicklung erst zutage treten, in Wirklichkeit aber schon vorher existierten" (s. [Lan 87], S. 280). Ein Beispiel für solche Mängel ist die Schachprogrammierung, bei der eine Theorie des strategischen Vorgehens in der mittleren Phase des Spiels fehlt. In diesem Fall kennt niemand die algorithmische Lösung. In anderen Fällen gibt es Lösungen, aber den Entwicklerinnen der Software sind sie nicht bekannt.

2.2 Manifestation von Softwarefehlern

„Erfahrungen sind Erinnerungen,
die man teuer bezahlen mußte.“
— **Werner Krauss**

Kapitel 2.1 hat die Ursachen für unangemessene und fehlerhafte Software aufgezeigt. Jetzt sollen die Auswirkungen des mangelnden Problemverständnisses, der Komplexität der Systeme, der menschlichen Irrtümer und Gedächtnisprobleme betrachtet werden. Es geht dabei um folgende Fragestellungen:

- Wie viele Fehler treten auf?
- Welche Fehler treten auf?
- Wann und weshalb werden die Fehler gemacht bzw. entdeckt?

Abschließend wird der Begriff des Fehlers problematisiert und vorläufig geklärt.

Die Zahl der erzeugten Fehler in Softwaresystemen ist erheblich. Diverse Studien ermitteln eine Fehlerrate von 30 bis 100 pro 1000 Anweisungen[6] im Quellcode (s. [Boe 81], S. 383, zitiert nach [Mye 86], S. 62, und [RDB 75]). Dies bezieht sich auf die während der Entwicklung gemachten Fehler.
Die Fehlerarten wurden in umfangreichen Untersuchungen (z. B. von Basili/Perricone, Chillarege, Endres, Glass, Rubey et al. und Thayer et al.) und in kleineren bzw. speziellen Untersuchungen (z. B. von Xu und Youngs) ermittelt (s. [BaP 84], [Ch& 92], [End 75], [Gla 81], [RDB 75], [TLN 78], [Xu 85], [You 74]).

In der Firma „TRW Systems und Energy“ in Kalifornien wurden z. B. vier Projekte mit Schwerpunkt Echtzeitprogrammierung untersucht (s. [TLN 78]). In den Projekten wurden je 173 bis 531 Routinen mit insgesamt 11.000 bis 115.000 Zeilen (Lines of Code) in den Sprachen JOVIAL, PWS und FORTRAN geschrieben. Zur Klassifizierung der Fehlerarten wurden 20 Fehlerkategorien mit insgesamt 164 Unterkategorien benutzt. Die häufigsten Kategorien, bei denen Fehler auftraten, sind in Abbildung 2.2 dargestellt.

Die Einzelergebnisse der oben genannten Untersuchungen sind nicht direkt vergleichbar, da man in der Praxis die aufgedeckten Fehler nach verschiedenen Firmenstandards klassifiziert und in sogenannten Problemberichten (problem reports) dokumentiert hat. Werden die Ergebnisse der Fehleruntersuchungen dennoch zusammengefaßt, ergeben sich die folgenden Aussagen über die häufigsten Fehlerarten, die bei den Prozentangaben eine große Schwankungsbreite haben: Datenbehandlung und

[6] Der genaue Wert der Fehlerrate hängt von der Größe des Systems, der Art der Software, der Definition des Begriffs „Fehler“ (genaueres s. S. 31), der Art der Fehlererfassung und anderen Faktoren ab.

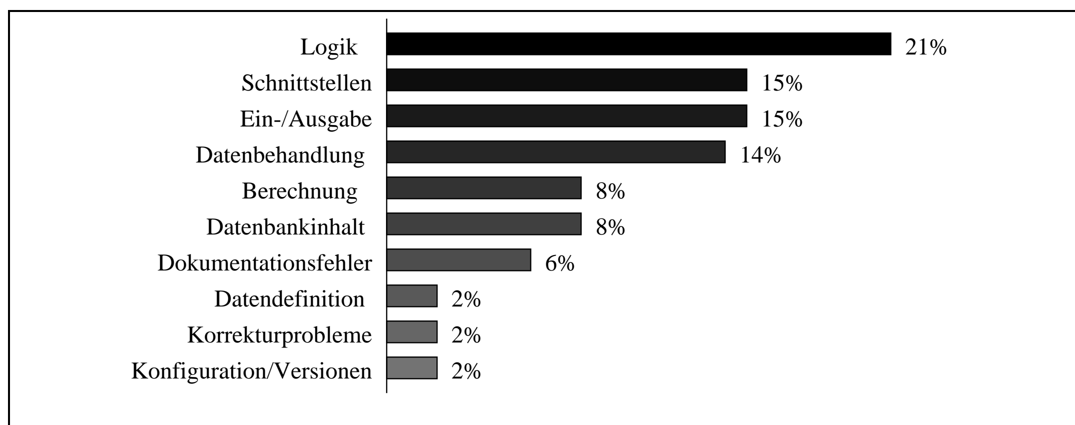

Abb. 2.2: Softwarefehler und ihre Manifestation in Projekten der Fa. TRW

-zugriff (10 bis 21%), Logik (10 bis 30%), Schnittstellen (7 bis 39%), Berechnungen (8 bis 41%), Datendefinitionen und -deklarationen (2 bis 17%), Ein-/Ausgabe (3 bis 15%), Dokumentation (6 bis 14%), Korrekturprobleme (2 bis 6%), sowie Spezifikation (6 bis 40%).
Die hier benutzte Klassifizierung ist leider nicht firmenübergreifend standardisiert, obwohl es erste Versuche dazu gibt[7]. Sinnvoll scheinen dagegen die sechs folgenden verschiedenen Klassifizierungen[8]:

1. Ursachen der Fehler,
2. Phase der Fehlerentstehung,
3. Phase der Fehlerentdeckung,
4. Fehlerart „Auslassung“ oder „(In-)Korrektheit“ (omission/commission),
5. Fehlerarten nach Programmkonstrukten klassifiziert (s. [BaR 87]),
6. Verteilung der Fehler auf die einzelnen Module.

Bei Klassifizierung 1 können Mißverständnisse oder die fehlerhafte Verwendung von Methoden oder Konstrukten in den folgenden Bereichen Fehler verursachen:

[7] IEEE Standard P-1044/D3 „A Standard Classification of Software Errors, Faults and Failures“, Techn. Commitee on Software Engineering, unapproved draft, Dez. 1987 (zitiert nach [Ch& 92], S. 944 und 955)

[8] Ob eine Klassifizierung/Ermittlung der Verteilung der Fehler auf einzelne Programmiererinnen als sinnvoll anzusehen ist, hängt nicht nur von der Interessenlage ab (Arbeitgeber/Arbeitnehmer), sondern auch vom Effekt solcher Ermittlungen auf die notwendige Kommunikation zwischen den beteiligten Personen (vgl. Fußnote 12 in Abschnitt 12.1.3).

- Mißverständnisse im Anwendungs- oder Problembereich,
- fehlerhafte Verwendung von semantischen oder syntaktischen Regeln der benutzten Sprache,
- fehlerhafte Verwendung der Hardware- und Softwareumgebung des geplanten Systems,
- fehlerhaftes Behandeln von Informationen, z. B. fehlerhafte Dokumentation,
- fehlerhafte mechanische Transkription von Information, z. B. Tippfehler beim Wechsel des Darstellungsmediums oder des Darstellungsformats.

Für eine konstruktive Qualitätssicherung ist es wichtig, die Fehlerursachen zu erkennen und für die Zukunft möglichst zu beseitigen.

Bei den Klassifizierungen nach der Phase der Fehlerentstehung (2) oder Fehlerentdeckung (3) ist zu beachten, daß Fehler in jeder Phase der Entwicklung von Software — von der Anforderungsanalyse bis zur Inbetriebnahme und Pflege — auftreten können. In der Studie von Thayer et al. wird z. B. für jede Fehlerart untersucht, ob die Fehler überwiegend im Entwurf oder bei der Codierung gemacht werden. Insgesamt ergibt sich, daß 62% der Fehler beim Entwurf und 38% der Fehler bei der Codierung entstehen (s. [TLN 78]). Die Gültigkeit dieses Ergebnisses setzt natürlich voraus, daß bei der Erforschung der Fehlerursachen immer eine eindeutige „Schuldzuweisung“ erfolgen kann. Wenn ein Komponentenentwurf beim Codieren falsch umgesetzt wird, kann dies aber mehrere Ursachen haben:

- der Entwurf ist unvollständig,
- der Entwurf ist ungenau,
- der Entwurf wird falsch interpretiert,
- es wird falsch codiert, obwohl der Entwurf richtig verstanden wurde.

Bei den ersten drei Ursachen ist die Schuldzuweisung aber unklar, da ein Kommunikationsproblem vorliegt. Daher wird nur die Phase der Fehlerentdeckung zweifelsfrei anzugeben sein.

Bei der Klassifizierung nach Fehlerarten (4) läßt sich ein Fehler nicht immer eindeutig an Konstrukten der Programmiersprache (5) festmachen. Daher muß eine Klassifizierung versucht werden, die für alle Programmarten und alle Phasen der Softwareentwicklung gültig ist. Sinnvoll scheint folgende Einteilung, wobei zusätzlich danach unterteilt wird, ob Angaben fehlen oder (vorhanden aber) falsch sind (s. [Ch& 92]):

1. Funktionsfehler:
   Ein Teil der Programmfunktionalität, der Benutzungs- oder Hardware-Schnittstellen oder der globalen Datenstrukturen fehlt, ist unvollständig oder falsch. Daher ist eine Entwurfsänderung notwendig.

2. Schnittstellenfehler:
   Die Interaktion zwischen den Programmkomponenten ist fehlerhaft, z. B. sind die Parameterlisten unvollständig oder falsch.

3. Algorithmusfehler:
   Algorithmusfehler beinhalten Effizienz- und Korrektheitsprobleme in einem lokalen Algorithmus, die ohne Entwurfsänderungen behoben werden können, z. B. durch eine andere Implementierung der Kontrollstruktur oder der lokalen Datenstruktur.

4. Zuweisungsfehler:
   Dies sind kleine Berechnungsfehler bei Initialisierungen oder Zuweisungen.

5. Abfragefehler:
   Durch fehlerhafte oder fehlende Abfragen ist die Programmlogik falsch.

6. Synchronisations- und Zeitfehler:
   Der Zugriff auf gemeinsame Ressourcen oder Echtzeit-Ressourcen ist fehlerhaft.

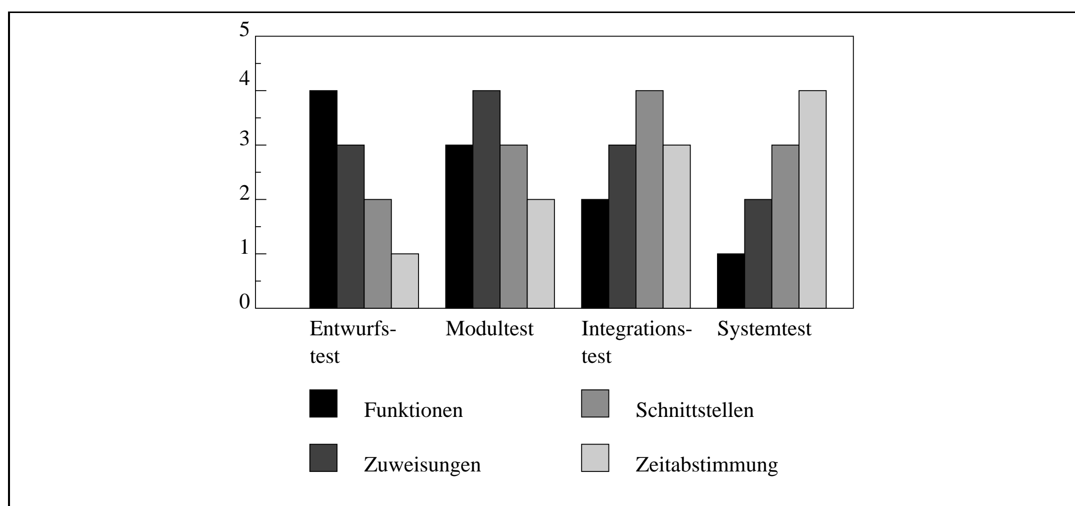

Abb. 2.3: Fehlerquellen und die relative Häufigkeit ihrer Entdeckung in verschiedenen Testphasen (nach [Ch& 92], S. 947)

7. Konfigurationsfehler:
   Bei größeren Systemen können Fehler bei der Verwendung von Bibliotheken, der Versionskontrolle oder dem Änderungsmanagement auftreten.

8. Dokumentationsfehler:
   Die Dokumentation des Programmsystems weicht von der (korrekten) Realisierung ab.

Die genannten Fehlerarten kommen zwar in allen Phasen der Entwicklung vor, es liegt aber keine Gleichverteilung vor. Für Funktions-, Schnittstellen-, Zuweisungs- und Synchronisations-/Zeitfehler gilt z. B. der in Abb. 2.3 dargestellte Sachverhalt.

Für die Ursachenforschung sind die Fehlerarten zusammen mit der Entdeckungsphase ein wichtiges Hilfsmittel. Für die Verbesserung des Prozesses der Fehleraufdeckung ist aber zusätzlich eine Klassifikation nach **Fehlerauslösern** (triggers) notwendig. Bei einer Inspektion des Grobentwurfs (high level design) wird die Fehlererkennung etwa durch folgende Sichtweisen bei der Prüfung durch die Inspektionsgruppe ausgelöst:

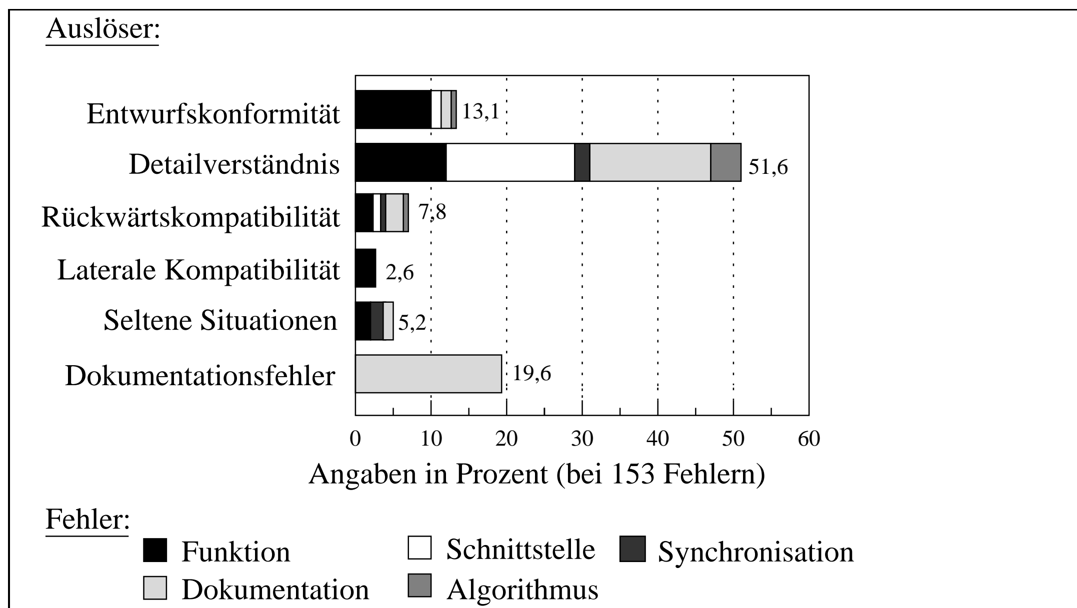

Abb. 2.4: Fehlerauslöser bei Entwurfsüberprüfungen (s. [Ch& 92], S. 953)

- Konformität mit den Anforderungen (Entwurfskonformität)
- Detailverständnis (Bedeutung von Operationen, Seiteneffekten, Nebenläufigkeit)
- Kompatibilität mit anderen Produkten (Rückwärtskompatibilität) [9]

[9] **Kompatibilität** bezeichnet die Verträglichkeit mit anderen Produkten der Entwicklung. Ein Teilprodukt (z. B. ein implementiertes Modul) ist **rückwärtskompatibel** (backward compatible), wenn es mit seiner Spezifikation und seinem Entwurf übereinstimmt. Das Modul ist **lateral** („seitlich") **kompatibel**, wenn es die Annahmen, die andere Module über seine Schnittstelle und sein Verhalten machen, erfüllt. Ein Sortier-Modul muß z. B. aufsteigend sortieren, wenn die aufrufenden Module dies erwarten.

- seltene Fälle und Situationen
- Inkonsistenzen und Unvollständigkeiten in der Dokumentation

Bei einer Untersuchung von 153 Fehlern hat sich z. B. der in Abbildung 2.4 dargestellte Zusammenhang zu den Fehlerarten ergeben.

Im Vergleich zu Standardverteilungen ist hier z. B. verdächtig, daß die Prüfung auf laterale Kompatibilität nur Funktionsfehler und keine Schnittstellenfehler aufgedeckt hat. Dies lag vermutlich an fehlenden Fähigkeiten des Inspektionsteams und nicht an der Verwendung einer Programmiersprache samt Compiler, die Schnittstellenfehler besser aufdecken (etwa PASCAL statt C).

Die Verteilung der Fehler auf die einzelnen Module (Klassifizierung 6) wurde z. B. von Endres untersucht (s. [End 75]). Dabei zeigt sich eine starke Häufung von Fehlern: in nur 21% der Module sind 78% der Fehler enthalten, während 26% der Module nur jeweils einen Fehler und 48% der Module keine Fehler enthalten.

