18.7 Werkzeuge zur Codierung

Das Thema *Werkzeuge und Entwicklungsumgebungen* haben wir allgemein bereits in Kapitel 15 behandelt. Nachfolgend gehen wir kurz auf die wichtigsten Werkzeuge für die Codierung ein.

Die minimale Werkzeugausstattung eines Programmierers besteht aus einem einfachen Editor und einem Übersetzer, Binder und Laufzeitsystem für die verwendete Programmiersprache. Noch immer werden mit dieser kargen Ausrüstung Projekte durchgeführt. Dabei gibt es eine Reihe von *Programmierwerkzeugen* – zu denen die oben genannten nicht gerechnet werden –, die die Arbeit des Programmierers erleichtern. Dazu gehören

- ein sprachsensitiver Editor, der die syntaktischen Strukturen erzeugt und anzeigt und einfachste Fehler erkennt,
- ein Werkzeug, das den Übersetzungs- und Bindeprozess steuert (z. B. make oder ant),
- ein Werkzeug für die Verwaltung der Quellprogramme, auch aller anderen Software-Einheiten und ihrer Konfigurationen (z. B. CVS oder Subversion),
- Werkzeuge, die den Test unterstützen und die Testüberdeckung messen (z. B. JUnit, Logiscope),
- ein Werkzeug, das den Code gegen die Codierregeln prüft (z. B. für JAVA CheckStyle),
- ein Debugger, der bei der Fehlersuche auf unterster Ebene hilft – wenn diese Ebene denn wirklich bearbeitet werden muss, wie es etwa bei vielen »embedded systems« der Fall ist.

Sind Programmierwerkzeuge unter einer gemeinsamen Bedienoberfläche integriert, dann sprechen wir von einer integrierten Entwicklungsumgebung. Welche Entwicklungsumgebung verwendet werden kann, hängt entscheidend von der Projektsituation ab.

Müssen grafische Bedienoberflächen entwickelt werden, dann kann ein sogenannter »GUI-Builder« eingesetzt werden (GUI = Graphical User Interface). Er gestattet es, die Bedienoberfläche interaktiv und komfortabel zu entwickeln; der Programmcode, der die Darstellungen erzeugt und die Interaktion implementiert, wird automatisch generiert. Wichtig ist dabei, dass die funktionalen Programmeinheiten, die nicht mit diesem Werkzeug erstellt werden, einfach und architektonisch sauber mit dem generierten Code verbunden werden können. Andernfalls besteht die Gefahr, dass der GUI-Code und der funktionale Code eng gekoppelt sind und damit ein schlecht wartbares Programm entsteht.

Zeller und Krinke (2003) geben eine sehr gute Einführung in eine Vielzahl von Programmierwerkzeugen, die alle erprobt, bewährt und frei verfügbar sind.

19 Programmtest

In Kapitel 13 haben wir die Aufgaben und Schwerpunkte der Software-Qualitätssicherung beschrieben und im Detail vorgestellt, wie Reviews organisiert und durchgeführt werden. In diesem Kapitel widmen wir uns dem Programmtest (oder einfach *Test*), der gemäß der in Abschnitt 13.1.2 eingeführten Terminologie eine dynamische, mechanisch durchgeführte Maßnahme der analytischen Qualitätssicherung ist.

Seit Beginn der Programmierung werden Programme getestet. Das Gebiet entstand also in der Praxis; seit den Siebzigerjahren wurde es aber auch systematisch erforscht.

Wir konzentrieren uns hier auf allgemein wichtige Aspekte des Tests. Eine ausführliche Betrachtung des Programmtests ist beispielsweise in Spillner und Linz (2005) oder in Liggesmeyer (2002) zu finden. Das vergriffene Buch von Riedemann (1997) ist elektronisch verfügbar.

19.1 Begriffe und Grundlagen des Tests

Die folgenden Zitate zeigen, dass mit dem Test ganz unterschiedliche Absichten verbunden sind:

> *»Testen ist die Ausführung eines Programms mit dem Ziel, Fehler zu entdecken.«*
>
> Myers (1979)

> *»Testen ist die Vorführung eines Programms oder Systems mit dem Ziel zu zeigen, dass es tut, was es tun sollte.«*
>
> Hetzel (1984)

19.1.1 Definition

Das Wort »Test« wurde aus dem Englischen importiert. Ursprünglich ist seine Bedeutung sehr allgemein, synonym mit »Prüfung«, und so wird es im Englischen auch noch oft verwendet. Wir legen diesem Kapitel eine sehr eingeschränkte Definition zu Grunde, etwa in der Bedeutung »praktisch ausprobieren«:

Testen ist die – auch mehrfache – Ausführung eines Programms auf einem Rechner mit dem Ziel, Fehler zu finden.

Wir grenzen den Begriff damit in mehrere Richtungen ab; *kein* Test in diesem Sinne ist

- irgendeine Inspektion eines Programms,
- die Vorführung eines Programms,
- die Analyse eines Programms durch Software-Werkzeuge, z. B. die Erhebung von Metriken,
- die Untersuchung eines Programms mit Hilfe eines Debuggers.

Das zu testende Programm – der Prüfling – entsteht durch Übersetzung aus dem Quellprogramm und enthält direkt oder indirekt Programme wie Editor, Compiler, Linker, Laufzeitsystem und Betriebssystem, aber auch Testtreiber und kooperierende Programme (z. B. ein Datenbanksystem). Wir können nur das Verhalten des gesamten Systems beobachten, nicht den Einfluss der einzelnen Komponenten.

Der Prüfling wird in der Regel auf einem Rechner ausgeführt, der für die Ausführung des Prüflings die gleiche virtuelle Maschine zur Verfügung stellt wie der Rechner, auf dem er später auch laufen soll. Es kann aber auch sinnvoll oder unvermeidlich sein, dass die Zielumgebung nur nachgebildet wird. Das ist vor allem dann notwendig, wenn das Programm später auf einem primitiven Prozessor ohne die zum Testen notwendige Peripherie läuft, wie es in technischen Anwendungen meist der Fall ist.

Wenn jemand ein Programm startet und spontan ein paar Werte eintippt, um zu sehen, ob das Programm damit umgehen kann, testet er. Aber diese Art des Tests ist ineffizient, meist auch ineffektiv. Wir schränken den Begriff darum weiter ein:

Ein **systematischer Test** ist ein Test, bei dem

- die Randbedingungen definiert oder präzise erfasst sind,
- die Eingaben systematisch ausgewählt wurden,
- die Ergebnisse dokumentiert und nach Kriterien beurteilt werden, die vor dem Test festgelegt wurden.

Betrachten wir die genannten Punkte genauer:

Die Randbedingungen eines Tests sind sämtliche Gegebenheiten, die auf die Resultate Einfluss haben oder haben können. Das ist zunächst der Prüfling selbst: Welches Programm, übersetzt von welchem Übersetzer, wird auf welchem Betriebssystem getestet? Welche andere Software ist beteiligt? Wer hat wann getestet? Wie viel Speicher steht zur Verfügung? Welche Geräte sind angeschlossen, und in welchem Zustand sind sie?

Aus der Spezifikation geht hervor, welche Eingaben das Programm akzeptieren muss und welche Reaktionen des Programms gefordert sind. Neben den Eingabedaten im engeren Sinn (interaktive Eingaben über Tastatur oder Maus, Dateien und Datenbanken, auf die das Programm zugreift) gehören im Allgemeinen auch die Anfangsbelegung des Speichers und die variablen Zustände der beteiligten Geräte zu den Eingaben. Natürlich können manche Einflüsse im konkreten Fall ausgeschlossen werden; wenn beispielsweise das Laufzeitsystem den Speicher automatisch initialisiert, hat die Vorbelegung keine Bedeutung.

Die Ausführung eines Programms wird im Allgemeinen durch den zu Beginn gegebenen Speicherzustand, die Eingabedaten und alle Randbedingungen beeinflusst, die aus der Umgebung einwirken. Natürlich wird in vielen Fällen angestrebt, dass sich nur die Eingabedaten auswirken, aber gerade beim Test dürfen wir nicht unterstellen, dass das auch wirklich so ist. Schon eine nicht initialisierte Variable im Programm bedeutet eine Abhängigkeit von der zufälligen Vorbelegung des Speichers. Eventuell haben auch die Signale aus einem Netzwerk, an das der Rechner angeschlossen ist, Folgen für die Programmausführung. All das ist im weitesten Sinne Eingabe.

Die Resultate des Tests werden dokumentiert, also aufgezeichnet und abgelegt. Bei interaktiven Tests muss diese Aufzeichnung von Hand erstellt werden, wenn keine speziellen Werkzeuge dafür zur Verfügung stehen.

Vor dem Test wurde bereits ermittelt, welche Resultate laut der Spezifikation zu erwarten sind. Im einfachsten Fall erhält man auf diese Weise Soll-Resultate, in der Praxis können meist nur Kriterien definiert werden, die notwendig, aber nicht hinreichend sind, um das Resultat als richtig zu beurteilen. Dieses Problem wird in Abschnitt 19.1.2 näher betrachtet. Die Ist-Resultate werden mit den Soll-Resultaten verglichen oder gegen die Kriterien geprüft.

Wenn beim Soll-Ist-Vergleich eine Abweichung festgestellt wird, liegt *ein* Fehler vor. Das bedeutet nicht sicher, dass der Prüfling fehlerhaft ist. Es kann auch sein, dass das Soll-Resultat oder der Vergleich fehlerhaft war; möglicherweise wurde auch das falsche Programm getestet. Das Resultat des Tests kann also falsch positiv sein (siehe Abschnitt 13.4.2). Das muss bei der Analyse des Fehlers geklärt werden.