Die Ergebnisse der Fehlerstudien können nicht repräsentativ sein, da die untersuchten Programme nicht repräsentativ sind und uneinheitliche Fehlerklassifizierungen benutzt wurden. Dennoch können die Fehlerstudien als Grundlage für angemessene Test- und Prüfmethoden verwendet werden: Die prozentualen Angaben zu den auftretenden Fehlern (s. oben bei Abb. 2.2) zeigen, daß bei der Testdatenerzeugung neben der klassischen Orientierung am Kontrollfluß, womit Logikfehler aufgedeckt werden können, eine Orientierung an der Spezifikation und an der Datenbehandlung und -definition nötig ist. Entsprechende Methoden werden in den Kapiteln 4, 5 und 9 vorgestellt. Schnittstellen- und Ein-/Ausgabefehler können durch eine statische Analyse oder durch spezielle Integrationstestmethoden erkannt werden (genaueres siehe Kapitel 12.2 und 13). Die Ergebnisse über die Häufung von Fehlern in den frühen Phasen der Entwicklung und in bestimmten Modulen ist Anlaß für eine entsprechende Konzentration des Testaufwands auf diese Phasen und Module, wobei das Testen in frühen Phasen meist nur durch manuelle Inspektion möglich ist (genaueres siehe Kapitel 12.1).

Eine andere Sichtweise ist erforderlich, wenn nicht nur der Aspekt der Funktion, sondern auch die Kriterien Leistung und Umfeld der Software als Orientierung dienen sollen. Ein Schachprogramm kann beispielsweise korrekt bzgl. der gegebenen Spezifikation sein; dennoch kann mangelnde Leistung als Fehler konstatiert werden, wenn das Schachprogramm den optimalen Zug nicht innerhalb von durchschnittlich vier Minuten berechnen und daher nicht gegen internationale Großmeister gewinnen kann. Das Umfeld der Software wirft besondere Klassifizierungsprobleme auf. Wie soll etwa entschieden werden, ob bei einem aufgetretenen Fehlverhalten ein Programmfehler, ein Bedienungsfehler oder nur ein Fehler in der Benutzerdokumentation der Software vorliegt. Ist es ein Fehler, wenn das Programm oder die Dokumentation nicht benutzungsfreundlich ist? Eine vorläufige Antwort auf diese Fragen kann durch folgende Festlegungen für Softwareprobleme erreicht werden:

- Die Nichterfüllung einer festgelegten Forderung, insbesondere eines Qualitäts- und Zuverlässigkeitsmerkmals, gilt als **Fehler** (vgl. DIN [EN] ISO 8402).
  Ob ein Leistungsmangel ein Fehler ist, hängt also davon ab, ob die erforderliche Leistung in der Anforderungsdefinition exakt (oder mit einem Toleranzintervall) festgelegt ist.

- Die Nichterfüllung einer beabsichtigten Forderung oder einer angemessenen Erwartung für den Gebrauch eines Softwareprodukts gilt als **Mangel**. Dazu gehören auch Forderungen bzw. Erwartungen an die Sicherheit der Software (vgl. DIN [EN] ISO 8402).
  In diesem Falle entspricht die Anforderungsdefinition also nicht den Forderungen bzw. Erwartungen der Auftraggeber. Ein Beispiel dafür ist die Forderung nach „Benutzungsfreundlichkeit", hinter der meist viele nicht genau spezifizierte Erwartungen stecken.

Dieses Buch konzentriert sich auf Methoden zum Aufdecken von Fehlern in der Funktionalität von Softwareprodukten. Methoden zur Überprüfung von Fehlern und Mängeln in der Leistung oder der Benutzungsschnittstelle werden nicht vorgestellt[10].

2.3 Fehlverhalten von Software und seine Kosten

Welchen Effekt haben die in Kapitel 2.2 aufgezeigten Fehler auf das Verhalten der Software? Wie wirkt sich dies auf die menschliche Gesellschaft aus? Mit diesen Fragen wird sich dieses Kapitel unter technischen, benutzungsorientierten, ökonomischen, gesellschaftlichen und politischen Gesichtspunkten beschäftigen. Dazu ist zusätzlich zu den Klassifizierungen von Kapitel 2.2 eine Einteilung des Fehlverhaltens unter dem Aspekt der Auswirkungen[11] nötig.

Die hier vorgenommene Einteilung der Fehler orientiert sich an Basili und Rombach bzw. Thayer et al. (vgl. [BaR 87], S. 350; [TLN 78]), welche die Schwere von Fehlern wie folgt kategorisieren:

1. kritisch: kompletter Ausfall der Produktion bzw. Aufgabe nicht erfüllbar,
2. hoch: deutliche Beeinträchtigung der Produktion, Leistung herabgesetzt,
3. mittel: Verhinderung der vollen Ausnutzung von Programmfähigkeiten, aber mit Kompensationsmöglichkeiten,

[10] Literatur zum Thema Benutzungsfreundlichkeit: siehe [Op& 88] und DIN [EN] ISO 9241, Teile 10 bis 17; zur Leistungsüberprüfung: siehe [Smi 90].

[11] Da die Auswirkungen vom Einsatz der Software und nicht von der Art der Fehler abhängen, wurde dies nicht in Kapitel 2.2 aufgeführt.

4. niedrig: geringe oder „kosmetische“ Probleme, Leistung bleibt erhalten,
5. unproblematisch: nicht geforderte Softwareverbesserung.

Die aufgeführten fünf Auswirkungen des Softwarefehlverhaltens beziehen sich nur auf das unmittelbar betroffene Produktionssystem. Unter ökonomischen Gesichtspunkten ist aber eine Bewertung des Fehlverhaltens aufgrund der Kosten der notwendigen Korrektur- oder Ersatzmaßnahmen in der Software, im Produktionssystem und in der betroffenen Umgebung des Produktionssystems erforderlich. Der Fehlstart der Rakete „Ariane 5“ am 4. 6. 1996 ist ein Beispiel dafür. Unter gesellschaftlichen und politischen Gesichtspunkten ist das Fehlverhalten danach zu bewerten, wie weit es das Wirtschaftssystem, das Gesellschaftssystem, die Umwelt, das Leben einzelner Menschen oder gar die Existenz der Menschheit bedroht. Falls letzteres der Fall ist, müssen die Auswirkungen als *katastrophal* angesehen werden.

Als nur ärgerlich oder störend sind die folgenden Auswirkungen einzustufen:

- Studierende der NC State University erhielten im Jahre 1988 falsche Gebührenrechnungen. Das Programm erzeugte zwar korrekte Rechnungen, adressierte sie dann aber an die falschen Studierenden (*SEN*, Vol. 13, No. 4, Okt. 1988, S. 9).
- Eine Frau wurde durch mysteriöse Telefonanrufe jeden Morgen um $4^{\underline{30}}$ Uhr geweckt. Die Britische Telecom überwachte ihre Leitung mehrere Monate lang und fand schließlich heraus, daß die Anrufe durch einen Programmierfehler in ihrem eigenen Testcomputer verursacht wurden (*SEN*, Vol. 18, No. 1, Jan. 1993, S. 6 f.).
- 41.000 Einwohner/innen von Paris wurden wegen Mord, Erpressung, Prostitution, Drogenhandel und anderer schwerer Verbrechen angeklagt. Allerdings wurden nur Geldstrafen von (umgerechnet) ca. 100 DM bis 300 DM verlangt. Der Fehler entstand durch die Vertauschung bzw. Veränderung der Standardcodes für Vergehen und Verbrechen beim Lesen von einem Magnetband (*SEN*, Vol. 14, No. 6, Okt. 1989, S. 3).

Als kritisch sind dagegen folgende Fehler einzustufen:

- Eine Bereichsüberschreitung (Overflow) „im Verbund mit fälschlicherweise vom Rechner als Steuerungsdaten interpretierten Statusmeldungen [hat] dazu geführt, daß die Rakete [Ariane 5] über das fehlerhafte Programm mit einer Kurskorrektur bedacht wurde. Diese führte[n] dann innerhalb der Rakete zu Gegenmaßnahmen in Form einer Schubkorrektur zur Gegensteuerung und letztendlich zur Aktivierung der automatischen Selbstzerstörung. Der Fehler existierte bereits in der Ariane 4, ist dort aber wegen anderer Beschleunigungswerte nie aufgetreten. Eine nachträgliche Simulation konnte die Ursache exakt reproduzieren. Vor dem Start hatte das ESA-Team von einer solchen Simulation abgesehen — aus Kostengründen.“ (Zitat aus der Zeitschrift *iX*, Sept. 1996, S. 32).

- Das Weltwirtschaftssystem war durch den Kollaps des Aktienhandels im Oktober 1987 und abrupte Kursschwankungen in der Folgezeit bedroht. Als eine Ursache davon wurde der automatische Aktienhandel durch Programme identifiziert. Laut San Francisco Chronicle vom 11. 5. 1988 kündigten daher fünf große Wallstreet-Banken an, auf diese Form des Aktienhandels zu verzichten, vier andere Banken wollten dies aber fortsetzen (*SEN*, Vol. 13, No. 3, Juli 1988, S. 9).

- Ein 99 Jahre alter Mann ließ im Jahre 1989 sein Blut in einem Krankenhaus untersuchen. Dabei wurde die Anzahl seiner weißen Blutkörperchen vom Computer als „normal" eingestuft, obwohl sie weit außerhalb der Norm lag. Des Rätsels Lösung: das Geburtsjahr war als 89 (und nicht 1889) eingegeben und als 1989 interpretiert worden. Für einen neugeborenen Säugling waren die Werte normal (*SEN*, Vol. 15, No. 2, April 1990, S. 5).

Die gesundheitsbedrohenden oder lebensgefährlichen Auswirkungen von Fehlern oder Mängeln in der Software zeigten sich z. B. auf folgende Weise:

- Anfang Juni 1988 flossen innerhalb von fünf Stunden etwa 20 Millionen Liter ungereinigtes Abwasser in den Fluß Willamette bei Portland, Oregon. Die Umweltverseuchung wurde durch einen Stromausfall des Computers verursacht, der die Pumpen abschaltete. Ein ähnlicher Fehler war 1985 schon einmal aufgetreten. Daraufhin war das System so konzipiert worden, daß die Operateure die Computerkommandos außer Kraft setzen konnten. Aber erst am 6. Juni 1988 wurde bemerkt, daß dies nur möglich war, wenn der Computer mit elektrischem Strom versorgt wurde. Fazit: Das Konzept eines vollständig sicheren Systems erfordert manuelle Eingriffsmöglichkeiten, die unabhängig vom Computersystem funktionieren (*SEN*, Vol. 13, No. 3, Juli 1988, S. 4).

- Am 3. Juli 1988 ließ der Kommandeur des Kreuzers „Vincennes" der U.S. Navy versehentlich einen iranischen Airbus abschießen, was den Tod von 290 Menschen zur Folge hatte. Der offizielle Untersuchungsbericht bemerkt, daß auf den Bildschirmen des Aegis-Systems die korrekten Werte angezeigt wurden (Flugzeug im Steigflug mit korrekter Geschwindigkeit auf richtigem Kurs). Unter dem Streß der Schlacht im persischen Golf wurden diese Angaben von einem jungen Offizier falsch interpretiert (als vermutlich militärisches Flugzeug im Sinkflug mit viel größerer Geschwindigkeit auf falschem Kurs). Diese Angabe gelangte auf mehreren Wegen zum Kapitän, so daß dieser von einer sicheren Information ausging.
  Das amerikanische „Risks Forum" schließt daraus, daß die Benutzungsschnittstelle vermutlich nicht sehr gut entworfen wurde. Ein Bericht mit „ungünstigen Testergebnissen" soll außerdem vorliegen (und dem US-amerikanischen Kongreß vorenthalten worden sein). Tom Wicker, ein Kolumnist der New York Times, zieht daraus folgende Schlüsse:

> „Das Aegis-System im persischen Golf reagierte auf etwas, das jeder an Bord der Vincennes wahrscheinlich schon einmal gesehen hatte und für das er trainiert worden ist: die Signatur eines zivilen Flugzeugs. Wie aber werden die Mannschaften reagieren, die mit SDI oder ICBM [Interkontinentalraketen] zu tun haben und die im Ernstfall etwas sehen werden, was niemand zuvor gesehen hat? Dafür, daß wir uns unerbittlich in eine Welt bewegen, die von Computern kontrolliert wird, zahlen wir einen Preis. Unsere Fähigkeit, über unser eigenes Schicksal zu bestimmen, scheint eher abzunehmen als zuzunehmen.“ (*SEN*, Vol. 13, No. 4, S. 3 f.).

Die Häufigkeit des Fehlverhaltens von Software hängt natürlich von dem Aufwand ab, der vor der Freigabe für das Vermeiden oder Beseitigen von Fehlern getrieben wird. Damit beschäftigt sich das nächste Kapitel.

2.4 Vermeidung und Behebung von Fehlern

In Kapitel 2 wurde bisher behandelt, welche Ursachen die Softwarefehler haben, wie häufig und in welcher Art sie auftreten und welche Auswirkungen sie haben. Können diese Fehler nicht einfach durch größere Anstrengungen vermieden werden?

Wenn die Ursachenanalyse in Kapitel 2.1 richtig war, müßten dazu „nur“

- bessere Kommunikationsmethoden angewandt und psychologische Kommunikationsblockaden verhindert werden;
- die Motivation des Entwicklungspersonals entscheidend verbessert und Ermüdung und Streß abgebaut werden;
- Methoden und Werkzeuge entwickelt und angewandt werden, welche die Gedächtnisprobleme überwinden;
- das Anwendungs- und Problemverständnis, das Problemlösungsverhalten, die Beherrschung von Semantik und Syntax der benutzten Sprachen, das Verständnis der Hardware- und Softwareumgebung, die Informations- und Dokumentationstechniken sowie das mechanische Umsetzen von Information entscheidend verbessert werden;
- die Komplexität von Systemen mit Nebenläufigkeit bzw. mehreren Prozessoren (multiprocessing) gemeistert werden.

Dies sind allgemeine Probleme der Softwaretechnik und der Menschenführung, die über das Thema dieses Buches hinausgehen. Es werden daher im folgenden nur einige Beispiele angeführt.

Eine funktionierende Kommunikation setzt eine gemeinsame Erfahrungswelt voraus. Diese wird z. B. bei neuartigen Anwendungen durch die Herstellung von Prototypen geschaffen. Die Gedächtnisprobleme bei der Programmiersprachenbeherrschung können z. B. durch syntaxgesteuerte Editoren oder ausreichende Compilermeldungen überwunden werden. Entsprechende Probleme mit Maschinenkonfigurationen oder Softwarevarianten lassen sich durch entsprechendes Konfigurationsmanagement mit geeigneten Projektbibliotheken bekämpfen.

Da die zuvor dargestellten Methoden aber nicht perfekt sind, können Fehler nicht vollständig vermieden werden. Allerdings führen die angegebenen Maßnahmen zu einer Fehlerreduzierung.

Es bleibt nun die Hoffnung, in einem nachfolgenden analytischen Prozeß alle Fehler zu finden. Dabei sollten die Fehler natürlich so früh wie möglich beseitigt werden, da es schwierig und somit kostenintensiv ist, in frühen Phasen entstandene Fehler erst in der Codierungs- und Testphase zu beheben (s. Abbildung 2.5)

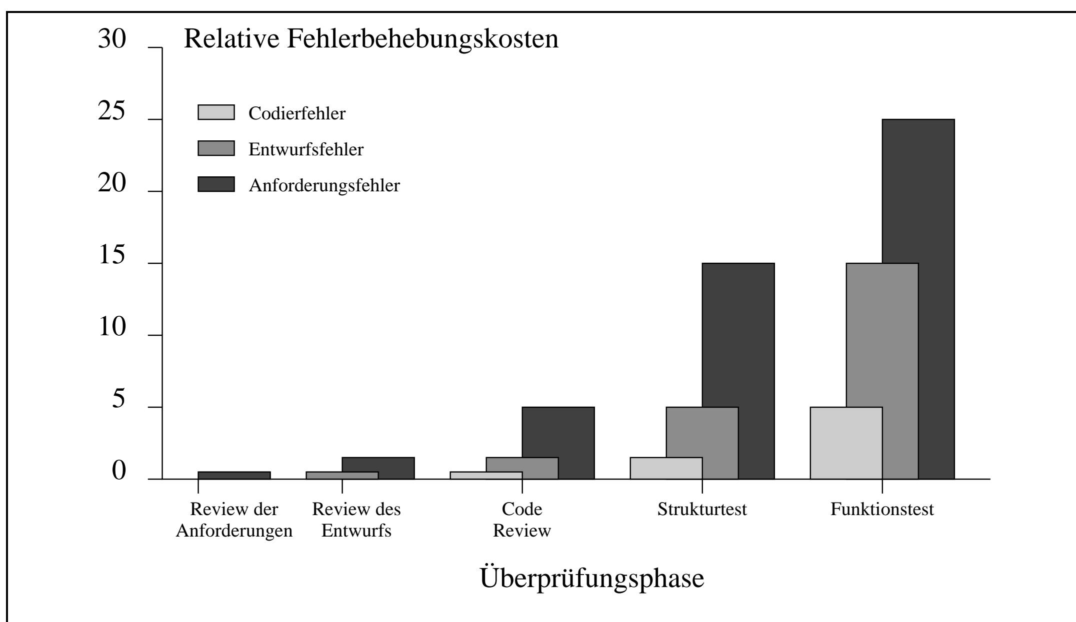

Abb. 2.5: Relative Fehlerbehebungskosten in Abhängigkeit vom Zeitpunkt der Überprüfung und Korrektur ([FLS 91a], S. 19, nach [Dun 84])

Mängel und Fehler in der Anforderungsdefinition können nur durch ein nochmaliges gedankliches Durchgehen und Durchspielen der Konsequenzen (**Review**) aufgedeckt werden. Fehler in der Systemspezifikation können durch ein ähnliches Vorgehen oder durch „Ausführung“ der Spezifikation aufgedeckt werden, falls die Art der Spezifikation dies möglich macht. Es handelt sich dabei also um eine frühzeitige **Simulation** des geplanten Systemverhaltens. Mängel und Fehler in der Anforderungsdefinition und Systemspezifikation lassen sich damit aber offenbar nicht vollständig aufdecken:

die Reviews unterliegen den allgemeinen Problemen der Kommunikation zwischen Problemfachleuten und Softwareentwicklungsfachleuten, und es gibt keine systematischen Verfahren, komplexes Systemverhalten hinreichend zu simulieren.