Der Zweck des Tests ist, Fehler zu entdecken. Ein Testfall ist also gut, wenn er hohe Chancen hat, einen vorher noch nicht bekannten Fehler anzuzeigen. Tritt dieser Fall ein, so war der Test *erfolgreich*. Ist kein Fehler entdeckt worden, dann war der Test *erfolglos*. Der Test sollte die Einsatzsituation so simulieren, dass aus einem *erfolglosen Test* auf einen *erfolgreichen Einsatz* geschlossen werden kann.

Abbildung 13–3 auf Seite 276 zeigt den prinzipiellen Informationsfluss bei einer Prüfung. Abbildung 19–1 zeigt den Informationsfluss, der bei der Vorbereitung, Durchführung und Auswertung eines Tests entsteht: Auf Basis der Anforderungen, bei einem Glass-Box-Test (siehe Abschnitt 19.6) auch auf Basis der bereits gemessenen Überdeckungen und des Codes, werden die Testfälle entworfen. Jeder Testfall besteht aus den Voraussetzungen, die erfüllt sein müssen, der Testeingabe und dem Soll-Resultat. Wenige Soll-Resultate stehen explizit in der Spezifikation (beispielsweise, wenn eine bestimmte Anfangsmeldung spezifiziert ist), die übrigen müssen aus den Anforderungen durch eine simulierte Ausführung abgeleitet werden. Ein Testfall ist also mehr als nur eine Testeingabe!

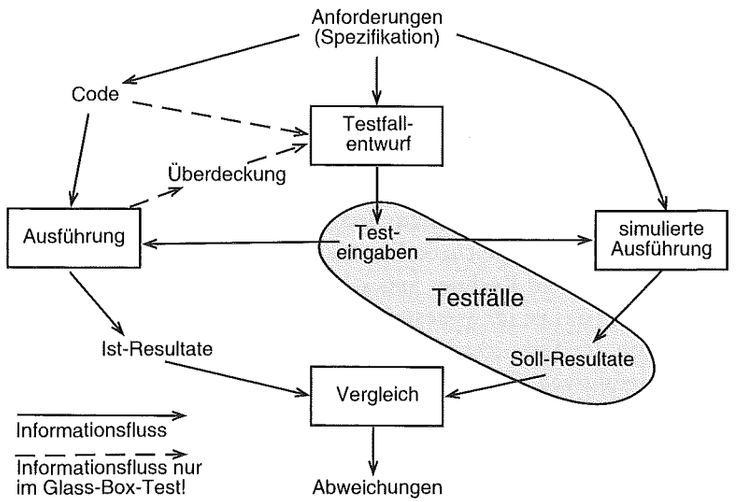

Abb. 19–1 *Informationsfluss beim Test (schematisch)*

19.1.2 Soll-Resultate

Zur »Bilderbuch-Vorstellung« von einem mustergültigen Test gehört das Soll-Resultat. Wenn beispielsweise eine Prozedur zu einem Datum zwischen 1900 und 2100 die Anzahl der Tage im betreffenden Monat liefern soll, dann ist genau definiert, was richtig ist. Im Test kann für die Eingabe 10.9.2003 das Resultat 30, für den 13.2.1900 das Resultat 28 verlangt werden. In vielen Fällen ist das aber nicht so einfach möglich, weil

- es mehrere oder viele richtige Ergebnisse gibt (Beispiele: der von einem Compiler erzeugte Objektcode, die von einem Reservierungssystem gewählte Zuordnung zwischen Buchungen und Plätzen im Flugzeug),

- das Ergebnis nicht präzise spezifiziert ist, weil Anforderungen weich (Abschnitt 16.4.2) oder nur vage definiert sind (Beispiele: Positionierung eines Ausgabefensters auf dem Bildschirm, der exakte Text einer Fehlermeldung),
- es kein mathematisch exakt richtiges Ergebnis gibt, das auf dem Rechner darstellbar wäre (Beispiele: alle numerischen Berechnungen, in denen Werte wie 1/3, $\sqrt{2}$ oder π vorkommen),
- das richtige Ergebnis einfach nicht bekannt ist (Beispiele: Berechnung der Zahl π auf 10000 Stellen genau, wenn π noch nie zuvor so genau bestimmt wurde, Berechnung astronomischer Größen auf der Grundlage neuer Beobachtungen oder neuer Modelle).

In allen diesen Fällen können nur Eigenschaften und Merkmale eines richtigen Resultats angegeben werden, nicht das richtige Resultat. Für die genannten Beispiele kommen folgende Merkmale in Frage:

- Wenn der Prüfling (der Compiler) ein Programm übersetzt und der generierte Code ausgeführt wird, muss er sich so verhalten, wie es die Definition der übersetzten Programmiersprache verlangt; das Buchungssystem muss so viele Buchungen akzeptieren und konfliktfrei durchführen, wie Plätze verfügbar sind.
- Das Ausgabefenster muss in einem bestimmten Bereich des Bildschirms erscheinen; die Fehlermeldung muss verständlich und informativ sein.
- Das Resultat der Rechnung (mit gewissen Eingabedaten) muss in einem bestimmten Intervall liegen, z. B. zwischen 917,1 und 917,3.
- Die ersten n Stellen der neu berechneten Zahl π müssen mit den bislang als gesichert geltenden ersten n Stellen von π übereinstimmen; die Gesamtmasse des Universums muss größer sein als die Masse aller bekannten Galaxien.

Die Forderung nach einem einzigen konkreten Soll-Resultat ist also für die Praxis meist zu streng. Wir geben stattdessen *Prädikate* an, die das Resultat eingrenzen. Diese Prädikate haben für die Teilresultate $r_1, r_2, \ldots, r_n$ die Form einer Funktion $f(r_1, r_2, \ldots, r_n)$, die als Resultat wahr (in der Bedeutung »plausibel«) oder falsch (in der Bedeutung »fehlerhaft«) liefert. Wenn nachfolgend von Soll-Resultaten die Rede ist, so sind damit meist die beschriebenen Prädikate gemeint.

Natürlich bietet es sich an, die Prüfung der Resultate gegen die Soll-Resultate zu automatisieren. Beispielsweise könnte das Prädikat »das Resultat x sollte zwischen 917,1 und 917,3 liegen« im JUnit-Testrahmenwerk folgendermaßen realisiert werden:

```
assertTrue(917.1 <= x & x <= 917.3);
```

19.1.3 Der Regressionstest

Eine spezielle, in der Software-Wartung besonders wichtige Technik, die Soll-Resultate zu erzeugen, wird beim Regressionstest angewendet.

> **regression testing** — Selective retesting of a system or component to verify that modifications have not caused unintended effects and that the system or component still complies with its specified requirements.
>
> IEEE Std 610.12 (1990)

Wenn ein Programm im Zuge der Wartung verändert wurde, hoffen wir, dass der angestrebte Effekt (eine Korrektur, Anpassung oder Erweiterung) erzielt wurde, dass aber keine unbeabsichtigten Effekte, also Fehler, entstanden sind. Leider sind solche Fehler aber sehr wahrscheinlich (siehe Abschnitt 22.3.1). Darum ist es zweckmäßig, auch einen Test durchzuführen, der speziell darauf abzielt, Fehler durch Veränderungen zu entdecken. Das ist der Regressionstest.

Dem Regressionstest liegen folgende Feststellungen und Annahmen zu Grunde:

- Die Änderung des Programms ist geringfügig, das Verhalten des Programms ist nach der Änderung überwiegend wie vorher.
- Das Programm wurde vor der Änderung eingesetzt und hat dabei keine oder nur wenige falsche Resultate geliefert; es war also grundsätzlich in Ordnung, selbst wenn die Änderung eine Korrektur war.
- Nach einer Korrektur liefert das Programm unter gleichen Bedingungen gleiche Resultate (mit Ausnahme der bislang falschen Resultate).
- Nach einer Erweiterung liefert es gleiche Resultate, denn die neue Funktionalität kann bislang noch nicht verwendet worden sein. Die alte Funktionalität darf sich nicht geändert haben.
- Nach einer Modifikation dürfen die neuen Resultate dort und nur dort von den alten abweichen, wo dies gewünscht war.

Betrachten wir dazu ein Beispiel: Ein Programm, das die Einwohner einer Gemeinde verwaltet, hat der ältesten Bürgerin der Stadt zum sechsten Geburtstag gratuliert. Anscheinend wurde das Alter (06) nur zweistellig gespeichert. Ansonsten gab es mit diesem Programm keine Probleme. Es soll nun korrigiert und auch erweitert werden, sodass die Verwaltung feststellen kann, ob ein Einwohner noch einen Zweitwohnsitz hat. Nach der Änderung muss das Programm mit den gleichen Eingaben die gleichen Ergebnisse liefern wie vorher, nur die alte Dame sollte keinen Brief mit dem Inhalt »Bald kommst du zur Schule ...« bekommen. Die oben genannten Annahmen treffen in diesem Fall also zu.

Um einen Regressionstest durchführen zu können, muss präzise dokumentiert worden sein, wie sich das Programm früher verhalten hat. Dazu sind alle Eingaben, im Beispiel auch der aktuelle Stand der elektronischen Einwohnerkartei, zusammen mit den Ausgaben zu archivieren. Das geänderte Programm wird

mit den alten Daten gefüttert. Die (meist sehr umfangreichen) Resultate, darunter auch die veränderte Einwohnerkartei, können dann mit den archivierten Resultaten verglichen werden. Wenn alle Ergebnisse in elektronischer Form vorliegen, kann das vollautomatisch geschehen. Die im Vergleich erkannten Unterschiede werden gemeldet. Im Beispiel sollte der einzige Unterschied sein, dass der unsinnige Brief nicht erzeugt wird.

Außerdem muss natürlich auch in einem weiteren, separaten Schritt getestet werden, ob die beabsichtigten Änderungen erfolgreich waren. Dazu brauchen wir im Beispiel Testfälle für Leute mit und ohne Zweitwohnsitz.

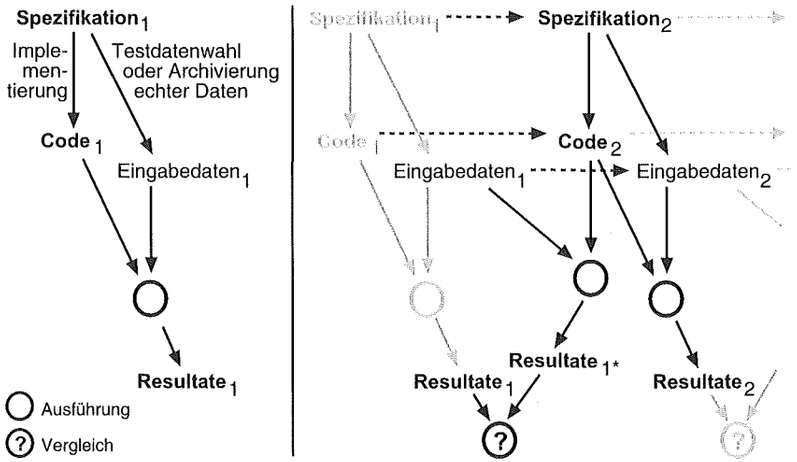

Abb. 19–2 *Vorgehen beim Regressionstest (ohne Korrekturen nach den Tests)*

Abbildung 19–2 zeigt schematisch den Ablauf. Nachdem das Programm implementiert ist und eingesetzt wurde, ist die Situation so wie auf der linken Seite dargestellt; wichtig ist, dass alle Dokumente, neben Spezifikation und Code auch die Eingabedaten und Resultate, archiviert sind. Rechts sieht man, wie eine Änderung durchgeführt wird: Die Änderung der Spezifikation führt zu einer Änderung des Codes (gestrichelte Pfeile). Nun wird der *neue* Code mit den *alten* Daten ausgeführt, die Resultate (1^*) werden mit den alten Resultaten (1) verglichen, die Abweichungen werden wie oben beschrieben interpretiert. Neue Eingabedaten (entsprechend der veränderten Spezifikation) ergänzen die alten, es entstehen die Eingabedaten_2. Auch mit diesen wird der Code_2 ausgeführt, die Resultate_2 werden wieder archiviert. Damit ist der nächste Änderungszyklus (rechts grau angedeutet) vorbereitet, er wird genau wie der erste durchgeführt.

Der Vorteil des Regressionstests liegt darin, dass man nicht (nur) die ursprünglichen Testdaten verwendet, sondern die meist reichlich verfügbaren Daten aus dem realen Einsatz der Software. Man verwendet also die alten Resultate als Soll-Ergebnisse für das geänderte Programm.

Allerdings lässt sich der Regressionstest nicht in allen Fällen anwenden: Wenn beispielsweise ein optimierender Compiler minimal verändert wurde, kann er durchaus völlig anderen Code erzeugen als zuvor. In diesem Falle gilt nicht mehr die Voraussetzung, dass sich die Ausgabe des Programms nicht oder nur sehr wenig ändert.

19.1.4 Fehler und Fehlersymptome

Wer einen Löwen jagt, muss wissen, wie ein Löwe aussieht. Er sollte auch wissen, wo sich der Löwe gern aufhält, welche Spuren er hinterlässt, welche Geräusche er erzeugt. Bei der Jagd nach Fehlern im Programm ist das nicht anders.

Leider wissen wir aber über die Fehler fast nichts (vgl. Abschnitt 13.3.2), nur dies: Es gibt mindestens einen Testfall, der den Fehler anzeigt, bei dem also Soll- und Ist-Resultate nicht übereinstimmen.

Wenn es genau einen solchen Testfall gibt, sprechen wir von einem *Punktfehler.* Wenn es dagegen Wertebereiche gibt (beispielsweise alle negativen Zahlen oder alle Zeichenreihen, die zwei gleiche Zeichen enthalten), liegt ein *Bereichsfehler* vor.

Punktfehler entstehen nur durch Sabotage oder (viel wahrscheinlicher) durch Dummheit und Schlamperei, selten auch durch Zufall. Wer beim Testen bestimmte Variablen sehen will und darum eine Abfrage einbaut, die auf die Eingabe »Donald Duck« reagiert, hat, wenn er die Abfrage im Programm vergisst, einen Punktfehler eingebaut. Einen solchen Punktfehler, dessen Ursache uns leider nicht bekannt ist, gab es in Microsoft Excel 4.0 (1992), das speziell auf die Zahl 1,407 374 883 553 28 bizarr reagierte. Anders lag der Fall in einer Triebwerkregelung der US Air Force, in der die Wertebereiche »> K« und »< K« unterschiedlich behandelt wurden, der Wert K aber vergessen worden war; auch das führt zu einem Punktfehler. Wenn der Fehler wie bei Excel für einen von 10^{15} Werten auftritt, ist seine zufällige Entdeckung praktisch unmöglich; sie ist weniger wahrscheinlich als zwei Sechser im Lotto in zwei aufeinanderfolgenden Ziehungen mit jeweils einem Tipp.

Bereichsfehler betreffen einen bestimmten Anteil q des Eingaberaums. Die Verwendung von n zufällig gewählten Eingaben deckt (unter der realistischen Annahme eines sehr großen Eingaberaums) den Fehler mit der Wahrscheinlichkeit $Q = 1-(1-q)^n$ auf. Tabelle 19–1 zeigt für einige Werte von n und q, wie wahrscheinlich es ist, dass ein Bereichsfehler entdeckt wird.

n \ q >	0,1	0,03	0,01	0,003	0,001
10	65 %	26 %	10 %	3 %	1 %
30	96 %	60 %	26 %	9 %	3 %
100	99,997 %	95 %	63 %	26 %	10 %
300	100,000 %	99,989 %	95 %	59 %	26 %
1000	100,000 %	100,000 %	99,996 %	95 %	63 %

Tab. 19–1 *Die Wahrscheinlichkeit, dass ein Bereichsfehler entdeckt wird*

Wie man an den Zahlen sieht, kommt es auf das Produkt von n und q an. Die folgende Rechnung zeigt, dass ein Fehler mit der Wahrscheinlichkeit $e^{-q \cdot n}$ unentdeckt bleibt: Die Wahrscheinlichkeit, dass ein Fehler in n Tests nicht erkannt wird, ist

$$F = (1-q)^n = (1-1/p)^n = (1-1/p)^{p \cdot a} = ((1-1/p)^p)^a = Z^a$$

Darin ist p als Kehrwert von q definiert und a als $n \cdot q = n/p$. Z steht für $(1-1/p)^p$. Für große p kann man statt 1-1/p auch den Kehrwert von 1+1/p setzen wegen $(1+x) \cdot (1-x) = 1-x^2 \approx 1$ für kleine x. Damit ist $Z \approx (1+1/p)^{-p} \approx 1/e$ entsprechend der Definition von e: $e = \lim (1+1/k)^k$ für $k \to \infty$. Also ist $F \approx e^{-a} \approx e^{-q \cdot n}$. Um einen bestimmten Wert von F zu erreichen, sind n Tests nötig mit $n = a/q = - \ln(F)/q$. Beispiel: Sei $q = 10^{-9}$, $F = 10^{-3}$. Daraus folgt $n = 3 \cdot \ln(10) \cdot 10^9 \approx 7 \cdot 10^9$. Wenn wir ein Programm, das in einem von 10^9 Fällen ein falsches Resultat liefert, mit 7 Milliarden Testfällen untersuchen, ist die Wahrscheinlichkeit, dass wir den Fehler übersehen, etwa eins zu tausend.

19.1.5 Vollständiger Test und Stichprobe

Wenn es genau x Möglichkeiten gibt, ein Programm auszuführen (z. B. x verschiedene Werte des einzigen Parameters einer Prozedur), und alle x Möglichkeiten im Test geprüft werden, haben wir einen *vollständigen Test* durchgeführt. Sind die Resultate in allen Fällen korrekt, beweist der Test die Korrektheit des Prüflings.

Leider ist ein vollständiger Test nur in extrem einfachen Fällen möglich, beispielsweise für eine Prozedur, die prüft, ob mindestens zwei von drei Parametern des Typs boolean TRUE sind; in diesem Fall gibt es nur acht Ausführungsmöglichkeiten. Sobald Zahlen vorkommen, ist die Zahl der möglichen Ausführungen viel zu hoch, um einen vollständigen Test zu gestatten. Schon eine Funktion mit einem einzigen INT-Parameter hat 2^{32}, also mehr als vier Milliarden Ausführungsmöglichkeiten, die beliebig unterschiedlich sein können. Ein Programm, das von drei Variablen, Parametern o. Ä. mit je 32 bit abhängt, hat 2^{96} oder etwa $80 \cdot 10^{27}$ verschiedene Startzustände, erfordert also im vollständigen Test $80 \cdot 10^{27}$ Testfälle. Wer die Möglichkeit hätte, pro Sekunde eine Milliarde Testfälle zu bearbeiten, brauchte dazu etwa das 190-fache Alter des Universums . Darum ist der

vollständige, das Programm verifizierende Test völlig ausgeschlossen; ein Programm ist niemals *ausgetestet*, es ist auch nach vielen tausend Tests kaum *angetestet*, denn mehr als 99,999 % der möglichen Eingabedaten wurden im Test *nicht* verwendet!

Wir müssen beim Testen also eine *Stichprobe* wählen, einige wenige Fälle von den praktisch unendlich vielen möglichen; wenn wir dabei geschickt vorgehen, können wir die Chancen, Fehler zu entdecken, verbessern. Wir können aber niemals sicher sein, alle Fehler entdeckt zu haben. Dijkstra hat das in den berühmten Satz gefasst:

> *Program testing can be used to show the presence of bugs,*
> *but never to show their absence!*
>
> E.W. Dijkstra (1970)

Eine bewährte Faustregel besagt, dass in einem (leidlich systematischen) Test die Hälfte aller Fehler auffällt. Eine schlampige Software-Entwicklung kann also nicht durch einen intensiven Test ausgeglichen werden. Wer nach Test und Korrektur weniger als n Fehler in 1000 LOC haben will, muss dafür sorgen, dass 1000 LOC *vor* dem Test nicht mehr als 2n Fehler enthalten, d. h., es gilt in erster Linie, durch geeignete konstruktive Maßnahmen Fehler zu *vermeiden*.