Es bleibt die Hoffnung, wenigstens Programme abzuliefern, die keine Fehler mehr gegenüber der Systemspezifikation aufweisen. Ein solcher Nachweis könnte durch eine formale **Verifikation** des Programms, d. h. durch einen formalen Beweis der Korrektheit des Programms gegenüber der Spezifikation, erbracht werden. Dabei tritt aber eine Vielzahl von Problemen auf, von denen hier[12] nur zwei genannt seien:

- Fehler in Compiler, Betriebssystem oder Hardware werden nicht berücksichtigt.
- Ein fehlerhaftes Programm kann nicht als korrekt bewiesen werden. Der Beweisversuch liefert nur begrenzt Hinweise für die Fehlerbehebung: wenn ein Modul A nicht verifiziert werden kann, muß A nicht fehlerhaft sein — vielmehr können auch die Spezifikationen der Schnittstellen falsch sein.

Es bleibt also nur die Möglichkeit, durch eine systematische Analyse der Spezifikation, des Entwurfs und der Implementierung des Programms alle Fehler zu finden. Wie sieht dies z. B. bei der Implementierung aus? Wenn ein Programm z. B. 60 Fehler enthält, würde es ausreichen, das Programm mit 60 Testdaten zu testen, die jeweils ein Fehlverhalten aufdecken, und dann die Fehlerursache zu lokalisieren und zu beseitigen. Es müßte also nur dieser **ideale Test** angewendet werden. Leider gibt es kein Berechnungsverfahren, welches für ein beliebiges Programm und eine gegebene funktionale Spezifikation einen idealen Test erzeugt (s. [How 76])[13]. Diese Aussage ist auf praktische Programme nicht genau anwendbar, da sie nur mit beschränkten Zahlwerten (z. B. $2^{31} - 1$ als Maximalwert für Integer) rechnen können. Dabei gibt es also nur endlich viele Eingabekombinationen, die im Prinzip alle durchgespielt werden könnten. Das ist aber nicht praktikabel: Bei nur drei Eingabewerten von je 16 Bit und einer Testgeschwindigkeit von 0,0001 Sekunden pro Testdatum benötigt ein vollständiger Test aller Möglichkeiten schon ca. 900 Jahre. Dieser Test heißt also mit Recht **erschöpfender Test** (im doppelten Wortsinn).

Es bleibt nur die praktische Möglichkeit, die Software stichprobenartig statisch zu überprüfen oder dynamisch zu testen. Die Überprüfung kann dabei nur die Anwesenheit gewisser Typen von Fehlern ausschließen (sonst wäre es eine totale Verifika-

[12] genaueres siehe Kapitel 12.4

[13] Falls dies möglich wäre, könnte folgendermaßen die Äquivalenz von zwei Programmen P_1 und P_2 festgestellt werden, d. h. entschieden werden, ob sie dieselbe Funktion realisieren: Da Programme auch als Spezifikation interpretiert werden können, kann im Beweis P_1 als Spezifikation für P_2 gewählt werden. Wenn P_1 und P_2 nicht äquivalent sind, würde das Berechnungsproblem (für den idealen Test) eine Testdatenmenge mit Eingabewerten liefern, bei denen P_1 und P_2 verschiedene Ergebnisse liefern. (Wenn die Programme äquivalent sind, wird eine leere Testdatenmenge erzeugt, da es keine Eingabe gibt, bei denen sie abweichende Ergebnisse liefern.) Die Tatsache, ob die erzeugte Testdatenmenge leer ist (oder nicht), entscheidet also die Äquivalenz von P_1 und P_2. Dies ist aber schon für stets anhaltende Programme unentscheidbar.

tion). Das Testen kann sich ebenfalls nur an gewissen Typen von Fehlern orientieren oder stichprobenartig den Eingabebereich bzw. Eingabefolgen des Programms überdecken. Das kostengünstige Ermitteln geeigneter Eingaben für das Testen eines Programms stellt also ein typisches ingenieurwissenschaftliches Problem dar. Noch schwieriger scheint die Auswertung der Programmausgaben zu sein, denn nur durch einen Vergleich der Ausgaben des Programms mit den von der Spezifikation geforderten Ausgaben kann das Fehlverhalten festgestellt werden. Das Aufdecken von Fehlern stellt aber für das Programmierpersonal offenbar ein psychologisches Problem dar. Da Programmiererinnen ihre Tätigkeit (Spezifikation, Entwurf, Codierung) als konstruktiven Prozeß erleben, empfinden sie das Testen als einen destruktiven, ja geradezu sadistischen Prozeß (s. [Mye 79], Kap. 2), obwohl das Fernziel der Fehlerbehebung ehrenwert ist. Nach Gruenberger (s. [Gru 72]) macht eine Programmiererin daher folgende psychologische Phasen durch:

1. Optimismus: Testen ist nicht nötig; ich weiß, daß meine Programme perfekt sind.
2. Bockigkeit: Hoppla, mein „perfektes“ Programm ist falsch. Ich will meine Programme aber nicht testen.
3. Dienst nach Vorschrift: Ich gebe kein Programm mehr für die Produktion frei. Ich teste und teste und teste ...
4. Kompromißbereitschaft: Es gibt einen goldenen Mittelweg. Die Kunst des Testens ist es zu wissen, wann mit dem Testen aufgehört werden kann.

Programmiererinnen müssen daher psychologisch geschult werden, um „selbstlos“ (**egoless**) zu programmieren und zu testen (s. [Wei 71]). Die Bewertung von möglichen Fehlern muß dabei geändert werden. Eine Programmiererin sollte einen Fehler in dem von ihr geschriebenen Programm — nicht in „meinem“ Programm — einfach als Mangel des Produkts ansehen, den es zu entdecken und zu verbessern gilt. Keinesfalls sollte das Erzeugen von Fehlern als charakterlicher Mangel des produzierenden Menschen aufgefaßt werden, den es zu verbergen gilt.
Das psychologisch angenehmere Testprinzip „zu zeigen, daß ein Programm korrekt ist“, ist nicht erfolgversprechend; dies liegt daran, daß mit Testen nur das Vorhandensein von Fehlern, nicht aber die Fehlerfreiheit eines Programms gezeigt werden kann[14]. Nur das psychologisch unangenehmere destruktive Testen ist ein erfolgversprechender Prozeß: es ist relativ leicht, in einem großen Programm Fehler zu finden, und jeder gefundene Fehler motiviert die Testperson.
Aus der „Psychologie des Testens“ wird oft gefolgert, daß Programmiererinnen nicht ihre eigenen Programme testen sollten. Ein zusätzliches Argument sind Mißverständnisse beim Interpretieren der Spezifikation der Anforderungen: die Programmiererinnen würden auch beim Testen — wie beim Programmieren — das funktionale

[14] Trotzdem haben Informatikerinnen eine Tendenz, viermal so viele „positive“ Tests wie „negative“ Tests auszuführen. Positive Tests zeigen nur, daß das Programm „läuft“, negative Tests fordern es dagegen mit Sonderfällen und nicht gewünschten Eingaben heraus (s. [Le& 93], S. 210).

Verhalten des Programms als richtig ansehen, obwohl es nicht der Spezifikation entspricht. Auch Programmierorganisationen, wie z. B. Projekt-Teams, sollten nicht ihre eigenen Programme testen, da folgender Konflikt vorliegt: Die Organisation wird daran gemessen, ob sie das Programm in einem bestimmten Zeitraum und für bestimmte Kosten erstellen kann. Bei gründlichem Testen sinkt die Wahrscheinlichkeit, diese Anforderungen zu erfüllen.
Testen sollte also als eigenständige Aufgabe angesehen werden. Folgende **Organisationsschemata** sind dabei möglich (s. [Mil 78]):

- Testen durch eine Untergruppe des Projekts bzw. eine unabhängige Gruppe der Firma, die nur für das Testen Verantwortung trägt.
- Testen als Funktion der Kundin bzw. der Auftraggeberin (als Akzeptanztest),
- Testen durch eine unabhängige Vertragspartnerin, die im Auftrage der Kundin handelt. Dies wird **unabhängige Verifikation und Validierung** (independent verification and validation [IV&V]) genannt.

Die Forderung nach einer personellen Trennung der Programmier- und Testaufgabe ist umstritten. Die Trennung setzt eine gute Programmdokumentation und objektive Kriterien für die Mängel- und Fehlerfreiheit der Software voraus. Unter diesen Voraussetzungen ist es eine kostengünstige Alternative, Programme durch die entwickelnden Personen selbst testen zu lassen. Allerdings sollten unabhängige Personen die Testergebnisse begutachten (s. [GeH 88], S. 692). Programme können nicht nur durch Testen, d. h. durch Ausführen des Programms mit konkreten Daten, überprüft werden. Vielmehr kann der Programmtext auch durch sogenannte **Inspektion** direkt von Menschen untersucht werden (genaueres s. Kapitel 12.1). Können mit dieser Methode alle restlichen Programmfehler gefunden werden? Offenbar nein, denn neben den Kommunikations-, Konzentrations- und Wissensproblemen, die zu Fehlern beim Programmieren und beim Inspizieren führen können, gibt es noch das Problem der Täuschung . Dabei wird zwischen verschiedenen Arten der Täuschung unterschieden (s. [Zem 85]):

1. Bei der **Trick-** oder **Ablauftäuschung** weicht der Programmtext von dem ab, was normale Programmiererinnen denken. Dies erschwert z. B. beim Ausnutzen von Seiteneffekten (Trickprogrammierung) das Verständnis eines Programms.
2. Eine **Erwartungstäuschung** entsteht durch Prägung oder Prädisposition, d. h. durch eine bestimmte Erwartung, mit der eine Person an eine Aufgabe herangeht. Durch eine Kommentarzeile wird eine Testerin z. B. zu der Annahme verleitet, daß der Programmtext das bewirkt, was der Kommentar aussagt. (Bei der Inspektion sollte also die Kommentarzeile abgedeckt werden.)

3. Zu einer **Hemmungstäuschung** kommt es unter dem Eindruck einer Hemmung durch zuvor gelerntes Wissen. Die Aufgabenstellung kann z. B. fälschlicherweise erweitert oder durch vorgetäuschte Zusatzbedingungen eingeschränkt werden.
   Ein Stenograf im Bundestag hatte z. B. gerade gelernt, daß es „freie Arztwahl" heißen muß, obwohl sich dies bei einem Sachsen wie „freie Ortswahl" anhört. Um diesen Fehler nicht zu wiederholen, machte er dann aus dem Zwischenruf eines Saarländers „Sie predigen ja den letzten Beter aus der Kirche!", weil er seinen Dialekt für sächsisch hielt, den Satz „Sie predigen ja den letzten Peter aus der Kirche!". (Jedes Textdokument muß also unvoreingenommen betrachtet werden.)

4. Bei einer **Überdeckungstäuschung** überdeckt ein großer, starker oder überwiegender Eindruck einen kleinen, schwachen oder unbedeutenden Eindruck.
   Das Wort „Testmethodon" wird man z. B. richtig als „Testmethoden" lesen[15]. (Ein Vorgehen wie bei einer [Lese-]Anfängerin, welche die kleinsten Einheiten [die Buchstaben] betrachtet, vermeidet diese Täuschung.)

5. Eine **Wiederholungstäuschung** entsteht durch den Analogieschluß, daß scheinbar gleiche Vorgänge auch gleiche oder ähnliche Ergebnisse haben. Daher werden z. B. ähnliche Programmteile nicht mehr (oder nur flüchtig) geprüft, weil ein Programmteil schon als fehlerfrei beurteilt wurde.
   (Kopierte Programmteile sind also besonders sorgfältig zu betrachten — die Entstehungsgeschichte ist also wichtig — bzw. gleiche Teile sind bei der Verwendung von Klassen nur einmal — in einer Oberklasse — zu realisieren).

Weitere methodische Hinweise, wie man diesen Täuschungen entgehen kann, gibt G. Zemanek (s. [Zem 85]). Aber nur „Supertestpersonen" werden mit dieser Hilfe alle restlichen Fehler in einem größeren Programm finden.

2.5 Kosten und Nutzen des Testens

In Kapitel 2.4 wurden verschiedene Analysemethoden vorgestellt, mit denen Fehler in Programmen aufgedeckt bzw. die Fehlerfreiheit von Programmen in gewissen Fällen festgestellt werden kann. Dabei wurde erläutert, warum die Programmverifikation für große Programme nicht anwendbar ist und das Fehleraufdecken per Review und Inspektion nicht hundertprozentig zuverlässig ist. Damit ist das Testen von Programmen eine unverzichtbare Analysemethode zum Aufdecken von Fehlern.

Dieses Kapitel wird folgender Frage nachgehen: Welche Kosten sind mit dem Testen verbunden und welchen Nutzen hat das Testen? Das hängt natürlich davon ab, mit welchen Methoden gearbeitet wird und nach welchen Kriterien das Testen beendet wird. Für die in diesem Buch vorgestellten Methoden werden im Einzelfall in den entsprechenden Kapiteln Kosten und Nutzen aufgeführt. An dieser Stelle sollen nur

[15] in einem Informatikbuch — in einem Buch über Drogen aber wahrscheinlich als „Testmethadon"

pauschale Angaben über Erfahrungswerte aufgrund der bisherigen Praxis gemacht werden (die sich hoffentlich in der Zukunft verbessern lassen).
Schätzungsweise über 80% der Kosten der Datenverarbeitung sind Softwarekosten, da die Fortschritte im Softwarebereich in den letzten Jahrzehnten geringer waren als im Hardwarebereich (s. [Zwe 81], S. 3). Bei den Softwarekosten ist der Wartungsaufwand von Firmen ca. 40% bis 45% des Entwicklungsaufwands und sogar 70% pro Softwaresystem (s. [Boe 77], S. 7; [Can 81]). (Der Unterschied ist damit zu erklären, daß es viele Neuentwicklungen gibt, die noch keine Wartungskosten verursachen.) In Abhängigkeit von den Anforderungen, welche an die Zuverlässigkeit der Software gestellt werden, schwankt der Testaufwand zwischen 15% und 50% des Gesamtentwicklungsaufwands (s. [Boe 77], S. 8 und S. 10 f.).

Der Aufwand für das Testen bzw. Überprüfen hängt von neun Faktoren F_1 bis F_9 ab, d. h. es gilt näherungsweise:

$$\text{Prüfkosten} = (F_1 + F_2 + \cdots + F_9) * \text{Entwicklungskosten}$$

Es handelt sich dabei um folgende Faktoren (s. [RDB 75]):

F_1: Programmgröße .... Faktor 0,250 bis 0,050

F_2: Beginn der unabhängigen Überprüfung (Validierung) .. Faktor 0,125 bis 0,008

F_3: Vorhandensein von Werkzeugen .... Faktor 0,093 bis 0,000

F_4: Dokumentation des Programms .... Faktor 0,062 bis 0,016

F_5: Hardware und Systemsoftware .... Faktor 0,031 bis 0,016

F_6: Typ des Programms .... Faktor 0,031 bis 0,008

F_7: Vertrautheit mit Programm und Werkzeugen .... Faktor 0,031 bis 0,008

F_8: Informationsfluß zum Auftraggeber .... Faktor 0,031 bis 0,008

F_9: Hardwareressourcen .... Faktor 0,031 bis 0,008

Der Prüfaufwand ist bei kleinen Programmen prozentual größer als bei großen Programmen; beginnt das Überprüfen erst beim Fehlerbeseitigen, so ist der Aufwand am größten, während die Überprüfung von früher schon einmal überprüften (aber geänderten) Programmen den geringsten Faktor impliziert. Das Vorhandensein von Werkzeugen und eine gute Programmdokumentation vermindern ebenfalls den Prüfaufwand. Entsprechendes gilt für ausgereifte Hardware/Systemsoftware, Nicht-Echtzeit-Programme (im Gegensatz zu interaktiven oder Echtzeit-Anwendungen), einen freien Informationsaustausch mit dem Auftraggeber und im richtigen Moment und in ausreichender Menge verfügbare Hardware für Testläufe.

Für die Behebung eines Fehlers in der Anforderungsdefinition muß ein Betrag aufgewendet werden, der von der Entwicklungsphase abhängt, in welcher er entdeckt und beseitigt wird. Nach Stahl/Nomicos sind dies z. B. folgende Dollarbeträge (s. [StN 85], zitiert nach [GoT 86]):

Anforderungs- und Systemspezifikation	349 $
Entwurf	876 $
Codierung und Modultest	1.750 $
Test und Integration	12.782 $

Fehlerbehebung in späteren Phasen bedeutet also einen erhöhten Aufwand um den Faktor[16] 2,5 oder 5 oder sogar 37. Frühzeitiges Testen bzw. Überprüfen hat also einen hohen Nutzen.

Wie bereits erwähnt wurde, gibt es vor der Testphase etwa 30 bis 100 Fehler pro 1000 Anweisungen im Programm (s. Kapitel 2.2, S. 25). Ca. 95% dieser Fehler werden bis zur Auslieferung beseitigt, d. h. 28,5 bis 95 Fehler pro 1000 Anweisungen (s. [Mye 86], S. 65). Das ist zwar ein gewisser Erfolg, bedeutet aber, daß ca. 5% der Fehler trotz aller Anstrengungen noch vorhanden sind, pro 1000 Anweisungen also noch 1,5 bis 5 Fehler. Eine Spanne von nur 0,5 bis 3,3 Fehler pro 1000 Anweisungen gibt W. Myers an, während Valk dagegen 0,25 bis 10 Fehler angibt (s. [Mye 86], S. 62; [Val 87], S. 59). Ein großes Programmsystem mit 100.000 Anweisungen hat demnach noch etwa 25 bis 1.000 Fehler, wenn es in Betrieb genommen wird.

2.6 Leben mit Fehlern

„Diejenigen, die glauben, daß es leicht sein wird, Software zu entwerfen,
und daß Fehler verschwinden werden,
haben nicht die wirklichen Probleme angepackt."
— **David L. Parnas**

Das vorherige Kapitel hat aufgezeigt, daß bei der Entwicklung von Software ein erheblicher Teil der Kosten für das Testen aufgewendet wird. Dennoch sind ausgelieferte Programme mit Fehlern behaftet, die erst während der Wartungsphase behoben werden können. Das Problem wird noch dadurch erschwert, daß bei der sogenannten Korrektur von Fehlern die Fehlerursachen oft nur z. T. behoben werden oder sogar neue Fehler in das Programm eingebaut werden. Das Problem verkleinert sich andererseits dadurch, daß viele Fehler geringe Auswirkungen haben, d. h. nur ca. 14% der Fehler sind schwerwiegend (s. [RDB 75], S. 152). Somit sind noch etwa

[16] Diese Faktoren ergeben sich aus obigen Zahlenwerten.

3 bis 140 schwerwiegende Fehler bei 100.000 Anweisungen vorhanden. Wie können und sollen die beteiligten Personen mit dieser Situation umgehen?

Eine Käuferin von Software kann sich auf die Gewährleistungspflicht des Herstellerbetriebs der Software berufen und auf rasche Korrektur der restlichen Fehler[17] in der Garantiezeit hoffen. Diese Hoffnung trügt aber oft, denn die Herstellerbetriebe können die in den vorherigen Unterkapiteln aufgezeigten Probleme aus prinzipiellen und wirtschaftlichen Gründen nicht lösen. Sie sichern daher nur wenige und schwache Eigenschaften des Produkts zu, indem sie z. B. im Kaufvertrag formulieren:

> „Fa. CHAOS[18] leistet Gewähr dafür, daß die lizenzierte Software im wesentlichen entsprechend dem Benutzerhandbuch arbeitet, wie es zum Tage der Lieferung vorliegt. Die Verpflichtung von CHAOS im Rahmen des vereinbarten Gewährleistungszeitraums beschränkt sich darauf, das Mögliche zur Behebung etwaiger Fehler zu unternehmen und dem Lizenznehmer eine korrigierte Version der lizenzierten Software so bald wie möglich nach einer Fehlermeldung zur Verfügung zu stellen. CHAOS leistet keine Gewähr dafür, daß der Betrieb einer lizenzierten Software unterbrechungsfrei oder fehlerfrei erfolgt oder in der lizenzierten Software enthaltene Funktionen in den Kombinationen arbeiten, die der Lizenznehmer zur Lösung seiner Anforderungen gewählt hat."

Das Produkthaftungsgesetz verringert die Möglichkeiten der Herstellerbetriebe, sich der Gewährleistung zu entziehen, erheblich. Es sind nur noch eng definierte Entlastungsmöglichkeiten bei der Haftung vorgesehen, etwa die objektive Unvorhersehbarkeit eines konkreten Fehlers nach dem Stand von Wissenschaft und Technik. Die Haftungsfrage wird somit zu einer Methodenfrage (s. [Koc 89], [KrE 92]). Allerdings liegt nach dem Produkthaftungsgesetz ein Fehler nur vor, wenn der gekaufte Gegenstand nicht die Sicherheit bietet, die erwartet werden darf. Ein Softwarefehler muß sich also auf Leben oder Gesundheit von Menschen auswirken (vgl. Beispiele aus Kapitel 2.3), einfache Rechenfehler fallen nicht unter die Produkthaftung.

Eine Käuferin von Software kann auf eine Art „Stiftung Warentest" hoffen, die ihr hilft, den Kauf von sehr fehlerhaften Programmen zu vermeiden. In der Tat wurde eine „Gütegemeinschaft Software e. V." (GGS) ins Leben gerufen, die vom RAL (Deutsches Institut für Gütesicherung und Kennzeichnung e. V.) im August 1985 anerkannt wurde. Mit der Verleihung des Gütezeichens Software durch diese Institution wird zugleich auch das Zeichen „DIN-geprüft" vergeben, da das Deutsche Institut für Normung e. V. die Prüfrichtlinien der GGS als Norm 66285 in sein Regelwerk übernommen hat[19]. Das Gütezeichen bestätigt u. a. die Übereinstimmung

[17] Laut §459 Abs. 1 BGB besteht ein Fehler in der Abweichung von dem vertraglich vorausgesetzten oder gewöhnlichen Gebrauch. Dieser Begriff umfaßt also die Definition von Fehler und Mangel aus Kapitel 2.2. Genaueres zu in der Rechtsprechung anerkannten Fehlern und Mängeln findet sich in [Hül 94], Teil 2.

[18] Der Name der Firma wurde verändert, ist mir, E. R., aber bekannt.

[19] Die Norm ist mittlerweile durch DIN ISO/IEC 12119 abgelöst worden.

der Herstellerangaben mit dem Programm und der Dokumentation. Das Gütezeichen darf nur von bestimmten Prüfstellen der GGS vergeben werden, die ihrerseits von der Gesellschaft für Mathematik und Datenverarbeitung (GMD), einer Großforschungseinrichtung des Bundes, anerkannt und überwacht werden (s. [Krü 85]) bzw. neuerdings von der DEKITZ (DIN).

Damit sind also alle Voraussetzungen gegeben für die **Zertifizierung**, d. h. die Prüfung und Bewertung eines Software-Produkts oder eines Qualitätssicherungssystems, mit dem die Erfüllung vorgegebener Anforderungen (z. B. Normkonformität) nachgewiesen wird. Dabei wird bei Erfüllung der gestellten Anforderungen von einer unabhängigen Prüfstelle (z. B. TÜV) ein Zertifikat vergeben (s. [Wal 90]). Nach dieser Definition kann also eine Zertifizierungsautorität anderen Organisationen das Zertifizierungsrecht übertragen. Das Zertifizierungsrecht für COBOL-Compiler hat z. B. die GMD vom Federal Software Testing Center der USA erhalten. Dies ist zugleich ein Beispiel für eine Zertifizierung, die das Erfülltsein einer Norm (hier DIN 66028 und ISO 1989-1978 für COBOL) bestätigt, indem eine bestimmte **Normkonformitätsprüfung** angewandt wird.

Neuerdings verlagert sich der Schwerpunkt der Zertifizierungen von den für jedes Produkt (und eigentlich jede Version) vorzunehmenden (teuren) Produktzertifizierungen auf die einmalige Zertifizierung des *Entwicklungsprozesses* einer Organisation, die Software herstellt. Was für einen geordneten Ablauf des Prozesses unbedingt notwendig ist, wird in 20 Absätzen[20] der DIN ISO 9001 beschrieben, ergänzt um Anwendungsrichtlinien in DIN ISO 9000–3 (s. [Lam 94]). Wegen der schwachen Anforderungen dieser Normen orientiert man sich aber besser am Konzept des totalen Qualitätsmanagements (TQM), das in ein fünfstufiges Reifegradmodell für Organisationen umgesetzt wurde. (Genaueres dazu siehe in Abschnitt 3.7).

Käuferinnen können also eine gewisse Verbesserung der Zuverlässigkeit von Software erwarten. In der Bundesrepublik Deutschland sind Zertifikate bei öffentlichen Beschaffungen zwar noch nicht zentral vorgeschrieben; nach einem Beschluß des Rates der Europäischen Gemeinschaft vom 22.12.1986 müssen die Mitgliederstaaten jedoch solche Vorschriften erlassen (s. [Weg 87]).

Trotz aller (oder wegen fehlender) Zertifizierung können noch Fehler auftreten, da alle diese Prüfungen nur falsifikationsorientiert arbeiten, d. h. die Korrektheit oder Konformität des Programms natürlich nicht beweisen können (vgl. Kapitel 2.4). Daher sind Käuferinnen nach Ablauf der Gewährleistungsfrist für Software auf die Behebung der restlichen Fehler in der Wartungsphase angewiesen. Auch für die Software-Entwicklung ist dies die letzte Möglichkeit der Fehlerbehebung, da im praktischen Einsatz der Software immer wieder Eingaben getätigt werden, die beim Testen nicht bedacht wurden. Der Abschluß eines Wartungsvertrages beinhaltet aber

[20] Von der „Verantwortung der obersten Leitung“ über „Prüfungen“ bis zu „Statistischen Methoden“.

nur die Korrektur von erkannten Fehlern und die Anpassung an neue Hardware und Systemsoftware.

Die Auswirkungen der Fehler auf den Geschäftsablauf der Kundenbetriebe sind damit noch nicht behoben. Man kann nur versuchen, das Risiko dafür zu verringern oder abzuwälzen. Dies kann durch technische und organisatorische Schadensverhütungsmaßnahmen geschehen und durch den Abschluß von Versicherungen, die die auftretenden Schäden und Verluste zum großen Teil decken. Von Versicherungen werden aber i. allg. nur Hardwareschäden versichert, das Wiederherstellen von Datenbanken muß z. B. selbst organisiert und finanziert werden (s. [Bre 88]).

Die geschilderte Vorgehensweise ist für kommerzielle, unkritische Software unter ökonomischen Gesichtspunkten sinnvoll. Den notwendigen Softwaretestkosten werden die Kosten durch den Einsatz der fehlerhaften Software gegenübergestellt. Wenn genau bekannt wäre, zu welchem Testaufwand (mit entsprechenden Testkosten) welche Einsatzfehlerkosten gehören, könnte ein ökonomisch optimaler Testaufwand berechnet werden. Da die Beziehungen aber nicht bekannt sind, müssen Vermutungen angestellt und die möglichen Kosten der Fehler grob klassifiziert werden[21].

Für Software, die fehlerfrei funktionieren muß, damit die Gesundheit oder das Leben von Menschen nicht gefährdet wird, stellt sich natürlich eine andere Frage als die der Kosten (obwohl es Institutionen gibt, die Gesundheit und Leben von Menschen in Geldbeträgen ausdrücken). Es stellt sich die Frage nach der Verantwortbarkeit solcher Software. Mit dem Philosophen Hans Jonas bin ich der Meinung, daß folgende Maxime beherzigt werden sollte: „Handle so, daß die Wirkungen deiner Handlung verträglich sind mit der Permanenz echten menschlichen Lebens auf Erden." (s. [Jon 89], vgl. [Ram 86]). In ähnlicher Weise hat es die amerikanische Organisation ACM (Association for Computing Machinery) in dem Entwurf eines Kodex für professionelles Verhalten und Ethik (Code of Ethics and Professional Conduct) vom 12. 2. 1992 ausgedrückt:

> „Computerfachleute sollten sich verpflichten, ihre Wissenschaft zum Nutzen der Menschheit zu entwickeln, zu erweitern und zu benutzen und negative Folgen von Computersystemen — inklusive Gefahren für Gesundheit und Sicherheit — zu minimieren." (*CACM*, Vol. 35, No. 5, S. 95)

Wie verträgt sich das mit folgenden Erkenntnissen?

> „Entgegen dem Mythos der Unfehlbarkeit von Rechensystemen können diese sehr wohl versagen und tun dies auch. Folglich kann die Zuverlässigkeit von rechnergestützten Systemen nicht als gesichert betrachtet werden. Diese Tatsache gilt für alle solche Systeme, deren Fehlverhalten ein besonderes öffentliches

[21] Der Fehlstart der „Ariane 5" (s. Kap. 2.3) ist ein Beispiel dafür, daß diese Klassifikation nicht funktioniert hat; die Simulationskosten wären sicher geringer gewesen als die Kosten durch den Verlust des Satelliten.

> Risiko darstellt. Zunehmend hängen Menschenleben vom zuverlässigen Funktionieren von Systemen ab wie Steuersystemen für Luftverkehr und Hochgeschwindigkeitszüge, militärischer Waffenproduktion und Verteidigungssystemen sowie Gesundheitsversorgungs- und medizinischen Diagnosesystemen.“ (Resolution der ACM vom 8.10.1984, zitiert nach [Val 87], S. 62)

Im Sinne der Verantwortungsbeschränkung und Verantwortungsindividualisierung (s. [Wed 87], S. 325) mag es als Problem jedes einzelnen Menschen angesehen werden, ob er bei der Entwicklung und dem Testen solcher Systeme mitwirkt oder nicht. Es sollte aber doch einen öffentlichen Diskurs[22] darüber geben, ob und unter welchen Bedingungen diese Systeme eingesetzt werden dürfen. Diese Diskussion kann aber nur dann rational erfolgen, wenn die Entwickelnden von Software schonungslos erläutern, daß bei komplexen Systemen Unzuverlässigkeit prinzipiell unvermeidbar ist und daß ein hoher Grad von Zuverlässigkeit nur mit entsprechendem Aufwand erreichbar ist.

Ein Beispiel dafür ist die Auseinandersetzung um die Machbarkeit und die Risiken eines „Schutzschildes“ gegen angreifende Raketen, wie sie der ehemalige US-Präsident Reagan gefordert und mit der Strategischen Verteidigungsinitiative SDI (Strategic Defense Initiative) gefördert hat. David L. Parnas, bekannt durch das Geheimnisprinzip bei der Verwendung von Datentypen, der in einem Beratungsausschuß der SDI-Organisation mitarbeitete, ging mit seiner Kritik an SDI in vorbildlicher Weise an die Öffentlichkeit. Er kritisierte, daß andere Wissenschaftler/innen weiter an SDI mitarbeiteten, da sie Geld für interessante Forschungen erhielten, obwohl ihnen klar war, daß SDI seine Aufgabe nicht erfüllen konnte (s. [Par 87], S. 9)[23]. Parnas wollte im Gegensatz zu seinen Kolleg(inn)en im Sinne seines Berufsethos „darauf bedacht sein, das eigentliche Problem anzugehen und nicht einfach meinen Vorgesetzten kurzfristig zufriedenzustellen“ (s. [Par 87], S. 3). Konsequenterweise kündigte er seine Mitarbeit an SDI auf, da das Problem nicht zu lösen ist. Parnas erfüllte damit auch die Forderung des Arbeitskreises 8.3.3 („Grenzen eines verantwortbaren Einsatzes von Informationstechnik“) der Gesellschaft für Informatik:

> „Sehr wichtig ist das Verhalten der Entwickelnden gegenüber dem für die Anwendung verantwortlichen Management. Schwierigkeiten pflegen hierbei in der Regel dadurch zu entstehen, daß der Entwickler zuviel verspricht oder das Management zuviel erwartet. So kann eine realitätsgerechte Prüfung eines Software-Produkts mißlingen, wenn der verantwortliche Manager z. B. auf Grund irreführender Lektüre oder auf Grund von Messevorführungen zu hohe

[22] vgl. Präambel der „Ethischen Leitlinien der Gesellschaft für Informatik“, *Inf.-Spektrum*, Band 16, Heft 4, 1993, S. 239 f.; s. auch [Lut 94], [Weh 94].

[23] Das Erreichen des Ziels, Angriffswaffen überflüssig zu machen, setzt voraus, daß die Zuverlässigkeit des Systems schon vor dem ersten Einsatz bekannt ist. Dies ist aber weder durch Testen noch durch Simulation möglich. Der Computereinsatz ist also nicht zu verantworten, da es nach einem Versagen von SDI im Ernstfall keine Möglichkeiten zur Fehlerkorrektur gibt (s. [Flo 85], S. 4).

> Erwartungen an das System hat und der zuständige Experte, mit Rücksicht auf seine Position und seine Karriere, Grenzen des Systems mit einigen sybillinischen Worten andeutet, jedoch nicht präzise benennt. Entwickler müssen sich daher die Grenzen des jeweils Verantwortbaren deutlich machen und derart erkannte Unsicherheiten und Risiken gegenüber dem Management zur Sprache bringen" (s. [Co& 88], S. 698)

Die Auseinandersetzung mit Scheinlösungen, die vordergründig als verantwortbarer Entwurf erscheinen mögen, ist ebenfalls erforderlich. Einige Verteidiger/innen von SDI, die sich als Eastport-Gruppe bezeichneten, schlugen z. B. folgendes vor: Die von Parnas festgestellten Softwareprobleme seien durch „lose Koordination" zu lösen. Dieses Konzept entspricht in etwa der von C. Floyd aufgestellten Forderung, sich wegen der Undurchschaubarkeit großer Programmsysteme bewußt auf lose gekoppelte kleine Systeme zu beschränken (s. [Flo 85], S. 4). Parnas weist aber nach, daß ein SDI-System mit loser Koordination die eigentliche Aufgabe, Tausende angreifender Raketen zu vernichten, nicht erfüllen kann, da nur bei zentraler Koordination die Verteidigungswaffen effektiv eingesetzt werden können (s. [Par 87], S. 7 f.).