19.1.6 Vor- und Nachteile des Testens

Die wichtigsten Vorzüge des Testens gegenüber anderen Verfahren (vor allem Inspektionen) sind:

- Testen ist ein »natürliches« Prüfverfahren. Denn wir probieren alles aus, um zu sehen, ob es funktioniert, oft auch, um es (im ursprünglichen Sinne) zu *begreifen*.
- Der (systematische) Test ist reproduzierbar und damit objektiv, wenn die Anfangssituation reproduzierbar ist und die Testumgebung deterministisch arbeitet. Sein Erfolg hängt also nicht von der Tagesform des Testers ab.
- Den investierten Aufwand kann man mehrfach nutzen. Denn ein Test lässt sich, einmal vorbereitet und sauber dokumentiert, mit geringem Aufwand wiederholen. Das ist vor allem in der Software-Wartung ein entscheidendes Argument.
- Die Testumgebung wird mitgeprüft. Fehler in den beteiligten Komponenten (Übersetzer, Bibliotheken, Betriebssystem usw.) fallen möglicherweise im Test auf.
- Das Systemverhalten wird sichtbar gemacht. Auch wenn der Test in der Regel keine Prüfung der Effizienz oder der Bedienbarkeit einschließt, fallen Mängel in diesen und anderen Punkten doch wahrscheinlich auf.

Diesen Vorteilen steht eine Reihe von Nachteilen gegenüber:

- Ein Korrektheitsnachweis durch Testen ist praktisch unmöglich (siehe Abschnitt 19.1.5).
- Im Test kann man nicht alle Anwendungssituationen nachbilden, sei es, weil sie sich nicht einfach herbeiführen lassen (z. B. eine bestimmte rasche Folge von Interrupts), sei es, weil die Anwendungssituationen noch nicht bekannt sind (z. B. die Lage kurz vor der Kernschmelze eines Atomkraftwerks).
- Im Test wird die Funktionalität geprüft. Eventuell werden (eher zufällig) auch Mängel der Bedienschnittstelle oder des Verhaltens unter Last bemerkt. Alle anderen Eigenschaften des Codes, vor allem die Wartbarkeit, sind völlig ausgeblendet.
- Nur der Code ist dem Test zugänglich, fast alle anderen Dokumente bleiben beim Testen ungeprüft. Testen fördert also die früher übliche Fixierung der Entwickler auf den Code und die Geringschätzung der übrigen Dokumente.
- Der Test zeigt die Fehlerursache nicht. Bis zur Analyse der Testresultate ist nicht einmal klar, ob der Fehler im Prüfling liegt.

19.2 Einige spezielle Testbegriffe

Tests kann man nach verschiedenen Kriterien klassifizieren. Dabei gibt es unter den Begriffen starke Abhängigkeiten.

19.2.1 Klassifikation nach den Grundlagen des Tests

Für die Auswahl der Testfälle kann sich der Tester auf drei Informationsquellen stützen: auf die Spezifikation, auf den Code des Prüflings und auf Informationen, die in früheren Programmläufen gesammelt wurden. Außerdem hilft ihm seine Erfahrung.

- Werden die Testfälle auf Basis der in der Spezifikation geforderten Eigenschaften des Prüflings ausgewählt (z. B. Funktionalität, Antwortzeit), dann spricht man von einem *Black-Box-Test* oder auch von einem *Funktionstest* (Abschnitt 19.5).
- Berücksichtigt man bei der Wahl der Testfälle die innere Struktur des Prüflings und die (durch spezielle Werkzeuge erstellten) Aufzeichnungen früherer Programmläufe, dann handelt es sich um einen *Glass-Box-Test* oder *Strukturtest* (Abschnitt 19.6).

Im Allgemeinen weiß der Tester auf Grund seiner Erfahrung, wo mit Fehlern zu rechnen ist, und er wählt Testfälle so, dass sie die typischen Fehler anzeigen. Bekannte Beispiele sind die sogenannten »off-by-one«-Fehler, die bewirken, dass Schleifen einmal zu oft oder zu wenig durchlaufen werden, oder nicht abgefan-

gene Bedienfehler. Dieses Erraten der Fehler (*error guessing*) hilft vor allem beim Black-Box-Test; der erfahrene Tester entdeckt manche Fehler, die der unerfahrene erst durch einen weit aufwändigeren Glass-Box-Test erkennt.

Ein naiver Test durch Leute, die gar nichts über das Programm wissen, kann Fehler anzeigen, die ein Experte nicht findet. Denn ein Laie erzeugt oft (aus Sicht der Entwickler) völlig unsinnige Eingaben und bizarre Umgebungsbedingungen (z. B. Eingabegerät ausgeschaltet, Datenbank völlig leer), an die bei der Programmierung niemand gedacht hat.

19.2.2 Klassifikation nach dem Aufwand für Vorbereitung und Archivierung

Klassifizieren wir Tests nach dem Aufwand, der für Vorbereitung und Archivierung getrieben wird, dann gibt es die folgenden Formen:

Laufversuch
Der Entwickler übersetzt, bindet und startet sein Programm. Nach einigen Abstürzen und Korrekturen gelingt es, das Programm in ganzer Länge auszuführen und dabei Resultate zu erzielen, die nicht offensichtlich falsch sind.

Wegwerftest
Jemand führt ein Programm aus und gibt dabei spontan oder nach kurzer Überlegung Daten vor. Er betrachtet kurz die Resultate und erkennt in einigen Fällen Fehler. (Das Wort »Wegwerftest« ist hier analog dem Wegwerftaschentuch gemeint: Die Testfälle sind von schlechter Qualität, die Ergebnisse werden nicht archiviert.)

Systematischer Test (vgl. die Definition in Abschnitt 19.1.1)
Jemand – nicht der Autor des Programms – leitet aus der Spezifikation Testfälle ab, führt das Programm mit den Testeingaben aus und vergleicht die Ergebnisse mit den Soll-Resultaten. Prüfling, Randbedingungen, Testfälle und Ist-Resultate werden dokumentiert.

Die Kosten eines systematischen Tests betragen ein Vielfaches der Kosten eines Laufversuchs. Trotzdem ist der systematische Test rentabler, denn er

liefert objektive Aussagen über die durchgeführten Prüfungen,
lässt sich rasch und mit geringen Kosten wiederholen, weil die kreative Leistung dokumentiert ist,
erlaubt, wenn später Fehler auftreten, eine Analyse (»Warum wurde dieser Fehler nicht entdeckt?«), die zur stetigen Verbesserung der Testdatenwahl führt.

Zudem liefert der systematische Test als Nebenprodukt die Daten, die in jeder Prüfung erhoben werden sollten: Mit welchem Aufwand wurde geprüft? Wie viele Fehler wurden gefunden?

Aus diesen Gründen ist der systematische Test letztlich auch aus Kostengründen vorteilhaft. Nachfolgend gehen wir von diesem Ansatz aus.

19.2.3 Klassifikation nach der Komplexität des Prüflings

Wir unterscheiden folgende Tests:

- *Einzeltest*
  Bei diesem Test werden einzelne, überschaubare Programmeinheiten getestet, je nach verwendeter Programmiersprache also z. B. Funktionen, Unterprogramme oder Klassen. Er wird häufig auch als *Unit-Test* bezeichnet.
- *Modultest*
  Dieser Test ähnelt dem Einzeltest, aber der Prüfling ist keine einzelne Programmeinheit, sondern eine aus mehreren Einheiten bestehende Komponente (z. B. ein »package« oder »module«).
- *Integrationstest*
  Im Integrationstest wird geprüft, ob die zusammengesetzten Programmeinheiten richtig interagieren. Er zielt also darauf ab, Fehler in den Schnittstellen und in der Kommunikation zwischen den Teilen zu finden.
- *Systemtest*
  Der Systemtest ist ein Integrationstest auf höchster Ebene, der Kommunikationsprobleme zwischen den Subsystemen anzeigen soll. Gleichzeitig ist dies der einzige Test, der aufdecken kann, dass die geforderte Funktionalität nicht vollständig implementiert wurde.

Im Allgemeinen wird bottom-up, also beginnend mit kleinen Einheiten, getestet.

19.2.4 Klassifikation nach der getesteten Eigenschaft

Neben der Funktionalität gibt es weitere testbare Eigenschaften. Wenn sie seriös geprüft werden sollen, sind dafür spezielle Tests erforderlich.

- *Funktionstest*
  Die wichtigste testbare Eigenschaft eines Programms ist seine Funktionalität, die durch die Anforderungen spezifiziert ist.

  Installationstest
  Es muss möglich sein, die Software mit den gelieferten Anleitungen und Programmen zu installieren und in Betrieb zu nehmen. An die Umgebung (andere Software) und an die Betriebsmittel (z. B. den verfügbaren Speicherplatz) darf die Software nur Ansprüche stellen, die durch die Spezifikation abgedeckt sind.

  Wiederinbetriebnahmetest
  Ebenso wichtig wie die Installation ist die Wiederinbetriebnahme eines Systems, nachdem der Betrieb unterbrochen war. Anders als bei der Installation kann es bei der Wiederinbetriebnahme Probleme durch alte, defekte oder obsolete Daten desselben Programms geben.

Verfügbarkeitstest
Dieser Test prüft, ob das System über die geforderte Dauer ohne Störungen läuft.

Last- und Stresstest
Dabei wird getestet, ob sich das System auch unter hoher oder höchster Belastung so verhält, wie es gefordert war. Je nach Anforderungen wird auch das Verhalten bei Überlast getestet.