Auch bei Projekten, die nicht so spektakulär sind wie SDI, sollten die Entwicklerinnen ihre Verantwortung gegenüber den Käuferinnen bzw. Benutzerinnen wahrnehmen und Software nach bestem Wissen und Gewissen entwickeln und testen und die notwendige Zeit dafür gegenüber Projektmanagement und Käuferinnen mit dem Zitat von H. Zemanek verteidigen:

> „Unsere Nanosekunden verleiten alle Beteiligten und Unbeteiligten zu dem Irrglauben, man könnte im Computer und mit Computerhilfe auf schnellstem Wege wirken und korrigieren. Gewiß, es gibt superschnelle Prozesse — aber nicht im Entwurfsvorgang und schon gar nicht für die Korrektur von Systemkonzepten. Es kann zu spät werden, einen mangelhaften Entwurf zu reparieren. Es kann zu spät werden, ihn zu ersetzen." (s. [Zem 78], S. 178)

3 Qualitätsmanagement-, Prüf- und Testmethoden im Überblick

Dieses Buch behandelt schwerpunktmäßig Software-Testmethoden. Diese Methoden bilden aber nur einen Teil der Maßnahmen, die für die Sicherung der Qualität von Software wichtig sind. Daher wird das Testen in Kapitel 3.2 in den Bereich des Qualitätsmanagements[1] eingeordnet und eine Abgrenzung zu den anderen Maßnahmen des Qualitätsmanagements gezogen. Vorher wird in Kapitel 3.1 das allgemeine Testziel und die grundsätzliche Vorgehensweise beim Testen betrachtet. Ein Überblick über die statischen und dynamischen Prüfmethoden, insbesondere über Teststrategien, Testansätze, Testmethoden, Testaktivitäten und Testphasen schließt sich in den Kapiteln 3.3 bis 3.6 an. Ein Einblick in Verfahren zur Bewertung und Verbesserung des Qualitätsmanagements, zur Bestimmung der Testgüte und zur Bestimmung des Testendes folgt in Kapitel 3.7 und 3.8.

Dieses Kapitel bietet einen groben Überblick über Testmethoden und zudem einen Blick auf Zusammenhänge und Randgebiete des Testens.

3.1 Ziel, Intention und Vorgehensweise des Testens

Beim Testen muß einerseits über das grundsätzliche Ziel des Testens und andererseits über die grundsätzliche Vorgehensweise des Testens Klarheit herrschen.
Für das grundsätzliche Ziel des Testens existieren verschiedene Begriffsbildungen mit verschiedenen Präzisionsgraden, die im folgenden vorgestellt und diskutiert werden:

„**Testen** ist jede Aktivität, die unseren Glauben bzw. unsere Zuversicht erhöht, daß das Programm sich so verhält, wie es sich verhalten soll.“ (Zitat aus [Het 72b], Hervorhebung von mir, E. R.)

Problematisch ist in obiger Formulierung der Passus „jede Aktivität, die unseren Glauben ... erhöht ... “. Damit könnten auch Werbemaßnahmen einer Firma gemeint sein, die die Software etwa mit dem Spruch „unsere Software wurde von den Redakteuren der DATAMATION zum Produkt des Jahres gewählt“ anpreist. (Siehe dazu auch das Stichwort „Zertifizieren“ in Kapitel 2.6.)

Präziser als obige Formulierung des Begriffs „Testen“ sind die folgenden Begriffsbildungen:

[1] früher auch **Qualitätssicherung** genannt

„Der **Zweck des Testens** ist es zu zeigen, daß ein Programm die geforderten Funktionen korrekt ausführt.“ (Zitat aus [Mye 79], Hervorhebung von mir, E. R.)

„Der **Testvorgang** soll sicherstellen, daß ein Programm zuverlässig die (vorgegebene) Funktion realisiert.“ (Zitat aus [ABC 82], Hervorhebung von mir, E. R.)

Beide Begriffsbildungen gehen davon aus, daß die Anforderungsspezifikation als eine Menge von Funktionen beschrieben ist. Dies ist aber ein eingeengter Begriff von Anforderungsspezifikation. Neben der funktionalen Korrektheit sind auch andere Eigenschaften von Bedeutung wie z. B. die Einhaltung von Zeitbedingungen bezüglich der Abarbeitung von Funktionen eines Programms.

Folgende Begriffsbildung soll daher die für dieses Buch verbindliche Definition des Testziels sein:

Definition 3.1.1 *(s. [Mil 78])*
Allgemeines **Ziel des Programmtestens** *ist es, die Qualität von Softwaresystemen durch systematisches Ausführen der Software unter sorgfältig kontrollierten Umständen zu erhöhen.*

Bei dieser Definition ist der Einwand zur ersten Begriffsbildung (aus [Het 72b]) ausgeräumt und die Einengung auf funktionale Korrektheit in den nächsten beiden Formulierungen (aus [Mye 79] und [ABC 82]) entfallen.
Die Definition 3.1.1 sagt auch etwas zur Vorgehensweise des Testens aus: „systematisch“ und unter „sorgfältig kontrollierten Umständen“ soll das Testen erfolgen. Dabei können zwei Arten von Vorgehensweisen unterschieden werden: die demonstrative und die destruktive. Bei der **demonstrativen** Vorgehensweise werden Testdaten mit dem Ziel ausgewählt, die Fehlerfreiheit des Programms nachzuweisen. Bei der **destruktiven** Vorgehensweise ist dagegen das Auffinden von Fehlern das primäre Ziel des Testens:

Definition 3.1.2 *(s. [Mye 79])*
Testen *ist der Prozeß, ein Programm mit der Intention auszuführen, Fehler zu finden.*

Eine Testdatenmenge wird also gemäß Def. 3.1.2 nicht mit der Intention zusammengestellt, die Fehlerlosigkeit eines Programms nachzuweisen. Damit wird folgender Nachteil der demonstrativen Vorgehensweise vermieden: Die Testperson kann sich leicht in trügerischer Sicherheit wiegen, da diese Vorgehensweise nicht die Auswahl fehlerprovozierender Testdaten unterstützt.
Eine beliebige Vorgehensweise führt nicht unbedingt zu dem in der Definition 3.1.1 festgelegten Ziel, vielmehr ist beim Testen folgendes notwendig:

Definition 3.1.3 *(vgl. [He& 84], Kap. 5.3.2; ANSI/IEEE Std. 610.12 von 1990)*
Testen *ist eine (i. allg. stichprobenartige) Ausführung von Experimenten*[2] *mit dem Prüfgegenstand (ein Modul, Teilsystem oder Programmsystem) unter spezifizierten Bedingungen.*

Zusammenfassend sei festgehalten: Die Definitionen 3.1.1, 3.1.2 und 3.1.3 sind die für dieses Buch gültigen Definitionen für das grundsätzliche Ziel und die grundsätzliche Intention und Vorgehensweise beim Testen (im engeren Sinn).

Da Fehlerfreiheit ein spezielles Qualitätsmerkmal ist (vgl. Kapitel 2.2, S. 31), wird in Kapitel 3.2 die Rolle des Testens innerhalb des erweiterten Rahmens des Qualitätsmanagements betrachtet. Dabei zeigt sich, daß Testen im engeren Sinn von Definition 3.1.3 nicht die einzige Möglichkeit ist, die Fehlerreduzierung oder sogar Fehlerfreiheit eines Softwareprodukts zu erreichen.
Um die geforderte Qualität der Software zu bestätigen oder zu widerlegen, ist beim Testen eine systematische Auswahl relevanter Testdaten notwendig. Dazu müssen geeignete Testkriterien (inklusive Meßverfahren) bereitgestellt werden, mit denen die Güte der durchgeführten Tests ermittelt werden kann. Damit werden sich (nach einem Überblick in Kapitel 3.3) die Kapitel 3.4 bis 3.6 beschäftigen.

3.2 Einordnung des Gebiets Testen in das Qualitätsmanagement

3.2.1 Aufgabe des Qualitätsmanagements

Aufgabe des Qualitätsmanagements ist es, definierte Qualitätsmerkmale für ein Softwareprodukt[3] zu erreichen. Die Grundbegriffe Qualität und Merkmal sind dabei folgendermaßen definiert:

Definition 3.2.1.1
Qualität *ist die Gesamtheit von Eigenschaften und Merkmalen eines Produktes oder einer Tätigkeit, die sich auf deren Eignung zur Erfüllung festgelegter oder vorausgesetzter Erfordernisse beziehen.*

Definition 3.2.1.2
Ein **Merkmal** *ist eine Eigenschaft, die das Unterscheiden von Einheiten einer Gesamtheit entweder in qualitativer oder quantitativer Hinsicht ermöglicht.*

[2] (altfranzösisch) test (von lateinisch testum = Geschirr, Schüssel) ist ein irdener Topf oder Tiegel für *alchimistische* Experimente

[3] Nach DIN ISO/IEC 12119 von August 1995 wird „Softwareprodukt" auch „Software-Erzeugnis" genannt.

Qualitätsmerkmale werden mit Hilfe von hierarchisch strukturierten Begriffsbäumen beschrieben und präzisiert. Diese Begriffsbäume unterteilen die Qualitätsmerkmale in feinere, untergeordnete Qualitätsmerkmale. Solche Begriffsbäume werden auch als **Qualitätsmodell** bezeichnet.
Das Ziel der Gliederung von Qualitätsmerkmalen in immer feinere Merkmale ist die Ableitung von Qualitätsmerkmalen aus quantitativen Kenngrößen, die leicht zu messen und zu bestimmen sind. Quantitative Kenngrößen sind z. B. die Anzahl der Systemebenen, die Anzahl der Knoten einer Baumhierarchie, die Anzahl von Modulen und die Anzahl der Parameter pro Schnittstelle. Leider ist es nicht möglich, jedes Qualitätsmerkmal grundsätzlich auf quantitative Größen abzubilden. In die Bewertung der Benutzbarkeit z. B. fließen oft subjektive Kriterien ein, d. h. allumfassende und meßbare Kenngrößen zur Bewertung der Benutzbarkeit können sicherlich nicht angegeben werden[4]. Allerdings bieten die Begriffsbäume eine gute Orientierungshilfe zur Bewertung von Qualitätsmerkmalen. Im folgenden wird eine mögliche Klassifizierung von Qualitätsmerkmalen vorgestellt.

DEFINITION 3.2.1.3 (PRODUKTQUALITÄT [FÜR SOFTWARE])
Die **Produktqualität** *setzt sich aus folgenden Qualitätsmerkmalen zusammen:*

1. **Funktionalität:** *Eine Menge von Merkmalen, die sich auf das Vorhandensein einer Menge von Funktionen und auf deren festgelegte Merkmale beziehen. Die Funktionen sind jene, die die festgelegten oder vorausgesetzten Erfordernisse erfüllen.*

2. **Zuverlässigkeit:** *Eine Menge von Merkmalen, die sich auf die Fähigkeit der Software beziehen, ihr Leistungsniveau unter festgelegten Bedingungen über einen festgelegten Zeitraum zu bewahren.*

3. **Benutzbarkeit:** *Eine Menge von Merkmalen, die sich beziehen*

   - *auf den Aufwand, der zur Benutzung erforderlich ist,*
   - *auf die individuelle Bewertung einer solchen Benutzung durch eine festgelegte oder vorausgesetzte Gruppe von Benutzern.*

4. **Effizienz:** *Eine Menge von Merkmalen, die sich auf das Verhältnis zwischen dem Leistungsniveau der Software und dem Umfang der eingesetzten Betriebsmittel unter festgelegten Bedingungen beziehen.*

5. **Änderbarkeit:** *Eine Menge von Merkmalen, die sich auf den Aufwand beziehen, der zur Durchführung vorgegebener Änderungen (Korrekturen, Verbesserungen oder Anpassungen) notwendig ist.*

[4] Mit der internationalen Norm DIN EN ISO 9241, Teile 10 bis 17, liegt aber immerhin eine Norm für die Gestaltung der Benutzungsschnittstelle vor.

6. **Übertragbarkeit:** *Eine Menge von Merkmalen, die sich auf die Eignung der Software beziehen, von einer Umgebung in eine andere übertragen zu werden.*

Die Merkmale Änderbarkeit und Übertragbarkeit lassen sich nicht durch Testen überprüfen, da man den „Aufwand“ bei der Änderbarkeit und die „Eignung“ für die Übertragbarkeit nicht durch Ausführen der Software testen kann. Das Merkmal Effizienz läßt sich durch Testen nur dann überprüfen, wenn z. B. beim Zeitverhalten genau definiert ist, welcher Durchsatz und welche Antwort- und Verarbeitungszeiten bei bestimmten Hardwarekonfigurationen gefordert werden. Ähnliches gilt für den Test der Benutzbarkeit. Von Benutzern und Entwicklern können allerdings aus subjektiver Sicht im Rahmen gewisser Richtlinien (z. B. [Op& 88]) Testdaten zur Überprüfung der „ergonomischen“ Benutzbarkeit (Benutzungsfreundlichkeit), welche z. B. die Effizienz mit einschließt, ausgewählt werden. Die einzigen Merkmale, welche durch Testen systematisch und objektiv unterstützt werden können, sind die Zuverlässigkeit und die Funktionalität. Im folgenden werden daher die Qualitätsmerkmale Funktionalität und Zuverlässigkeit etwas genauer betrachtet.

Definition 3.2.1.4 (Merkmale der Funktionalität)
Die **Funktionalität** *ist abhängig von der Ausprägung folgender Qualitätsmerkmale eines Softwareprodukts:*

1. **Angemessenheit:** *Ein Merkmal von Software, das sich auf das Vorhandensein und die Eignung einer Menge von Funktionen für spezifizierte Aufgaben bezieht.*

2. **Richtigkeit:** *Ein Merkmal von Software, das sich auf das Liefern der richtigen oder vereinbarten Ergebnisse oder Wirkungen bezieht.*

3. **Interoperabilität:** *Ein Merkmal von Software, das sich auf ihre Eignung bezieht, mit vorgegebenen Systemen zusammenzuwirken.*

4. **Ordnungsmäßigkeit:** *Ein Merkmal von Software, das bewirkt, daß die Software anwendungsspezifische Normen oder Vereinbarungen oder gesetzliche Bestimmungen und ähnliche Vorschriften erfüllt.*

5. **Sicherheit:** *Ein Merkmal von Software, das sich auf ihre Eignung bezieht, unberechtigten Zugriff (sowohl versehentlich als auch vorsätzlich) auf Programme und Daten zu verhindern.*

Das Merkmal „Angemessenheit“ läßt sich z. T. durch einen (System-)Test überprüfen (genaueres siehe Kapitel 13.6) — jedenfalls das (Nicht-)Vorhandensein gewisser Funktionen. Die (Nicht-)Erfüllung des Merkmals „Richtigkeit“ wird durch einen Test mit Soll-Ist-Vergleich festgestellt (genaueres siehe Abschnitt 3.5.4). Ob „Interoperabilität“ (nicht) vorliegt, kann z. T. mit einem Schnittstellentest aufgedeckt werden. Die „Sicherheit“ kann dadurch bestätigt oder widerlegt werden, daß Personen mit Hacker-Fähigkeiten das Gesamtsystem „testen“. Die „Ordnungsmäßigkeit“

kann dagegen i. allg. nur durch Vergleich der Software und/oder ihrer Ergebnisse mit den entsprechenden Vorgaben festgestellt werden, also nicht nur durch Testen und einfaches Betrachten der Ausgaben.

Definition 3.2.1.5 (Merkmale der Zuverlässigkeit)
Die **Zuverlässigkeit** *ist abhängig von der Ausprägung folgender Qualitätsmerkmale eines Softwareprodukts:*

1. **Reife:** *Ein Merkmal von Software, das sich auf die Häufigkeit von Versagen durch Fehlzustände in der Software bezieht.*
2. **Fehlertoleranz:** *Ein Merkmal von Software, das sich auf ihre Eignung bezieht, ein spezifiziertes Leistungsniveau bei Nicht-Einhaltung ihrer spezifizierten Schnittstelle oder bei Software-Fehlern zu bewahren.*
3. **Wiederherstellbarkeit:** *Ein Merkmal von Software, das sich auf die Möglichkeit bezieht, bei einem Versagen mit angemessenem (Zeit-)Aufwand ihr Leistungsniveau wiederherzustellen und die direkt betroffenen Daten wiederzugewinnen.*

Jedes Untermerkmal der Zuverlässigkeit kann durch Testen überprüft werden. Voraussetzung für einen Test der Merkmale ist aber eine genaue Festlegung der Anforderungen zur Erreichung dieser Merkmale. So kann z. B. die Fehlertoleranz eines Systems nur dann getestet werden, wenn die zu berücksichtigenden Verletzungen der Schnittstellen und das geforderte Leistungsniveau des Systems genau festgelegt sind.

3.2.2 Maßnahmen des Qualitätsmanagements

Mit Maßnahmen des Qualitätsmanagements werden die gewünschten Qualitätsmerkmale herbeigeführt oder überprüft. Dabei werden konstruktive und analytische Qualitätsmanagementmaßnahmen unterschieden.

Definition 3.2.2.1
Konstruktive Qualitätsmanagementmaßnahmen *sind Methoden, Sprachen und Werkzeuge, die dafür sorgen, daß das entstehende Produkt von vornherein bestimmte Eigenschaften besitzt.*

Die Verwendung abstrakter Datentypen und die modulare Erstellung von Software unterstützt z. B. die Qualitätsmerkmale „Änderbarkeit" und „Übertragbarkeit". Die Verwendung formaler Spezifikationen unterstützt das Qualitätsmerkmal „Richtigkeit" und damit die „Funktionalität" des Systems.

Da trotz der konstruktiven Qualitätsmanagementmaßnahmen Fehler nicht ausgeschlossen sind, müssen die Softwareprodukte noch nachträglich überprüft werden:

DEFINITION 3.2.2.2
Analytische Qualitätsmanagementmaßnahmen *sind diagnostische Maßnahmen, die der nachträglichen Überprüfung gewünschter Qualitätsmerkmale dienen.*

Die analytischen Qualitätsmanagement- bzw. Prüfverfahren lassen sich nach folgenden Gesichtspunkten unterscheiden:

(a) Der Ablauf der Prüfung kann **statisch** (in der Reihenfolge, in welcher der Prüfgegenstand aufgeschrieben wurde) oder **dynamisch** (in der Reihenfolge der Ausführung des Prüfgegenstands) sein.

(b) Die Vorgehensweise bei der Prüfung kann **informell** (bei menschlicher Begutachtung), **formal** (beim Einsatz mathematischer/logischer Verfahren) oder **experimentell** (beim Beobachten des Verhaltens des Prüfobjekts) sein.

(c) Prüfungen können **vollständig** oder **stichprobenartig** sein.

(d) Der Prüfgegenstand und/oder der Gegenstand, gegen den geprüft wird [5], kann **formal** , teilweise formal oder **informell** beschrieben sein.

(e) Von der Beschreibungsform und der Vorgehensweise hängt es meistens ab, ob die Prüfung vollständig, teilweise oder nicht **automatisiert** erfolgen kann, d.h. ob Rechnerunterstützung möglich und sinnvoll ist.

3.3 Statische und dynamische Prüfverfahren

Die folgende Übersicht ist nach den Aspekten „Ablauf“ und „Vorgehensweise“ gegliedert. Die zusätzlichen Aspekte **Verantwortung** für die Prüfung (Entwickler oder Auftraggeber) und **Gegenstand** der Prüfung (Modul, Komponente, Software-System sowie Entwicklungs- und Qualitätsmanagementsystem) werden erst in Kapitel 3.6 und 3.7 behandelt.

3.3.1 Statische Prüfverfahren

Statische Verfahren umfassen folgende **informelle** Verfahren:

- Schreibtischtest durch eine Person,
- Audits, Reviews und Inspektionen durch Personengruppen.

[5] z. B. Programmcode gegen Programmentwurf

Diese menschliche Begutachtung wird meist vollständig durchgeführt, erfordert daher einen hohen Aufwand, deckt aber frühzeitig Fehler auf und enthält die sofortige Fehlerlokalisierung.
Inspektionen beinhalten beispielsweise genaues Gegenlesen, mündliches Erklären des Programms gegenüber einer Person oder mehreren Personen, Überprüfung des Programms anhand von Checklisten usw. (genaueres s. Kapitel 12.1).

Die **formalen** statischen Verfahren umfassen Verfahren, die sich auf Teilaspekte konzentrieren und automatisch durchführbar sind, und vollständige Verfahren. Im Einzelnen sind dies:

Syntaxanalyse
Dabei werden syntaktische Konsistenzbedingungen überprüft, z. B. bei Schnittstellen zwischen einzelnen Softwarekomponenten. Diese Überprüfungen werden oft schon vom Übersetzungsprogramm (Compiler) durchgeführt.

Datenflußanalyse
Dies beinhaltet die Überprüfung des Datenflusses eines Programms oder der Spezifikation, z. B. Prüfung auf Datenflußanomalien wie die Referenzierung einer Variablen mit undefiniertem Wert (genaueres s. Kapitel 12.2).

Kontrollflußanalyse
Die Kontrollflußanalyse überprüft den Kontrollfluß eines Programms oder der Spezifikation, z. B. werden offensichtlich unendliche Schleifen oder nichterreichbare Codestücke aufgedeckt (genaueres s. Kapitel 12.2).

Formale Verifikation
Bei der formalen Verifikation wird mit mathematisch-logischen Verfahren die Korrektheit des Programms bzgl. der formalen Spezifikation gezeigt, d.h. ein vollständiger Beweis erbracht. Zuerst wird bewiesen, daß das Programm richtige Ergebnisse liefert, wenn es anhält. Für einen totalen Korrektheitsbeweis ist anschließend noch zu zeigen, daß das Programm nicht in eine endlose Schleife gerät. Der gesamte Beweis kann informell (per Hand) oder formal (mit einem automatischen Theorembeweiser) durchgeführt werden. In beiden Fällen gibt es gravierende theoretische und praktische Probleme (genaueres s. Kapitel 12.4).

Die formalen statischen Verfahren dienen außerdem dazu, Informationen aus der Programmstruktur abzuleiten, die beim Testen als Informationsbasis gebraucht werden. Dies sind z. B. der Kontrollflußgraph und der Datenflußgraph eines Programms (genaueres s. Kapitel 7 und 8).

3.3.2 Dynamische Prüfverfahren

Bei den dynamische Prüfverfahren lassen sich wiederum informelle und formale Verfahren unterscheiden.

Zu den **informellen** Verfahren gehört das manuelle Durchgehen eines Dokumentes mit Testfällen (**walkthrough**). Dabei wird gegenüber dem (vollständigen) statischen informellen Verfahren der Inspektion einerseits Aufwand gespart, andererseits werden evtl. nicht alle Programmzweige angesprochen, so daß nicht alle Fehler aufgedeckt werden können.
Zu den **formalen** Verfahren gehören die symbolische Ausführung und die Ausführung mit konkreten Daten, das Testen im engeren Sinn:
Bei der **symbolischen Ausführung** wird das Programmteil mit symbolischen Werten durchgerechnet. Die Sequenz $X := Y + Z;\ U := X * Y$ ergibt dabei z. B. (mit den symbolischen Anfangswerten x und y für X und Y) die Berechnungsfolge $X = y+z$ und $U = (y+z)*y = y^2+yz$. Bei Entscheidungen im Programm, z. B. bei *if* $U > 0$ *then* ...*else*..., muß für jeden Zweig eine eigene symbolische Rechnung durchgeführt werden, wobei die Entscheidungsbedingung der sogenannten **Pfadbedingung** hinzugefügt wird. Im Beispiel ist die Bedingung für den *then*-Zweig $U > 0$, d. h. nach obiger Sequenz $y^2+yz > 0$, und für den *else*-Zweig $U \leq 0$, d. h. $y^2+yz \leq 0$. Bei Schleifen kann nur eine vorher begrenzte oder interaktiv festgelegte Anzahl von Durchläufen berechnet werden. Am Ende der Berechnung erhält man für den ausgeführten Pfad im Programm die symbolisch berechneten Werte für die Variablen und eine Pfadbedingung, die angibt, unter welchen Bedingungen der Pfad ausgeführt wird. Wenn die Semantik der Programmiersprache formal beschrieben ist, kann eine symbolische Ausführung im Prinzip automatisch erfolgen. Die Pfadbedingung und die berechneten Variablenwerte können nur durch menschliche Begutachtung mit der Spezifikation verglichen werden, wobei fehlerhafte Ausdrücke — wie im Programm selbst — nicht immer sicher erkannt werden (genaueres s. Kap. 12.3).
Testen (im engeren Sinn) ist eine dynamische Prüfung der Software durch experimentellen Ablauf in der realen Umgebung (vgl. Definition 3.1.3). Wegen der Probleme oder Unzulänglichkeiten der statischen Verfahren und der symbolischen Ausführung ist das Testen unverzichtbar. Wünschenswert ist ein **idealer Test**, d. h. ein Test, der genau dann ein Fehlverhalten aufzeigt, wenn die Software tatsächlich einen Fehler enthält. Ein solch idealer Test existiert zwar, kann aber nur durch Zufall gefunden werden (vgl. Kap. 2.4). Da ein theoretisch möglicher **erschöpfender** Test aller Eingabekombinationen praktisch undurchführbar ist (vgl. Kap. 2.4), muß also mit möglichst idealen **Stichproben** [6] getestet werden.
Da ein Softwarefehler nur zu einem Fehlverhalten führt, wenn die fehlerhafte(n) Anweisung(en) durch die Testdaten ausgeführt werden, sind nur passend ausgewählte Testdaten zum Aufdecken der Fehler geeignet. Die Menge der Testdaten sollte dabei bei minimaler Testdatenanzahl eine maximale Menge von Fehlern aufdecken. Da dies konkurrierende Anforderungen sind, müssen geeignete **Testkriterien** für Testdatenmengen formuliert werden. Diese Kriterien (s. Kapitel 3.4) sollten sich ei-

[6] Stichprobe ist — historisch gesehen — ein Begriff aus dem Hüttenwesen (Abstich aus dem flüssigen Eisenerz im Hochofen machen). Dies reicht als Qualitätsüberprüfung, da die Materialeigenschaften im gesamten Hochofen nur sehr gering variieren. Bei einer digitalen Software ist das Verhalten aber diskontinuierlich, daher reicht *eine* Stichprobe nicht aus.

gentlich an den (im konkreten Fall unbekannten) Fehlern orientieren, können sich aber an den (bekannten) fehleranfälligen Konstrukten in der Software orientieren.

Die grundsätzlich möglichen Teststrategien und existierende Testansätze werden im folgenden Kapitel betrachtet.

3.4 Grundsätzliche Teststrategien und Testansätze

Bei einem systematischen Test auf Fehler müssen immer zwei Dokumente gegeneinander getestet werden, z. B. die Implementation (das Programm in Quellsprache) gegen die Spezifikation[7]. Die für einen solchen Test jeweils relevanten Dokumente werden **Testbasen** genannt. Abhängig von der Wahl der Testbasen sind die folgenden grundsätzlichen Strategien möglich: die spezifikationsorientierte oder die implementationsorientierte Vorgehensweise. Beide Teststrategien haben verschiedene Vorteile und Nachteile (s. Abschnitte 3.4.1 und 3.4.2) und ein gemeinsames Problem: Fehler in der Spezifikation können durch einen Test meistens nicht entdeckt werden. Ist nämlich bei einer fehlerhaften Spezifikation auch die Implementation fehlerhaft (also der Spezifikation nach richtig), so kann kein Fehler mittels Testen festgestellt werden. Das Problem wird damit verlagert: Es muß nachgewiesen werden, daß die Spezifikation die Anforderungen korrekt wiedergibt. Dies ist mit formalen Mitteln nur in eingeschränktem Umfang möglich. Es kann z. B. nur für einzelne Funktionen nachgewiesen werden, daß sie vollständig und widerspruchsfrei definiert sind, nicht aber, ob alle angeforderten Funktionen und Daten definiert sind[8].

3.4.1 Spezifikationsorientierter Test

Beim spezifikationsorientierten Test werden die Testfälle bzw. Testdaten durch Analyse der Spezifikation[9] erzeugt, ohne die interne Struktur des Programms zu berücksichtigen. Die Güte der erstellten Testdaten ist abhängig von der semantischen Aussagekraft der Spezifikation und von den angewandten Verfahren zur Testdatenauswahl auf Basis der Spezifikation. Der erste Punkt ist der entscheidende: angenommen, in der Spezifikation sind nur die Schnittstellen der Funktionen formal beschrieben (Wertebereich der Ein- und Ausgabeparameter usw.) und die Semantik der Funktionen nur informell. Dann können auch nur solche Testdaten systematisch erstellt werden, die die Korrektheit der Grenzen der Ein- und Ausgabebereiche der Funktion testen, aber keine Testdaten, die die Semantik der Funktion berücksichtigen (Einzelheiten siehe Abschnitt 3.4.3).

[7] Mängel — wie „Programmabsturz“ und offensichtliches Fehlverhalten — können natürlich auch entdeckt werden, wenn nur das lauffähige Programm vorliegt.

[8] im Unterschied zum Merkmal „Angemessenheit“ in der Definition (3.2.1.4) der Funktionalität, das sich auf *spezifizierte* Funktionen bezieht

[9] Anstelle der Spezifikation können auch (Benutzer-)Handbücher verwendet werden, z. B. beim Abnahmetest oder Beta-Test (genaueres siehe Kapitel 13.6).

Vorteile des spezifikationsorientierten Tests

- Nicht implementierte, aber spezifizierte Teile eines Programms können entdeckt werden.
- Ist eine aussagekräftige Spezifikation vorhanden, so können auch Testdaten zum Test semantischer Eigenschaften systematisch erstellt werden. (Globale semantische Eigenschaften sind z. B. formal aufgestellte Integritätsbedingungen bezüglich der verwalteten Daten des Systems.)
- Portierungen werden unterstützt. Die einmal auf Basis der Spezifikation erstellten Testdaten können wegen ihrer Implementationsunabhängigkeit zum Test verschiedener Portierungen benutzt werden.

Nachteile des spezifikationsorientierten Tests

- Zusätzlich implementierte Fallunterscheidungen, die in der Spezifikation nicht vorgesehen sind, können nur durch Zufall entdeckt werden.
- Ist die Spezifikation wenig aussagekräftig, so sind auch die Testdaten entsprechend unrepräsentativ. Hier wäre eine Transformation der Spezifikation in eine aussagekräftige, formalere Notation notwendig.

Anwendung auf nebenläufige Softwaresysteme
Die bestehenden spezifikationsorientierten Testmethoden für sequentielle Softwaresysteme können für den Test der sequentiellen Teile eines nebenläufigen Systems genutzt werden. Zum Test von Eigenschaften nebenläufiger Softwaresysteme — wie z. B. Verklemmung durch inkorrekte Synchronisation — sind sie aber nicht geeignet. Dafür müssen besondere Methoden entwickelt werden (genaueres s. Kapitel 14.4).

3.4.2 Implementationsorientierter Test

Beim implementationsorientierten Test werden die Eingabedaten durch strukturelle Analyse des Programms gewonnen. Die Menge dieser Daten wird also vollständig von der internen Struktur des Programms abgeleitet. Beim implementationsorientierten Test ist i. allg. der Kontroll- und Datenfluß des Programms die Informationsbasis. Darauf beziehen sich Überdeckungskriterien, die eine relevante Teilmenge aller möglichen Programmpfade definieren. Eine hinreichende Testmenge ist dann eine Menge von Eingabedaten (zusammen mit den Solldaten, s. Kap. 3.5), die alle über das Überdeckungskriterium bestimmten Programmpfade ausführen. Spezielle Techniken des implementationsorientierten Tests betrachten außerdem die Struktur der einzelnen Anweisungen des Programms genauer (s. Abschnitt 3.4.3).

Vorteile des implementationsorientierten Tests

- Eine automatische Unterstützung des Tests ist möglich, da das Programm eine Testbasis mit einer wohldefinierten Syntax und Semantik ist.
- Mit Hilfe des implementationsorientierten Tests können auch Programme ohne Vorhandensein einer ausreichenden Spezifikation „getestet" werden. Wenn die Spezifikation nur unvollständig oder nur in den Köpfen einiger Entwickelnder vorhanden ist, kann die Programmstruktur als Testbasis dienen, um wenigstens einen teilweise systematischen Test durchführen zu können. Dies ist allerdings kein objektiver Test, da anstatt einer Spezifikation die subjektive Vorstellung der Testperson vom gewünschten Programm als Entscheidungsgrundlage dafür dient, ob der Test einen Fehler aufgedeckt hat oder nicht.
- Durch einen implementationsorientierten Test können zusätzliche Fallunterscheidungen, die in der Spezifikation nicht vorgesehen sind, sowie notwendige Verfeinerungen und Implementationsdetails getestet werden.

Nachteile des implementationsorientierten Tests
Nicht implementierte Teile einer Spezifikation können durch den implementationsorientierten Test nur zufällig entdeckt werden.

Anwendung auf nebenläufige Systeme
Bei nebenläufigen Systemen existiert aufgrund der nichtdeterministischen Eigenschaften der Systeme eine viel größere Menge von möglichen Anweisungsfolgen als bei sequentiellen Systemen. Im Prinzip können die sequentiellen Methoden zur Pfadauswahl und damit zur Testdatenerzeugung herangezogen werden. Die oben erwähnten Überdeckungsmaße würden aber eine viel zu große Menge von Testdaten erfordern. Deshalb können sequentielle Überdeckungmaße nur für die Überprüfung der sequentiell durchlaufenen Teile eines nebenläufigen Systems eingesetzt werden. Für die Überprüfung, ob nebenläufig durchlaufene Teile eines Systems korrekt kooperieren, sind andere Überdeckungsmaße erforderlich. Sie müssen nicht nur Testdaten zum Auffinden von Rechenfehlern bestimmen, sondern auch Testdaten zur Aufdeckung nicht gewünschter nebenläufiger Eigenschaften, wie z. B. Synchronisationsfehler (genaueres siehe Kapitel 14.4).

3.4.3 Überblick über bestehende Testansätze

Bei einer groben Einteilung lassen sich zwei Testansätze unterscheiden: der funktionale Testansatz („Black-Box"-Test) und der strukturelle Testansatz („White-Box"- oder „Glass-Box"-Test). Diese beiden Testansätze decken sich im Wesentlichen mit den in den Abschnitten 3.4.1 und 3.4.2 vorgestellten Teststrategien. Im Unterschied zum funktionalen Testansatz betrachtet die spezifikationsorientierte Teststrategie

allerdings nicht nur das funktionale Verhalten eines Programms, sondern auch nichtfunktionale Eigenschaften wie z. B. die Einhaltung von Zeitbedingungen.