Regressionstest
Nach einer Korrektur oder Veränderung des Programms muss man damit rechnen, dass die Programme neue Fehler enthalten. Um diese zu erkennen, vergleicht man im Regressionstest die Resultate des veränderten Programms mit entsprechenden alten Resultaten (siehe Abschnitt 19.1.3).

Generell muss jede testbare Anforderung – nicht nur jede funktionale – durch mindestens einen Testfall abgedeckt werden. Das gilt beispielsweise für maximale Antwortzeiten oder für die minimalen Hardware-Anforderungen. Auch hier liefert der Test natürlich keinen Beweis der Korrektheit.

19.2.5 Klassifikation nach den beteiligten Rollen

Besonders bei Software-Produkten spricht man von Alpha- und Beta-Tests. Dagegen ist der Abnahmetest charakteristisch für Auftragsprojekte.

Alpha- und Beta-Test
Als Alpha-Test wird der Test eines Software-Produkts beim Hersteller bezeichnet. Die Software kann dabei noch erhebliche Mängel haben. Sind diese Mängel behoben, so wird das Produkt im Beta-Test speziellen Kunden zur Verfügung gestellt, damit sie es entsprechend seinem Zweck benutzen. Diese Kunden genießen früher als andere die Vorteile eines neuen Produkts, müssen dafür aber auch mehr oder minder gravierende Mängel in Kauf nehmen. Ihre Erfahrungen werden vom Hersteller ausgewertet, um das Produkt zu verbessern.

Abnahmetest
Der Akzeptanz- oder Abnahmetest ist aus Sicht des Software-Herstellers kein Test, sondern eine Vorführung. Er soll zeigen, dass sich das System so verhält, wie es der Vertrag, die Spezifikation verlangt. Der Kunde wird allerdings versuchen, Schwachstellen zu erkennen. Dazu kann er eigene Testfälle mitbringen. Besteht die Software den Abnahmetest, so werden in der Regel Zahlungen des Kunden fällig, und die Gewährleistungsfrist beginnt.

19.3 Die Testdurchführung

Wie wir in Abschnitt 19.2.2 beschrieben haben, ist ein systematischer Test aus vielen Gründen einem Laufversuch oder Wegwerftest vorzuziehen. Nachfolgend stellen wir den prinzipiellen Ablauf vor, der jedem systematischen Test zu Grunde liegt.

19.3.1 Der prinzipielle Testablauf

Jeder systematische Test besteht im Kern aus den Schritten Vorbereitung, Ausführung und Auswertung. Planung und Analyse sind nicht minder wichtig, stehen aber getrennt davon am Anfang und am Ende des Projekts (Abb. 19–3).

Weil der Test eines Systems aus mehreren einzelnen Tests besteht und insgesamt erhebliche Ressourcen beansprucht, müssen die Tests sorgfältig geplant und aufeinander abgestimmt werden. Dabei ist zu entscheiden, welche Eigenschaften durch welche Tests geprüft werden sollen, es sind also die Testziele und die Teststrategie zu bestimmen. Auf dieser Basis wird jeder einzelne Test geplant: Die erforderlichen Schritte werden festgelegt, die benötigten Ressourcen und die Dauer werden ermittelt.

Eine Analyse der Testberichte sollte Teil des Projektabschlusses sein. Hier ist zu prüfen, ob die Teststrategie und der Testaufwand sinnvoll gewählt waren. Die dokumentierten Ergebnisse der Analyse fließen in Maßnahmen zur Verbesserung des Entwicklungsprozesses und ganz speziell der Tests ein.

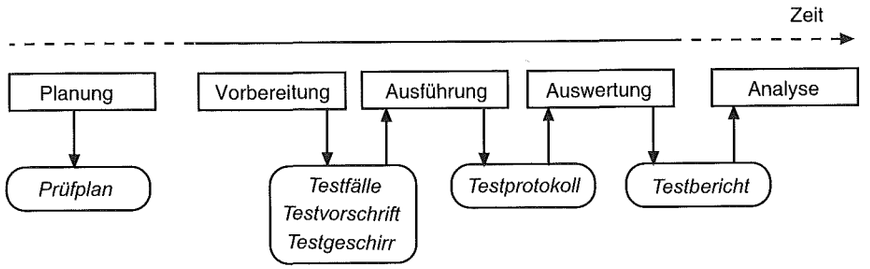

Abb. 19–3 *Prinzipieller Testablauf*

19.3.2 Testvorbereitung

In der Testvorbereitung wird die eigentliche intellektuelle Leistung des Testens erbracht. Je gründlicher sie durchgeführt wird, desto einfacher wird die Testausführung. Im Extremfall kann die Ausführung ganz automatisiert werden.

Der wichtigste und umfangreichste Teil der Vorbereitung ist die Auswahl und Spezifikation der Testfälle. Dazu ist zunächst festzulegen, welche Testgüte er-

reicht werden soll. Anschließend werden die Testfälle möglichst systematisch gewählt; wir gehen darauf in den Abschnitten 19.4 bis 19.6 näher ein. Die Testfälle sind in sogenannten Testszenarien zu gruppieren und sollten nach Prioritäten geordnet werden, damit bei der Testdurchführung die wichtigsten Testfälle auch dann ausgeführt werden, wenn der Test unter Termindruck verkürzt werden sollte, was in der Praxis eher die Regel als die Ausnahme ist.

Daneben muss das benötigte *Testgeschirr*, das ist die Umgebung, um den Test durchzuführen, geschaffen werden. Zum Testgeschirr gehören die *Testtreiber* (test drivers) und die *Platzhalter* (stubs), die die fehlenden Komponenten vertreten.

test driver — A software module used to invoke a module under test and, often, provide test inputs, control and monitor execution, and report test results. Syn: test harness.

stub — (1) A skeletal or special-purpose implementation of a software module, used to develop or test a module that calls or is otherwise dependent on it.

(2) A computer program statement substituting for the body of a software module that is or will be defined elsewhere.

IEEE Std 610.12 (1990)

Ein Testtreiber versorgt den Prüfling mit Testeingaben, ein Platzhalter steht für eine Komponente, die vom Prüfling benötigt wird, aber (noch) nicht integriert ist. Ein Platzhalter liefert entweder vordefinierte Werte oder simuliert (meist nur oberflächlich) das Verhalten der fehlenden Komponente. Da Platzhalter, vor allem solche mit einer gewissen Funktionalität, beträchtlichen Aufwand verursachen, sollte der Test so angelegt werden, dass möglichst wenige davon benötigt werden (siehe Abschnitt 20.2).

Größere Einheiten erfordern oft zusätzliche Hard- und Software, z. B. eine Datenbank in definiertem Zustand oder einen Motorprüfstand, der die vom Prüfling zu verarbeitenden Messdaten in Echtzeit liefert. Aber auch spezielle Testwerkzeuge, die den Test unterstützen, gehören zum Testgeschirr. Bei sicherheitskritischen Anwendungen kann das Testgeschirr bis zu einem Drittel des Gesamtaufwands beanspruchen. Darum wird es nicht weniger sorgfältig erstellt und verwaltet als das eigentliche Produkt. Jeder Entwickler sollte daran denken, dass das Testen mit der Auslieferung der Software nicht endgültig beendet ist, sondern während der Wartung immer wieder stattfindet.

Das Testvorgehen, also die Sequenz der im Test durchzuführenden Schritte, wird so festgelegt, dass die Tests in möglichst rascher Folge und mit möglichst wenigen Eingriffen und Änderungen durchgeführt werden können. Beispielsweise können durch die Testfälle die Belegungen der Datenbank aufgebaut werden, die in weiteren Testfällen vorausgesetzt werden. Auf diese Weise lassen sich die »Rüstzeiten« minimieren. Das Testvorgehen wird in der Testvorschrift dokumentiert.

19.3.3 Testausführung

Ist die Testumgebung fertiggestellt, kann der Prüfling aus der Konfigurationsverwaltung kopiert und in der Testumgebung installiert werden. Nachdem sichergestellt wurde, dass der Prüfling in der Testumgebung tatsächlich ausgeführt werden kann (was nicht immer der Fall ist), kann die Testausführung nach der Testvorschrift beginnen. Dabei wird genau protokolliert, welche Testfälle ausgeführt und welche Ergebnisse dabei erzielt wurden. Das Testprotokoll ist ein wichtiges Dokument der Testausführung, dem nicht nur die Ist-Resultate, sondern auch die exakten Angaben zu allen beteiligten Software-Einheiten (Prüfling, Testdaten usw.) zu entnehmen sind.

Soweit der Test nicht automatisiert ist, also vor allem bei interaktiven Programmen, wird der Tester offensichtliche Fehler sofort erkennen und die betreffenden Resultate im Protokoll kennzeichnen; weitere Konsequenzen sollten zu diesem Zeitpunkt nicht gezogen werden. Insbesondere sollte der Tester weder den Test abbrechen noch versuchen, den Fehler zu beheben, sondern wenn möglich den Test vollständig durchführen.

Natürlich kann es vorkommen, dass ein schwerwiegender Fehler eine Fortsetzung unmöglich macht. Wenn das Programm bei der ersten Eingabe abstürzt, muss der Test möglicherweise abgebrochen werden. In diesem Falle hat die Qualitätssicherung in der Entwicklung versagt. Der Test und die daran geknüpfte Korrektur der Programme soll und kann aus guter Software sehr gute Software machen, nicht aber einer Programmruine Leben einhauchen.

Der Prüfling wird also während des Tests nicht verändert oder korrigiert, sodass der gesamte Test mit einer ganz bestimmten Version durchgeführt wird. Auch darf der Prüfling nicht speziell für den Test modifiziert werden, er kann aber feste, speziell den Test unterstützende Bestandteile haben (wenn z. B. der normalerweise nicht erkennbare Zustand des Systems überprüft werden soll). Für die Testausführung fordern wir also:

- Keine spezielle Testvariante des Prüflings!
- Kein Abbruch des Tests, wenn Fehler erkannt wurden!
- Keine Modifikationen des Prüflings im Test!
- Kein Wechseln zwischen Test und Debugging!

Die Einhaltung dieser Regeln hat folgende Vorteile:

- Der Aufwand für den Test lässt sich recht genau abschätzen.
- Der Test deckt die grundsätzlichen Mängel auf.
- Es werden keine kumulativen Korrekturen durchgeführt.
- Das Testgeschirr wird effizient genutzt.
- Der Prüfling wird nicht durch vergessene Zusätze für die Fehlersuche beschädigt.
- Die erzielten Ergebnisse betreffen ein ganz bestimmtes Programm, im letzten Test das später auszuliefernde Produkt.

Wie schon erwähnt, ist das Testprotokoll ein wichtiges Ergebnis. Es belegt, mit welchem Testgeschirr an welchem Prüfling (in welcher Version und Variante) welche der Testfälle ausgeführt und welche Ergebnisse dabei erzielt wurden. Das Protokoll wird darum wie alle anderen Dokumente der Konfigurationsverwaltung unterstellt.

19.3.4 Testauswertung

Der letzte Schritt beim Testen ist der Vergleich der Ergebnisse mit den Soll-Resultaten. Wird der Test Schritt für Schritt interaktiv durchgeführt, dann werden Ist- und Soll-Ergebnis für jeden Testfall sofort verglichen. Wenn der Test automatisiert ist, übernimmt ein spezielles Werkzeug den Vergleich von Soll- und Ist-Resultaten. In beiden Fällen muss sichergestellt sein, dass Abweichungen nicht nur zufällig auffallen, sondern systematisch entdeckt werden. Dabei sollten auch unbeabsichtigte Effekte des Programms erkannt werden. Wenn beispielsweise eine Information aus einer Datenbank abgerufen wird, erkennt man leicht, ob das Resultat den Erwartungen entspricht. Man kann aber im Allgemeinen nicht ausschließen, dass der Inhalt der Datenbank anschließend verändert ist. Eine solche Kontrolle, ob das Programm keine unbeabsichtigten Nebenwirkungen hat, ist meist sehr viel schwieriger als die Kontrolle der geforderten Wirkungen.

Jede erkannte Abweichung, die nicht als falsch positiv klassifiziert werden kann (siehe Abschnitt 13.4.2), löst eine Problemmeldung aus und liefert damit die notwendigen Informationen für die Fehlersuche und -behebung.

Das Ergebnis der Testauswertung ist der *Testbericht*. Er enthält Verweise auf alle den Test betreffenden Dokumente (insbesondere auf das Testprotokoll) und administrative Angaben (z. B. die Namen der beteiligten Personen, die Anzahl und Schwere der gefundenen Fehler und den benötigten Aufwand). Die zusammenfassende Schlussbewertung des Prüflings gibt auch Auskunft darüber, ob das Testendekriterium (Abschnitt 19.3.5) erreicht wurde.

19.3.5 Das Testendekriterium

In der Praxis werden Tests zwar eingeplant, die dafür vorgesehenen Zeiten werden aber oft als Puffer behandelt, die die in der Entwicklung eingetretenen Verzögerungen ausgleichen. Das Testendekriterium ist damit: Auslieferungstermin erreicht. Auf diese Weise kommt der Test im wahren Sinne des Wortes zu kurz. Außerdem wird zwar meist der Test geplant, nicht aber die erforderliche Korrektur und die Wiederholung des Tests nach der Korrektur. Damit bleibt den Testern gar keine andere Wahl als der unsystematische Wechsel zwischen Test und Debugging.

Besser ist es, ein sinnvolles Testendekriterium festzulegen. Man kann den Test beenden, wenn

a) alle spezifizierten Testfälle ohne Befund absolviert wurden,
b) er x Stunden (Tage, Wochen) gedauert hat,
c) er den Aufwand y beansprucht hat,
d) n Fehler gefunden sind,
e) z Stunden (Tage, Wochen) lang kein Fehler mehr entdeckt wurde,
f) die durchschnittlichen Testkosten für die Entdeckung eines Fehlers x Euro übersteigen (siehe Abb. 19–4).

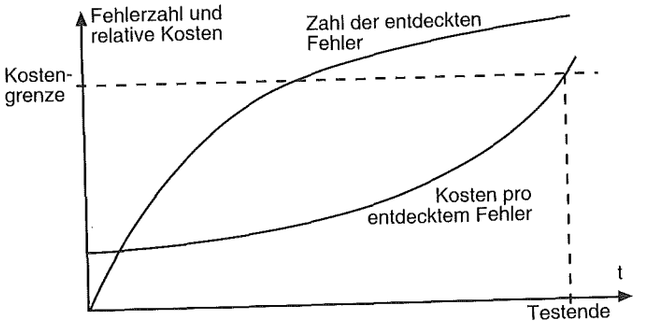

Abb. 19–4 *Kosten als Testendekriterium*

Die Zahlenwerte für x, n, z werden auf Grund von Erfahrungen oder Schätzungen festgelegt.

Die Kriterien a bis f sind nicht alle gleich sinnvoll und auch nicht mit jedem Testverfahren verträglich. Wenn eine abgeschlossene Menge von Testdaten vorliegt, ist offensichtlich das Kriterium a zweckmäßig. Das Kriterium d setzt voraus, dass man sehr genau abschätzen kann, mit wie vielen Fehlern zu rechnen ist. Eine untere Grenze kann in Kombination mit b oder c als notwendiges, aber nicht hinreichendes Kriterium für das Testende sehr sinnvoll sein. Kriterium e kommt in Frage, wenn mit Zufallsdaten getestet wird. Das ist nicht nur beim eigentlichen statistischen Testen (siehe Abschnitt 19.7) der Fall, sondern auch, wenn es (wie z. B. beim Test einer Software für eine Motorsteuerung mit Signalen von einem Motor auf dem Prüfstand) Echtzeiteffekte gibt, die nicht deterministisch geplant werden können.

Auch die Testgüte, z. B. die erreichte Überdeckung im Glass-Box-Test (siehe Abschnitt 19.6), liefert sinnvolle Testendekriterien.

19.3.6 Werkzeuge für den Test

Wir gehen hier nicht auf die zahlreichen Werkzeuge ein, die auf dem Markt angeboten werden; sie können grob in die folgenden Kategorien geordnet werden:

Werkzeuge für den Glass-Box-Test, die den Code instrumentieren und Überdeckungsmaße ermitteln; oft bieten sie weitere Metrik-Funktionen an.
Werkzeuge für den Einzel- und Systemtest, die Testfälle und Resultate verwalten und die automatische Durchführung der Tests unterstützen.
Capture/Replay-Werkzeuge zur automatischen Aufzeichnung der Interaktionen, damit auch interaktive Programme automatisch getestet werden können, sowie Hilfsmittel zur Bearbeitung der Skripts, die von den Capture/Replay-Werkzeugen generiert werden.
Hilfsmittel für den Soll-Ist-Vergleich und zur Verwaltung von Testfällen für den interaktiven Test (»manual testing«).
Spezialwerkzeuge, beispielsweise Werkzeuge für den Last- und Stresstest oder Werkzeuge für den Test von eingebetteter Software.

Zumindest ein Werkzeug für den Glass-Box-Test gehört zwingend zur Grundausstattung des Testers; der Regressionstest muss durchgängig mit Werkzeugen unterstützt werden.

Wie bei allen Werkzeugen im Software Engineering ist die Literaturlage betrüblich; die meisten der im Web reichlich angebotenen Arbeiten sind alles andere als neutrale Informationen. Eine sehr umfangreiche Liste meist kommerzieller Testwerkzeuge findet man beispielsweise auf den Webseiten der IMBUS AG (Testtools, o.J.); Informationen über Open-Source-Testwerkzeuge werden auf den von M. Aberdour erstellten Webseiten angeboten (Open-Source-Testtools, o.J.).

19.4 Die Auswahl der Testfälle

Für jeden Test muss eine sinnvolle Menge von Stichproben (Testfällen) ausgewählt werden; diese Auswahl ist die zentrale Aufgabe des Testers. Dabei versucht er, mit einer möglichst kleinen Menge von Testfällen möglichst vielen Fehlern auf die Spur zu kommen.

19.4.1 Der Testfall

Ein Testfall ist gut, wenn er mit hoher Wahrscheinlichkeit einen noch nicht entdeckten Fehler aufzeigt; er ist dann erfolgreich, wenn er einen noch nicht entdeckten Fehler nachweist. Ein idealer Testfall ist

repräsentativ, d. h., er steht stellvertretend für viele andere Testfälle,
fehlersensitiv, d. h., er hat nach der Fehlertheorie eine hohe Wahrscheinlichkeit, einen Fehler anzuzeigen,
redundanzarm, d. h., er prüft nicht, was auch andere Testfälle schon prüfen.

Die Praxis verlangt auch hier Kompromisse; das Ziel, möglichst viele Fehler zu entdecken, veranlasst uns, bei den genannten Merkmalen Abstriche zu machen.

Damit Testfälle wiederverwendet werden können, müssen sie exakt dokumentiert werden. Zu jedem Testfall sind die folgenden Informationen anzugeben:

- der Anfangszustand der Umgebung, eventuell auch des Prüflings,
- die Werte aller Eingabedaten,
- die notwendigen Bedienungen,
- die erwarteten Ausgaben,
- wenn nötig die Maßnahmen, um das System nach dem Test in einen definierten Zustand zu bringen.

Zusätzlich sollten für jeden Testfall die durch ihn abgedeckten Anforderungen angegeben werden. Die vollständig definierten Testfälle werden in der Testfallspezifikation zusammengefasst. Es hat sich bewährt, die Testfälle nicht nur mit Prioritäten zu versehen, sondern sie auch inhaltlich zu strukturieren. Eine dreistufige Hierarchie, bestehend aus Testgruppe, Test-Suite und Testfall, ist zweckmäßig.