Zum funktionalen Testansatz gehören die folgenden Testmethoden:

- Datenbereichsbezogenes Testen (genaueres s. Kapitel 4.2)
- Testen von funktionalen Formen wie Sequenz, Fallunterscheidung, *while*- oder *repeat*-Schleife (genaueres s. Abschnitt 4.3.1)
- Entwurfsorientertes Testen (genaueres s. Abschnitt 4.3.2)
- Testen auf der Basis von Datenflußdiagrammen, Petrinetzen und endlichen Automaten (genaueres s. Abschnitt 4.3.3)
- Testen von Reihenfolgebedingungen (genaueres s. Kapitel 5.1)
- Testen auf der Basis von algebraischen Spezifikationen (genaueres s. Kapitel 5.2)

Zum strukturellen Testansatz gehören die folgenden Testmethoden:

- Kontrollflußbezogenes Testen (genaueres s. Kapitel 7 und 9.2)
- Datenflußbezogenes Testen (genaueres s. Kapitel 8).
- Ausdrucks-, anweisungs-, datenbezogenes Testen (genaueres s. Kap. 9.1, 9.4)
- Mutationsanalyse (genaueres s. Kapitel 9.3)

Die Testansätze lassen sich den Teststrategien folgendermaßen zuordnen: Die datenbereichsbezogenen Testmethoden und die Testmethoden auf der Basis anderer Spezifikationsformen sind dem funktionalen Testansatz zuzuordnen. Die kontrollflußbezogenen und datenflußbezogenen Testmethoden sind dem strukturellen Testansatz zuzuordnen. Das datenbereichsbezogene Testen kann in angepaßter Form (Betrachtung von Wegbereichen statt Datenbereichen) auch strukturell angewandt werden (genaueres s. Kapitel 16.3). Die Mutationsanalyse kann auch auf Spezifikationen (statt auf Programme) angewandt werden (s. [BuG 85]).

Zur Erhöhung der Zuverlässigkeit eines Tests sollten nicht nur Testmethoden einer einzigen Teststrategie bzw. eines einzigen Testansatzes angewandt werden, sondern verschiedene Strategien und Ansätze ausgewählt und kombiniert werden (genaueres s. Kapitel 16.2 und 16.3). Dabei sollte jeweils mindestens eine spezifikationsorientierte sowie eine implementationsorientierte Testmethode ausgewählt werden. Mit dieser Vorgehensweise können die Nachteile der einen Teststrategie durch die Vorteile der anderen Teststrategie ausgeglichen werden.

3.5 Testablauf

Während des Tests eines Softwareprodukts können verschiedene Aktivitäten unterschieden werden. Bei einer groben Betrachtungsweise sind dies die Testvorbereitung, die Testausführung und die Testauswertung, die zeitlich nacheinander durchgeführt werden. Diese Aktivitäten bilden zusammen einen Testzyklus. Für jede Testphase, also Modultest, Integrationstest und Systemtest, muß jeweils ein Testzyklus durchgeführt werden. Er wird solange wiederholt, bis mit der Testauswertung keine Fehler mehr entdeckt werden und die gewünschte Testgüte erreicht ist.
Bei der Testvorbereitung wird die Testplanung sowie die Erzeugung der Testdaten und der Testrahmen unterschieden.

3.5.1 Testplanung

Die (zuerst durchzuführende) Testplanung legt geeignete Maßnahmen zur Sicherstellung der gewünschten Qualitätsmerkmale fest. Dabei werden auch Maßnahmen definiert, die den Test durch vorbereitende Qualitätsmanagementsmaßnahmen, wie z. B. Syntaxanalyse, Kontrollflußanalyse, Datenflußanalyse und Überprüfung von Programmierrichtlinien, unterstützen. Zu den konstruktiven Maßnahmen gehört z. B. die Festlegung der zu benutzenden Programmiersprache. Zu den analytischen Maßnahmen des Tests (im folgenden **Testmaßnahmen** genannt) gehört die Festlegung der Testmethoden und die Festlegung der Abbruchkriterien für einen Test. Eine gute Testplanung legt auch die Durchführung des Tests fest:

- Bereitstellung von Werkzeugen, Methoden und Richtlinien zum Testen,
- Arbeitsplanung (wer führt welche Testmaßnahmen aus),
- Bereitstellung von geeigneten Ressourcen,
- Bestimmung von Personen, die die Testmaßnahmen kontrollieren und Hilfestellung bei der Durchführung geben (Testmanagement).

3.5.2 Testdatenerzeugung

Diese Aktivität generiert die Testdaten mit den in der Testplanung vorgegebenen Verfahren zur Testfall- oder Testdatenerzeugung. Zu jedem Eingabedatum wird anhand der Spezifikation ein gewünschtes Ergebnis, ein Solldatum, bestimmt.

Im folgenden werden die benutzten Begriffe Testfall, Testdatum, Eingabedatum, Solldatum usw. genauer definiert.

Definition 3.5.2.1
Ein **Testfall** *beschreibt ein Testdatum durch seine relevanten Eigenschaften.*

Eine solche Beschreibung läßt meistens mehrere Testdaten zu einem Testfall zu.

BEISPIEL 3.5.2.1
Ein Testfall besage, daß als Eingabe x ein ganzzahliger Wert aus dem Intervall $18 \leq x \leq 65$ *zu wählen ist. Dann ist jeder der 48 Werte aus der Reihe 18, 19, ..., 65 ein erlaubtes Eingabedatum eines Testdatums.*

DEFINITION 3.5.2.2
Ein **Testdatum** *ist aus einem Eingabe- und einem Solldatum zusammengesetzt.*

DEFINITION 3.5.2.3 (EINGABEDATUM/TESTWERT/EINGABE)
Ein **Eingabedatum** *ist ein Tupel von Werten für alle Eingabevariablen des Programms, wobei zusätzlich der Inhalt der Eingabedateien, der permanenten Dateien und Datenbanken spezifiziert ist. Die Komponenten des Eingabedatums heißen* **Testwerte**. *In manchen Fällen (z. B. bei Dialogprogrammen) wird bei einem Programmablauf eine Folge von Eingabedaten verarbeitet, die einfach* **Eingabe** *heißt.*

Nur bei Programmen, die reine Funktionen berechnen, hängt das Berechnungsergebnis ausschließlich von den Eingabedaten ab. Alle anderen Programme berechnen Ergebnisse, die auch vom inneren Zustand des Programms und seiner permanenten Daten abhängen. Diese Werte müssen also auch in den Test mit eingehen.

Um die Auswertung der Tests zu verbessern, ist es wichtig, als Testdatum nicht nur ein Eingabedatum, sondern auch ein Solldatum anzugeben.

DEFINITION 3.5.2.4
Ein **Solldatum** *ist ein*[10] *gemäß Spezifikation gewünschtes Ergebnis, daß erzeugt werden sollte, wenn das Programm mit einem Eingabedatum ausgeführt wird.*

DEFINITION 3.5.2.5
Ein **Istdatum** *ist das Ergebnis, das erzeugt wird, wenn das (implementierte) Programm mit einem Eingabedatum ausgeführt wird.*

DEFINITION 3.5.2.6
Das **Ergebnis einer Programmausführung** *ist ein Tupel von Werten für alle Ausgabevariablen, wobei außerdem der Inhalt der Ausgabedateien, der permanenten Dateien und Datenbanken mit spezifiziert ist.*

[10] Bei einer funktionalen, eindeutigen Abhängigkeit des Soll-Ergebnisses vom Eingabedatum gibt es nur ein („das") gewünschte(s) Ergebnis. In anderen Fällen — z. B. bei Berechnungen mit Ergebnissen vom Typ Real — gibt es eine relationale Abhängigkeit, d. h. eine Menge von erlaubten Ergebnissen, z. B. alle Werte x mit $0 \leq x \leq 10^{-3}$. In diesem Fall ist als Solldatum diese Menge anzugeben bzw. der Mittelpunkt des Intervalls und die maximal erlaubte Abweichung.

Ein Programm ist nur dann korrekt, wenn Istdatum und Solldatum bei allen Programmausführungen übereinstimmen[11]. Dies ist zu testen.

Bei einem systematischen Test muß der Eingabewertebereich eines Programms in geeignete Teilmengen (Testklassen) unterteilt werden. Die Eingabedatenmenge für das zu testende Programm besteht dann aus einer Stichprobe, d. h. aus mindestens einem Repräsentanten für jede Testklasse (vgl. Def. 3.1.3 auf Seite 49). Zu jedem Eingabedatum wird anhand der Spezifikation ein Solldatum bestimmt, d. h. ein komplettes Testdatum erzeugt. Die Menge dieser Testdaten bildet dann die **Testdatenmenge** für das zu testende Programm.

BEISPIEL 3.5.2.2
Ein Kontenverwaltungsprogramm einer Bank habe folgende Eingabevariablen:

- *Betrag (vom Typ Real*[12] *mit nichtnegativen Werten als erlaubte Eingabe),*
- *Vorgang (vom Typ Character mit den erlaubten Werten „E“ bei Einzahlung und „A“ bei Abheben eines Betrages),*
- *Konto-Identifizierung (vom Typ Integer, abgekürzt „Konto-Id“) sowie*
- *Zugriff auf eine Datenbank, die für jedes Konto (jede „Konto-Id“) den aktuellen Kontostand enthält (vom Typ Real).*

Ein Eingabedatum ist dann z. B. (90.10, „E“, 532700, 350.00), wobei folgende Testwerte vorkommen: Betrag = 90.10, Vorgang = „E“, Konto-Id = 532700 und Kontostand = 350.00 in der Datenbank für diese Konto-Identifizierung (der Rest ist hier ohne Bedeutung). Das Solldatum ist dann Kontostand = 440.10 für Konto-Id = 532700 in der Datenbank. (Der Rest ist irrelevant, sollte aber nicht verändert werden.) Eine spezifikationsorientierte Testklasse enthält z. B. alle Testdaten, die einen nichtnegativen Betrag, den Vorgang „E“, und eine gültige Konto-Id beinhalten, für die es in der Datenbank einen Kontostand gibt.

3.5.3 Testrahmenerstellung

Die Testrahmenerstellung erzeugt die für die Durchführung von Modultests benötigten Treiber und Stellvertreter (engl. stubs). (Nur bei der Durchführung des abschließenden Systemtests sind keine Treiber und Stellvertreter notwendig.) Treiber ersetzen die Module, von denen das zu testende Modul normalerweise benutzt wird. Stellvertreter ersetzen die Module, auf die ein zu testendes Modul zugreifen muß, um seine Leistung zu realisieren (genaueres siehe Kapitel 13.3).

[11] bzw. das Istdatum in der Menge der Solldaten enthalten ist

[12] bzw. intern vom Typ *Integer*, wobei Pfennige gezählt werden und dies extern in Markbeträge umgerechnet/ausgegeben wird.

Treiber und Stellvertreter können als interaktive Schnittstelle zwischen der Testperson und dem zu testenden Objekt realisiert werden. Die Testperson kann über die Treiber das zu testende Modul mit Testdaten versorgen. Über die Stellvertreter gibt die Testperson die Daten ein, die ein Modul zu seiner Weiterarbeit benötigt.
Bei einer anderen Realisierungsart lesen die Treiber und Stellvertreter die benötigten Daten automatisch aus einer zuvor erstellten Datei ein.

3.5.4 Testausführung und Testauswertung

Nach der Testvorbereitung erfolgt die Ausführung der Softwarekomponente mit den Testdaten. Bei imperativen, sequentiellen, deterministischen Programmen, die bei den üblichen Testverfahren vorausgesetzt werden, bereitet dies keine Probleme. Dabei können Testabläufe beliebig reproduziert werden, d. h. die wiederholte Eingabe gleicher Testdaten bewirkt die Ausführung derselben Anweisungsfolge. Das entfällt bei nichtdeterministischen Programmen, insbesondere nebenläufigen, verteilten oder Echtzeit-Programmen. Sie erfordern eine geeignete Testumgebung mit besonderen Teststeuerungsmaßnahmen und benötigen eventuell sogar zusätzliche Testhardware (genaueres s. Kapitel 14.5).

Die nachfolgende **Testauswertung** umfaßt folgendes:

1. Messung der **Testgüte**, d. h. den Grad der Erfüllung des Testkriteriums durch die Testdatenmenge (auch **Testwirksamkeitsmaß** genannt),

2. **Soll-Ist-Vergleich**, d. h. für jedes Testdatum ist das tatsächliche mit dem spezifizierten Ergebnis zu vergleichen, um ein Fehlverhalten festzustellen. Ein sogenanntes **Testorakel** muß dabei die Sollwerte aus der Spezifikation ableiten. Dabei gibt es die folgenden Möglichkeiten:

   (a) Ein Mensch fungiert als Testorakel, indem er ein vermeintliches Solldatum (anhand der Spezifikation) aus dem Eingabedatum ableitet. Je nach der Güte der Spezifikation und der Fähigkeit und Konzentration des Menschen ist dies mehr oder minder fehlerhaft.

   (b) Beim **diversitären** Testen wird das Programm von unabhängigen Entwicklungsgruppen mehrfach entwickelt und jede Programmversion wird gegen die anderen getestet. Damit werden alle Fehler entdeckt, die nur in einer Teilmenge der Versionen existieren; nur Fehler, die in jeder Programmversion vorhanden sind, werden nicht entdeckt. Je mehr Versionen erstellt und durch Tests miteinander verglichen werden, desto geringer ist die Wahrscheinlichkeit, daß Fehler auftreten, die in allen Versionen vorhanden sind.

   (c) Wenn formale Spezifikationen (z. B. algebraische Spezifikationen) vorliegen, kann daraus (fast) automatisch ein Prototyp erzeugt werden, gegen den — ähnlich wie beim diversitären Testen — getestet werden kann.

In jedem Fall kann der eigentliche Vergleich (Solldaten gegen Istdaten) automatisch erfolgen, wenn die Sollwerte zu den Eingabedaten vorliegen.

Tritt ein Fehlverhalten auf, so muß der verursachende Fehler lokalisiert und behoben werden (genaueres s. Kapitel 15). Nach der Fehlerkorrektur wird der Test erneut ausgeführt und ausgewertet (**Regressionstest**). Tritt kein Fehler auf, so muß überprüft werden, ob die gewünschte Testgüte erreicht ist und der Test beendet werden kann (siehe Kapitel 3.8 und genaueres in Kapitel 16.4 und 16.5). Ist die Testgüte nicht erreicht worden, müssen weitere Testdaten erstellt werden, d. h. der Testzyklus wird erneut durchlaufen.

3.6 Testphasen im Software-Entwicklungsprozeß

3.6.1 Testphasen

In allen Phasen der Softwareentwicklung (Anforderungsdefinition und -spezifikation, Entwurf und Modulimplementierung) sind jeweils einige passende (der in Kapitel 3.3 und 3.4 vorgestellten) Prüf- und Testverfahren anzuwenden. In jeder Phase können Beschreibungsregeln auf der jeweiligen Entwicklungsstufe verletzt werden, bei jedem Phasenübergang können (Transformations-)Fehler gemacht werden; beides ist durch statische Überprüfung festzustellen.
Der Testprozeß für ein implementiertes Programm kann in drei Phasen unterteilt werden, die zeitlich nacheinander durchgeführt werden: den Modultest, den Integrationstest und den Systemtest. Dabei wird das jeweilige Testobjekt (Modul, Teilsystem oder System) gegen seine Spezifikation (inklusive Schnittstellenspezifikation) getestet. (Die entsprechenden Testdaten für den Modul-, Integrations- und Systemtest sollten dabei schon in der entsprechenden, vorher durchgeführten Spezifikations- und Entwicklungsphase festgelegt werden.) Anschließend erfolgt eine Montierung der Module zu Komponenten, Subsystemen und schließlich dem gesamten System.

3.6.1.1 Modultest

Bei den bisherigen Betrachtungen waren die zu testenden Objekte Programme oder Teile von Programmen. Ein Programmteil wird als **Modul** bezeichnet, falls dieser Programmteil eine logisch separierbare Einheit mit klar definierten Schnittstellen darstellt. Die Menge aller Module eines Programms bildet ein (Modul-)**System**. Beim Modultest wird ein Modul individuell und unabhängig von den anderen Modulen des Systems getestet. Zur Durchführung eines Modultests sind Treiber und Stellvertreter (Platzhalter, engl.: stubs) notwendig (vgl. Abschnitt 3.5.3).

3.6.1.2 Integrationstest

Das Ziel des Integrationstests ist es, die Schnittstellen der Module untereinander zu testen und das Zusammenspiel der Module zu überprüfen. Der Integrationstest orientiert sich an der Benutzungshierarchie der Module. (Bei objektorientierten Programmen sind als Modulbeziehungen neben den Aufrufbeziehungen auch die Vererbungsbeziehungen zu beachten.)

Das Vorgehen kann nichtinkrementell oder inkrementell sein. Für das inkrementelle Testen sind zwei grundsätzliche Teststrategien zu unterscheiden: die absteigende Strategie und die aufsteigende Strategie (genaueres siehe Kapitel 13.3).

3.6.1.3 Systemtest, Abnahmetest und Regressionstest

Wenn alle Module integriert sind, wird ein abschließender, spezifikationsorientierter Systemtest durchgeführt. Beim **Systemtest** wird das gesamte System gegen die funktionalen und nichtfunktionalen[13] Systemanforderungen, welche in der Anforderungsspezifikation festgelegt sind, und gegen die Handbücher getestet. Wird das System von den Auftraggebern bzw. Benutzern getestet, ob es die ursprünglichen oder aktuellen Einsatzzwecke erfüllt, spricht man von einem **Abnahmetest**.
In der sogenannten Wartungsphase werden verbliebene Fehler korrigiert und Funktionen bzw. Datenstukturen des Systems geändert oder ergänzt. Daher sind **Regressionstests** erforderlich, um unbeabsichtigte Verfälschungen des bisherigen Systemverhaltens aufzudecken.

3.6.2 Testen im Software-Entwicklungsprozeß

Ein **Testprozeß** ist die Durchführung aller Testphasen und Testaktivitäten zur Realisierung eines Gesamttests. Ein **Testprozeßmodell** beschreibt die grundsätzlichen Merkmale eines Testprozesses beziehungsweise einer Klasse von Testprozessen. Es werden zwei grundsätzliche Testprozeßmodelle, das Phasenmodell und das Lebenszyklusmodell, unterschieden.

Das **Phasenmodell** betrachtet Testen als eine eigenständige Phase im Softwarelebenszyklus, die *nach* der Entwicklung des Systems einsetzt. Nicht nur die Testausführung wird dieser Phase zugeordnet, auch die Testvorbereitung (Testplanung, Testdatenerstellung und Testrahmenerstellung) wird erst nach der Systementwicklung ausgeführt. Das Phasenmodell wird in der Praxis noch sehr häufig eingesetzt.

Das **Lebenszyklusmodell** betrachtet Testen nicht als eine getrennte Phase am Ende der Entwicklung der Software. Vielmehr werden die Testaktivitäten *parallel* zur

[13] Effizienz (z. B. Antwortzeitverhalten und Durchsatz unter normalen Bedingungen und bei Überlast), Benutzbarkeit und andere Qualitätsmerkmale (s. Def. 3.2.1.3 auf Seite 50 f.)

Entwicklung durchgeführt, in enger Verflechtung mit den frühen Phasen. Das Lebenszyklusmodell kann — im Unterschied zum Phasenmodell — die Informationen, die während der Erstellung eines Softwaresystems anfallen, gewinnbringend für den Testprozeß einsetzen und die während der Entwicklung erstellten Testdaten für die manuelle Überprüfung der Spezifikation verwenden. Das Lebenszyklusmodell ist daher dem Phasenmodell vorzuziehen. Nachteilig ist allerdings, daß die Anwendung des Lebenszyklusmodells aufwendiger ist als die Anwendung des Phasenmodells.

3.7 Überprüfung des Entwicklungs- und Qualitätsmanagementsystems

Mit den bisher vorgestellten Überprüfungs- und Testansätzen können Fehler zwar beseitigt, aber nicht verhindert werden. Außerdem kann es vorkommen, daß der Ablauf oder die Methoden des Überprüfens und Testens Schwächen aufweisen und somit viele Fehler unentdeckt bleiben. Daher sollte das Entwicklungs- und Qualitätsmanagementsystem selbst überprüft werden.
Dazu wurde vom Software Engineering Institut SEI im Auftrag des US-Verteidigungsministeriums ein **Reifegradmodell** (capability maturity model, CMM) entwickelt, welches den Entwicklungsprozeß in einer Institution bewertet und in fünf Stufen (1 = initial, 2 = reproduzierbar, 3 = definiert, 4 = beherrscht, 5 = optimierend) einteilt. Auf der „initialen“ Stufe herrscht z. B. „weitgehendes Chaos. Es gibt keine definierte Vorgehensweise bei der Software-Erstellung und -Wartung ... Werkzeuge sind nicht vorhanden oder kommen nur sporadisch zum Einsatz. Änderungen an der Software werden nicht dokumentiert und unterliegen keinerlei Kontrolle ... “ (s. [Hoh 95], S. 326). Auf der Stufe „reproduzierbar“ ist dagegen ein einmal erzielter Erfolg wiederholbar, wenn es sich um vergleichbare Projekte handelt. Auf der höchsten Stufe („optimierend“) ist die gesamte Organisation dagegen „auf kontinuierliche Prozeßverbesserungen ausgerichtet. Daten über die Wirksamkeit von Prozessen sind zur Durchführung von Kosten-Nutzen-Analysen und zur Empfehlung von Prozeßänderungen verfügbar. Innovationen, welche die besten Software-Engineering-Praktiken ausnutzen, werden identifiziert und unternehmensweit eingeführt ... “ (s. [Hoh 95], S. 327).
Die Anforderungen der dritten Stufe („definiert“) sind erfüllt, wenn der Software-Prozeß „standardisiert und konsistent [ist], weil sowohl die Aktivitäten des Software-Engineering als auch die des Management stabil und reproduzierbar sind ... “ (s. [Hoh 95], S. 327). Diese Anforderungen entsprechen in etwa den Anforderungen der Norm ISO 9000-3, die allerdings auch das Umfeld des Software-Entwicklungsprozesses mit einbezieht. Betrachtet werden dabei folgende Qualitätsmanagementelemente (QM-Elemente):

- Führungselemente (Verantwortung der obersten Leitung, Schulung)
- phasenübergreifende QM-Elemente
  - systembezogene (z. B. Korrekturmaßnahmen)
  - allgemeine (z. B. Qualitätsaufzeichnungen)
  - produktbezogene (z. B. Prüfmittel, Lenkung fehlerhafter Produkte)
- phasenbezogene QM-Elemente (z. B. Vertragsprüfung, Beschaffung)

Aufbauend auf der CMM-Methode wurde in Europa als Ergebnis des ESPRIT-Projekts BOOTSTRAP ein ähnliches Bewertungsverfahren entwickelt. Dabei sollen sowohl die Fähigkeitsstufen von Organisationen als auch von Entwicklungs- und Wartungsmethoden untersucht und bewertet werden[14].

Fazit: Mit Reifegradmodellen und entsprechenden Bewertungen läßt sich — wie Untersuchungen zeigen — die Produktivität und Qualität der Software-Entwicklung steigern, und zwar durch systematische Prozeßverbesserung. Allerdings sind die Bewertungsergebnisse in der Regel nicht genau genug, um konkrete Verbesserungen ableiten zu können. Bei den Stufen 1 bis 3 wird nur der Entwicklungsprozeß — und nicht die Produktqualität — betrachtet und es werden keine speziellen Werkzeuge oder Methoden für die Produktherstellung verlangt. Daher könnte — überspitzt formuliert — ein Hersteller zertifiziert werden, der Schwimmwesten aus Beton fabriziert, wenn diese Westen in Übereinstimmung mit den dokumentierten Abläufen hergestellt werden[15]. Erst bei Stufe 4 und 5 werden quantitative bzw. qualitative Produkteigenschaften betrachtet. Es gibt allerdings wenig Erfahrungswerte mit diesen hohen Reifegraden und die Anforderungen dafür sind entsprechend vage formuliert. Daher besteht die Gefahr, daß sich Organisationen auf einer positiven Bewertung für Stufe 2 oder 3 „ausruhen", der tatsächliche Reifegrad der Entwicklung und des Qualitätsmanagements nicht ansteigt[16] und die Entwicklungsprodukte nicht die gewünschte Qualität besitzen. Daher müssen die Reifegrad-Modelle konkretisiert werden, d. h. neben dem prozeßbezogenen muß für das produktbezogene Qualitätsmanagement eine Menge von geeigneten Werkzeugen und Methoden zur Erreichung und Überprüfung der Produktqualität vorgeschrieben werden.

[14] Im Rahmen des SPICE-Projekts (Software Process Improvement and Capability Determination) sollen die verschiedenen Verfahren integriert und eine entsprechende ISO-Norm entwickelt werden.

[15] Tatsächlich zeigen (von 1992 bis 1995 vorgenommene) umfangreiche Messungen von kommerzieller Software folgendes: Die Existenz eines „konsistenten" Software-Entwicklungsprozesses (nach ISO 9001 zertifiziert) führt nicht zu qualitativ höherwertigen Softwareprodukten (mit weniger statisch erkennbaren Fehlern und weniger Verletzungen von Standards).

[16] Eine Organisation braucht (nach Underwood) 9 bis 14 Jahre, um von Stufe 1 nach Stufe 5 zu gelangen (s. [Und 95], S. 428).

3.8 Testmanagement

Zum Testmanagement gehört u. a. die Auswahl und Kombination der Test- und Prüfmethoden und die Entscheidung über die Beendigung des Testens.

Da es eine Vielzahl von Prüfmethoden und Testkriterien gibt, ist eine Auswahl zu treffen. Dies kann durch Komplexitätsmaße gesteuert werden, die anzeigen, welche Programmkonstrukte besonders häufig und/oder mit hoher Komplexität vorkommen, so daß bestimmte Methoden besonders geeignet oder ungeeignet sind (genaueres s. Kapitel 16.2).

Als Kombination von Test- und Prüfmethoden bietet sich i. allg. die (sequentielle) Ausführung in folgender Reihenfolge an:

1. statische Prüfung und (eventuell) symbolische Ausführung,
2. spezifikationsorientierter (insbesondere funktionaler) Test,
3. implementationsorientierter (struktureller) Test,
4. formale Verifikation (bei kritischer Software).

Es ist eine wichtige, aber schwer zu entscheidende Frage, wann der richtige Zeitpunkt für das Ende des Testens gekommen ist. Das in der Praxis meistbenutzte Kriterium ist ein Abbruch wegen Terminüberschreitung und Drängen des Auftraggebers, die Software endlich auszuliefern.
Wenn diese Zwänge nicht vorliegen, sollte stattdessen die erzielte Testgüte als Abbruchkriterium herangezogen werden. Sie kann aus benutzungsorientierter, fehlerorientierter und struktureller Sicht beurteilt werden.

1. Ein benutzungsorientiertes Verfahren zur Abschätzung der Testgüte ist die Verfolgung der **Fehleraufdeckungsrate** (gefundene Fehler pro Tag oder Woche) in der Testphase. Geht diese Rate gegen Null, wird mit dem Testen aufgehört. Dieses Kriterium ist sehr subjektiv, da nichts über die Testanstrengungen ausgesagt wird. Es sollte daher zusätzlich ein objektiveres Kriterium eingesetzt werden.
2. Eine fehlerorientierte Beurteilung der bisher benutzten Testdatenmenge erfolgt durch die Mutationsanalyse und die Fehlereinpflanzung.
   Bei der **Mutationsanalyse** wird zu einem Programm eine Menge von Mutanten erzeugt, die sich vom Originalprogramm nur in jeweils einer Kleinigkeit unterscheiden (z. B. $X := A + 2$ statt $X := A + 3$). Bei der **Fehlereinpflanzung** werden mehrere Fehler gleichzeitig in das Programm eingebaut („eingepflanzt“) (genaueres siehe Kapitel 9.3 und Abschnitt 16.4.1).
3. Bei der strukturellen Sichtweise ist ein ausreichender (möglichst hundertprozentiger) **Überdeckungsgrad** bei den gewählten strukturellen Testkriterien eine Voraussetzung für die Beendigung des Testens.

3.9 Verwendete Quellen und weiterführende Literatur

Die **demonstrative Vorgehensweise** des Testens hat ihren zeitlichen Ursprung im Jahre 1957, als Charles Baker zum erstenmal „Debuggen" und Testen unterschied (s. [Bak 57]). Der Beitrag von Hetzel (s. [Het 72b]) liegt noch auf der gleichen Linie. Von Myers stammt die Definition der **destruktiven Vorgehensweise** des Testens (s. [Mye 79]. Die begriffliche Unterscheidung der demonstrativen und destruktiven Vorgehensweise stammt von Gelperin und Hetzel (s. [GeH 88]).
Ansätze zur **Klassifizierung von Qualitätsmerkmalen** wurden schon von Boehm et al. vorgestellt (s. [BBL 76]). Die hier angegebenen Definitionen von Produktqualität, Zuverlässigkeit und Funktionalität sind (meist wörtlich) den entsprechenden Angaben zu Qualitätsmerkmalen der Software in der DIN 66272 von Oktober 1994 entnommen worden. Die Definition der konstruktiven und analytischen Qualitätsmanagementmaßnahmen orientiert sich an Balzert (s. [Bal 82], S. 443 f.).
Die Klassifizierung der **Prüfverfahren** (Kap. 3.3) stammt von Hesse et al. (s. [He& 84]). Die Klassifizierung der **Testansätze** ist an die Klassifizierung von Sneed angelehnt (s. [Sne 88]). Die Unterscheidung des funktionalen und des strukturellen Testansatzes findet man schon bei Myers (s. [Mye 79]). Das funktionale Testen wurde zuerst von Howden (s. [How 86]) beschrieben. Der ablaufbezogene Testansatz wurde hier weiter verfeinert (in den kontrollflußbezogenen und den datenflußbezogenen Testansatz).
Die in Kapitel 3.4 bis 3.6 kurz vorgestellten **Testansätze**, **Testabläufe** und **Testphasen** werden in den folgenden Kapiteln genauer erläutert, allerdings nicht das **diversitäre Testen** (genaueres dazu s. [Kre 88], [NiG 90], [Vou 90], [Zei 86]). Die Unterscheidung der **Testprozeßmodelle** in Abschnitt 3.6.2 stammt von Gelperin und Hetzel (s. [GeH 88]).
Einen Überblick über das Reifegradmodell (**Capability Maturity Model)** und die Bewertung von Software-Entwicklungsprozessen gibt Liggesmeyer in [Lig 94]. Genauere Informationen dazu und zu den Weiterentwicklungen BOOTSTRAP und SPICE findet man bei Hohler, Kuvaja und Colette (s. [Hoh 95], [Kuv 95], [Col 95]). Die kritischen Bemerkungen zu den bisherigen Methoden und Modellen der Prozeßbewertung stammen von Hatton und Underwood (s. [Hat 95], [Und 95]).
Die in Kapitel 3.8 erwähnten Methoden zur Abschätzung der **Testgüte** und zur Beurteilung der Testdatenmenge wurden zuerst von Mills, Myers und DeMillo et al. vorgestellt (s. [Mil 72], zitiert nach [Mye 76], Kap. 18; [Mye 79], Kap. 6; [DLS 78]).

Zusammenfassung von Teil I

Anhand eines einfachen Programms zur Klassifikation von Dreiecken, das nur sieben Fälle behandelt, hat Kapitel 1 einen Eindruck von den Problemen des Testens vermittelt: bis zu 38 Testdaten sind für einen gründlichen Test sinnvoll. Damit ist aber trotzdem nicht klar, ob alle möglichen Fehler gefunden werden.
Kapitel 2 stellt Fallstudien über Art und Auftreten von Fehlern vor. Durch Irrtümer, Gedächtnisprobleme und mangelndes Problemverständnis erzeugen die Systementwickler etwa 30 bis 100 Fehler pro 1000 Anweisungen im Quellcode (s. Kapitel 2.1 und 2.2). Dies bewirkt ein Fehlverhalten, das oft nur ärgerlich ist aber manchmal auch kostenintensive oder sogar tödliche Auswirkungen hat (s. Kapitel 2.3). Mit konstruktiven Qualitätsmanagementmaßnahmen können die Fehlerquellen zwar eingedämmt werden, aber da es keine narrensicheren Methoden und Werkzeuge gibt, werden immer noch Fehler produziert (s. Kapitel 2.4). Es müssen also analytische Maßnahmen ergriffen werden, um die Fehler wenigstens anschließend zu beseitigen. Dazu müssen Test- und Überprüfungsmethoden möglichst frühzeitig eingesetzt werden, da die Kosten der Fehlerbeseitigung von ca. 350$ (bei früher Fehlererkennung) auf ca. 13.000$ pro Fehler (bei später Fehlerentdeckung) ansteigen (s. Kapitel 2.5). Da die (an sich ideale) formale Verifikation von fehlerhaften Programmen nicht möglich ist und ein vollständiger, erschöpfender Test aus Zeitgründen nicht praktikabel ist, müssen effektive und kostengünstige Test- und Überprüfungsverfahren eingesetzt werden, um möglichst alle (oder jedenfalls viele) Fehler aufzudecken. Solche Verfahren werden in Kapitel 3 kurz vorgestellt, die folgenden Abschnitte werden sie genauer behandeln: Die Teile II und III betrachten dynamische Verfahren, die Tests aus der Spezifikation des Programms ableiten bzw. aufgrund der Kenntnis der Struktur des implementierten Programms erzeugen. Teil IV enthält die statischen Verfahren, Verfahren zur Überprüfung von Systemen (insbesondere nebenläufigen Systemen) und Verfahren zum Lokalisieren und Beseitigen von Fehlern.
Der Einsatz der Test- und Prüfverfahren muß für den gesamten Softwarelebenszyklus und innerhalb jeder Phase sorgfältig geplant, ausgeführt und ausgewertet werden und das entsprechende Qualitätsmanagementsystem ist zu überprüfen (s. Kapitel 3.5 bis 3.7). Insbesondere sind für einen bestimmten Einsatzbereich die geeigneten Verfahren unter Kosten/Nutzen-Aspekten auszuwählen und geschickt zu kombinieren. Dazu gehören auch Verfahren, welche die Entscheidung liefern, ob der Test- und Überprüfungsprozeß beendet werden kann (s. Kapitel 3.8). Da der bisherige Methodeneinsatz nur ca. 95% der Fehler entdeckt und beseitigt (und sicher auch ein verbesserter Methodeneinsatz nicht alle Fehler behebt), müssen die Beteiligten mit Fehlern leben und sich die Entwickelnden von Software fragen, wie sie diese Situation verantworten können (s. Kapitel 2.6).