19.4.2 Testkonzepte

Die Ansätze, um Fehler in Programmen durch Testen zu finden, lassen sich drei Kategorien zuordnen, die durch unterschiedliche Modelle der (möglicherweise fehlerhaften) Programme gekennzeichnet sind:

a) Ein Programm wird als eine Menge von Elementen betrachtet, wobei verschiedene Elemente auf verschiedenen Abstraktionsebenen in Frage kommen, z. B. Anweisungen oder Funktionen im Sinne der Spezifikation. Nachdem man eine dieser Mengen gewählt, jedes Element darin (also jede Anweisung bzw. jede Funktion) geprüft und korrekte Resultate erhalten hat, erwartet man, dass kein Element defekt ist und zu einem Fehler führt.
b) Wird ein Programm mit bestimmten Daten ausgeführt, so entsteht eine Folge von Schritten, die man als *Programmpfad* bezeichnet. Nachdem man jeden möglichen Pfad mindestens einmal getestet und korrekte Resultate erhalten hat, erwartet man, dass keiner der Pfade falsch angelegt ist und darum zu einem falschen Ergebnis führt.
c) Wird ein Programm mit Daten getestet, die gleiche statistische Eigenschaften haben wie die Daten, die später beim Gebrauch des Programms verwendet werden, kann aus den Resultaten des Tests eine (statistische) Aussage über die Zuverlässigkeit des Programms abgeleitet werden.

Alle drei Ansätze beruhen auf Erfahrungen außerhalb des Software Engineerings:

a) *Wenn im Auto der Lüfter läuft, ist der Radioempfang gestört.*
Hier ist anscheinend eine Komponente schuld.

b) *Immer, wenn ich nach einer langen Fahrt das Auto kurz abgestellt habe, kann ich den Motor nicht mehr starten.*
Hier führt offenbar ein bestimmter Ablauf (Pfad) zu einem Fehler.

c) *Im Kurzstreckenverkehr ist das Auto sehr zuverlässig.*
Hier schließen wir (unter der Voraussetzung, dass weiterhin die gleichen Randbedingungen gelten) aus einer Erfahrung auf die Zukunft.

Keiner der Ansätze lässt sich für normale Programme so umsetzen, dass die oben formulierten Ansprüche wirklich erfüllt werden; die Ansätze beruhen eben auf vereinfachten Modellen:

a) Die Tatsache, dass ein Element unter bestimmten Bedingungen funktioniert, garantiert leider nicht, dass es immer funktioniert. Das Konzept der Überdeckungen ist blind für die Wechselwirkungen zwischen den Elementen.

b) Wir können die möglichen Pfade im Allgemeinen nicht feststellen, und in den meisten Fällen wären es auch viel mehr, als wir testen können. Zudem ist es möglich, dass auch von zwei Ausführungen auf demselben Pfad die eine korrekte, die andere falsche Resultate liefert.

c) Die statistischen Eigenschaften der Daten lassen sich im Allgemeinen nicht voraussagen, sie können sich auch im Laufe der Zeit verändern. Bei einem Test mit Zufallsdaten ist es schwierig bis unmöglich, die Soll-Resultate festzustellen. Zudem gibt eine statistische Aussage zur Zuverlässigkeit im Einzelfall keine Sicherheit.

Trotzdem geben uns die drei Modelle wichtige Hinweise, wie ein Test möglichst erfolgreich angelegt werden kann. Ansatz c führt zum statistischen Test (Abschnitt 19.7), Ansatz a zu den Überdeckungsmaßen, die im Black-Box-Test, vor allem aber im Glass-Box-Test (Abschnitt 19.6) verwendet werden. Als *(Test-)Überdeckung* bezeichnet man den Anteil der Elemente, die im Test benutzt wurden, an der Gesamtzahl.

Ansatz b führt zum Konzept der Äquivalenzklassen (siehe Abschnitt 19.4.4) und damit zum Black-Box-Test (Abschnitt 19.5); in Abschnitt 19.6.6 wird erläutert, warum die in der Literatur übliche Zuordnung der Pfadüberdeckung zum Glass-Box-Test nicht sinnvoll ist.

19.4.3 Starke und schwache Äquivalenz

Wir wollen mit wenigen Testfällen eine möglichst große Wirkung erzielen, d. h. viele Fehler entdecken. Mit anderen Worten: Wir versuchen, keine überflüssigen Testfälle zu bearbeiten. Myers hat dazu das Konzept der Äquivalenzklassen entwickelt (Myers, 1979).

Nehmen wir beispielsweise an, dass ein Programm Personennamen verwaltet. Ein eingegebener Name wird gespeichert; wenn er ein zweites Mal eingegeben wird, soll eine Fehlermeldung erzeugt werden. Im Test geben wir einen Namen (»Meier«) ein und erwarten, dass er akzeptiert wird. Geben wir diesen Namen erneut ein, so sollte die Fehlermeldung erscheinen. Natürlich können wir den Test mit dem Namen »Mayer« wiederholen. Sehr wahrscheinlich wird der Erfolg oder Misserfolg des Tests der gleiche sein wie für »Meier«. Denn es gibt keinen Grund zur Vermutung, dass der Ablauf für »Mayer« in irgendeiner Weise anders ist als für »Meier«: Entweder treten bei keinem der beiden Namen Fehler auf, oder bei beiden. Die beiden Testsequenzen sind also – wenn unsere Annahmen stimmen – *äquivalent*. Von mehreren äquivalenten Testfällen brauchen wir nur einen zu testen, denn das Resultat ist auch für alle anderen Fälle in dieser *Äquivalenzklasse* gültig.

Wir definieren: Zwei Testfälle f_1 und f_2 sind im Hinblick auf den Erfolg des Tests *stark äquivalent*, wenn sie austauschbar sind, weil beide geeignet oder beide ungeeignet sind, einen bestimmten Fehler anzuzeigen[1].

Wenn die Elemente einer Menge möglicher Testfälle $F = \{f_1, f_2, ..., f_n\}$ für ein Programm P stark äquivalent sind, reicht es aus, einen einzigen Repräsentanten von F für den Test auszuwählen. F ist eine (starke) *Äquivalenzklasse* der Testfälle für P. Ein »wasserdichtes« Verfahren, um alle starken Äquivalenzklassen zu identifizieren, wäre so etwas wie der Stein der Weisen für den Test. Den gibt es leider nicht, wenn man von der irrealen Idee des vollständigen Tests absieht.

Ein vollständiger Test auf der Basis der starken Äquivalenzklassen erfordert, dass aus jeder Klasse ein Testfall gewählt wird. Da die Fälle in derselben Klasse äquivalent sind, decken alle so erzeugten Testdaten *dieselben* Fehler auf. Da jede Klasse berücksichtigt wird, also (falls das Programm Fehler enthält) auch jede Klasse, die einen Fehler aufdeckt, decken die Testdaten *alle* Fehler auf.

Da wir keine Möglichkeit haben, die Klassen starker Äquivalenz zuverlässig zu erkennen, behelfen wir uns mit *schwacher Äquivalenz*: Zwei Testfälle sind schwach äquivalent, wenn wir Gründe haben, starke Äquivalenz zu vermuten. Solche Gründe sind zahlreich: Wenn z. B. zwei Eingaben den gleichen Ablauf des Programms bewirken, die gleiche Funktion des Programms in Anspruch nehmen oder die gleiche Fehlermeldung hervorrufen, können wir Äquivalenz vermuten. Jede schwache Äquivalenz beruht auf einer spekulativen Fehlertheorie: Wir rechnen damit, dass die Klassen starker Äquivalenz bestimmte Muster aufweisen, die wir erraten können. Wenn verschiedene Fälle denselben Programmablauf (Pfad) hervorrufen, sind sie wahrscheinlich äquivalent. Dies ist der Zusammenhang zwischen Pfadüberdeckung und Äquivalenzklassen.

Wenn ein Programm nach Eingabe des Datums »31. Juni 2010« eine Fehlermeldung bringt, wird es auch den 31. September 2010 nicht akzeptieren, und

---

[1] Unsere Begriffe der starken und schwachen Äquivalenz haben nichts zu tun mit dem in Jorgensen (1995) beschriebenen Ansatz des starken und schwachen Äquivalenzklassentests.

umgekehrt. Hinter dieser Aussage steckt kein *Wissen*, sondern eine *Hypothese*: Wir vermuten, dass der 31. September genau denselben Programmablauf, dieselben Entscheidungen auslöst wie der 31. Juni. Diese Spekulation kann falsch sein, dann stimmen schwache und starke Äquivalenz nicht überein, und wir entdecken Fehler *nicht*, die wir hätten entdecken können. Wenn der Programmierer geglaubt hatte, dass alle geradzahligen Monate 31 Tage haben (wie August, Oktober und Dezember »beweisen«), sind die beiden Datumsangaben *nicht* äquivalent. Bei den Namen im ersten Beispiel können wir vermuten, dass auch »Müller« in der Äquivalenzklasse liegt, zu der »Meier« und »Mayer« gehören. Es ist aber denkbar, dass wegen des Umlauts oder des Doppelkonsonanten der Name »Müller« anders behandelt wird und darum *nicht* mit Meier« und »Mayer« in einer Klasse liegt. Natürlich könnte es auch ganz anders sein, etwa wenn das »y« eine spezielle Reaktion hervorruft oder wenn der Programmierer Mayer hieß und die »originelle« Idee hatte, bei Auftreten seines Namens eine besondere Reaktion des Programms auszulösen.

Unsere Erfahrung beim Test von Programmen zeigt, dass die Behandlung von Eingaben, die an den Grenzen von Wertebereichen liegen, häufig fehlerhaft ist. Darum werden die Grenzwerte als eigene Äquivalenzklassen behandelt. Wenn wir mit REAL-Zahlen arbeiten, müssen wir statt des (im Allgemeinen nicht darstellbaren) Nachbarwertes einen geringfügig größeren oder kleineren Wert verwenden.

19.4.4 Anwendung der Äquivalenzklassen zur Testdatenauswahl

Die Bildung der Äquivalenzklassen (für gültige und vor allem ungültige Eingaben) erfolgt nach Erfahrung und Intuition, also heuristisch! Um die Äquivalenzklassen für die Eingaben zu bestimmen, geht man wie folgt vor:

1. Der Spezifikation werden die Eingabegrößen und ihre Gültigkeitsbereiche entnommen. Die Grenzen der Eingabebereiche, die geordnet sind, trennen Äquivalenzklassen gültiger und ungültiger Eingaben.
2. Wenn man vermutet, dass die Werte einer Äquivalenzklasse ungleich behandelt werden, teilt man die Klasse in neue Unterklassen auf.
3. Werte an Bereichsgrenzen, die erfahrungsgemäß oft falsch verarbeitet werden, sollten gesondert behandelt, also speziellen Äquivalenzklassen zugeordnet werden.
4. In jeder Äquivalenzklasse wird ein Testfall gewählt (Eingabe und Soll-Resultat).

Tabelle 19–2 zeigt einige einfache Beispiele. Bei der Enumeration wurde unterstellt, dass keine unterschiedliche Behandlung der drei Werte zu vermuten ist. Für die REAL-Zahlen wurde eine fiktive Darstellungsgenauigkeit angenommen.

Bereich gültiger Daten	Klasse gültiger Eingaben	Klassen ungültiger Daten	Grenzfälle	Repräsentant
Die ganzen Zahlen von a = 5 bis n = 555	a ≤ E ≤ n		5 555	100 5 555
		E < a		1
		E > n	4 556	999 4 556
Natürliche Zahl (inkl. 0) bis max. n = 1000	E > n		0 1000	227 0 1000
		E < 0		-171
		E > n	-1 1001	1033 -1 1001
Enumeration (Alpha, Beta, Gamma)	E ∈ {Alpha, Beta, Gamma}			Gamma
		E ∉ {Alpha, Beta, Gamma}		Omega
Bereich der REAL-Zahlen mit Betrag bis 1	-1,0 ≤ E ≤ 1,0		-1,0 1,0	0,555 -1,0 1,0
		E < -1,0		-2,500
		E > 1,0	-1,0001 1,0001	1,015 -1,0001 1,0001

Tab. 19–2 *Bildung von Äquivalenzklassen für den Eingabewert E*

Betrachten wir als weiteres Beispiel ein Programm, das in Zeichenreihen, die zwischen 1 und 100 Zeichen lang sein können, die Kleinbuchstaben durch Großbuchstaben ersetzen soll. Eine extrem simple Fehlertheorie wäre, dass die Umwandlung entweder immer versagt oder immer korrekt funktioniert. Dann gehören alle möglichen Eingaben zur selben Klasse. Da wir Fehler nur erkennen können, wenn eine Umwandlung stattfindet, wählen wir einen Testfall mit Kleinbuchstaben, beispielsweise »XYZ++abc«; das Soll-Resultat ist »XYZ++ABC«.

Vermutlich werden Zeichenreihen ohne Kleinbuchstaben anders behandelt, wir schaffen darum eine zweite Klasse, die wir durch den Testfall »$$RWTH$$« repräsentieren (Soll-Resultat = Eingabe).

Natürlich lassen sich die Fehler in der Regel nicht so leicht aufdecken, wir nehmen also unsere Erfahrung hinzu. Die Grenzfälle erfordern in den Programmen oft eine spezielle Bearbeitung. Wir identifizieren also zwei weitere Klassen, nämlich Zeichenreihen minimaler und maximaler Länge. Schließlich ist in vielen Fällen robustes Verhalten des Programms gefordert, es soll also sinnvoll reagieren, wenn die Längenbedingung verletzt wird. Damit entstehen zwei weitere

Klassen, die leere Zeichenreihe und eine mit mehr als 100 Zeichen; insbesondere 101 Zeichen sollten getestet werden.

Wenn wir vermuten, dass die Reihenfolge der Zeichen eine Rolle spielt, dass also »aB« möglicherweise anders behandelt wird als »Ba«, dann ergeben sich auch dadurch weitere Klassen. Und schließlich müssen wir überlegen, ob die Kombination der genannten Klassen neue Klassen ergibt oder nicht, ob wir also beispielsweise zwischen maximal langen Zeichenreihen mit und ohne Kleinbuchstaben unterscheiden müssen.

Das Beispiel zeigt: Die Äquivalenzklassen können nicht einfach mechanisch festgelegt werden, Intuition und Erfahrung sind von großer Bedeutung. Und es zeigt auch, dass wir niemals sicher sein können, alle Klassen der starken Äquivalenz erkannt zu haben. Wird die Zeichenreihe beispielsweise vom Programm in Abschnitte zu je 32 Zeichen gegliedert, dann muss man mit Problemen rechnen, wenn gerade 32, 64 oder 96 Zeichen eingegeben werden, auch 33, 65 und 97 wären dann speziell zu betrachten.

Ein letztes (nicht erfundenes) Beispiel zeigt, warum wir nie sicher sein können, alle Klassen der starken Äquivalenz entdeckt zu haben: Ein kleines ADA-Demonstrationsprogramm zur rekursiven Berechnung der Fakultätfunktion wurde mit den Mitteln des Exception Handlings gegen alle Fehleingaben abgesichert, also gegen Eingabe negativer oder gebrochener Zahlen und gegen die Eingabe von Text. Zusätzlich wurde sichergestellt, dass die Berechnung der Fakultät bei der Eingabe einer großen Zahl nicht zu einem Zahlenüberlauf führt. Mit den Eingaben -5, 3.14, Blabla und 1000 funktionierte alles einwandfrei. Trotzdem brachte ein naiver Benutzer das Programm sofort zum Absturz: Er hatte als Argument 1000000 eingetippt. Aber war dieser Fall nicht abgefangen und getestet worden (durch die Äquivalenzklasse »Zahl zu groß« mit dem Repräsentanten 1000)? Nein. Denn die Eingabe einer sehr großen Zahl führt, bevor es zum Zahlenüberlauf kommt, zu einem Kellerüberlauf (stack overflow). Eine Million Rekursionsstufen waren zu viel. Von der Klasse »Zahl zu groß« musste also eine Klasse »Zahl viel zu groß« abgespalten werden.

19.5 Der Black-Box-Test

Bei einem Black-Box-Test betrachtet man das Programm als Monolithen, über dessen innere Beschaffenheit man nichts weiß und nichts wissen muss. Man prüft, ob das Programm tut, was die Spezifikation verlangt. Dies ist die wichtigste Form des Tests. Alle anderen Tests ergänzen den Black-Box-Test, sie können ihn nicht ersetzen.

Testfälle für den Black-Box-Test sollten vorbereitet werden, sobald die Spezifikation vorliegt (siehe Punkt 10 in Abschnitt 16.9). Dadurch ist nicht nur gewährleistet, dass die Testfälle rechtzeitig verfügbar sind, sondern die Spezifikation wird einer zusätzlichen Prüfung unterzogen, die oft Fehler und Lücken

aufdeckt. Denn wer einen Testfall, also Eingabe und Soll-Resultat entwirft, muss genau hinschauen und kann nicht übersehen, wenn die Spezifikation seine Fragen nicht oder nicht eindeutig beantwortet.

19.5.1 Die Ziele des Black-Box-Tests

Ein umfassender Black-Box-Test sollte

- alle Funktionen des Programms aktivieren (Funktionsüberdeckung),
- alle möglichen Eingaben bearbeiten (Eingabeüberdeckung),
- alle möglichen Ausgabeformen erzeugen (Ausgabeüberdeckung),
- die Leistungsgrenzen ausloten,
- die spezifizierten Mengengrenzen ausschöpfen,
- alle definierten Fehlersituationen herbeiführen.

Die Funktionsüberdeckung geht von der Menge der Funktionen aus, die ein Programm anbietet. Die Eingabeüberdeckung ist auf die Menge möglicher Eingaben bezogen, die Ausgabeüberdeckung auf die Menge möglicher Ausgaben. Achtung, es geht hier nicht um alle im Detail unterschiedlichen Ein- und Ausgaben, also um vollständigen Test, sondern um die Klassen der Ein- und Ausgaben. Zum Beispiel fallen zwei Namen in dieselbe Klasse, wenn es nicht aus speziellen Gründen geraten erscheint, sie zu unterscheiden. Oft sind Ein- und Ausgabeüberdeckung durch die Funktionsüberdeckung impliziert, aber nicht immer.

Beispiel: Ein Programm, das Auskunft über Studenten gibt, biete die folgenden Funktionen: (a) Anfangsmeldung, (b) Einlesen einer Matrikelnummer, (c) Ausgabe der über den Studenten gespeicherten Informationen oder (d) einer Fehlermeldung (»Student existiert nicht«), (e) Endemeldung. Statt der Matrikelnummer kann auch der Name eingegeben werden. Wir können also alle Funktionen in Anspruch nehmen und dabei immer nur die Matrikelnummer verwenden. Wir haben dann die Funktionsüberdeckung, aber nicht die Eingabeüberdeckung erreicht.

Alle Funktionen außer d können wir mit einem einzigen Testfall abdecken. Fehlerfälle benötigen typischerweise jeweils einen eigenen Testfall, wenn der Programmablauf im Fehlerfall abgebrochen wird. Wenn ein Programm mit k verschiedenen Fehlerfällen enden kann, brauchen wir mindestens k+1 Testfälle, um die vollständige Funktionsüberdeckung zu erreichen; wenn sich verschiedene Funktionen gegenseitig ausschließen, sind es entsprechend mehr.

Die Grenzfälle (z. B. Namen minimaler und maximaler Länge sowie die angrenzenden Fehlerfälle) werden wie oben beschrieben speziell getestet.