Vorwort

Testen? Diese ganzen Testtechniken und -verfahren? Schön und gut, aber wir haben dazu gerade keine Zeit respektive kein Geld. Komplizierte Testverfahren werden bei uns sowieso nicht angenommen, das Thema ist undankbar.

Natürlich ist Testen sinnvoll und wichtig, gerade auch mittel- und langfristig – wir werden aber ausschließlich am Erfolg unseres Projekts gemessen und das geht nur bis wenige Wochen nach Einführung. Was danach kommt, interessiert nicht. Und wir arbeiten schließlich iterativ-inkrementell, das kompensiert doch den eigentlichen Testbedarf.

Testen ist als Thema in vielen Projekten kaum mehr als Wunschdenken oder Sonntagsreden: Ja, eigentlich, man müsste, sollte usw. Es wird Ausprobieren mit Testen gleichgesetzt und der Einsatz eines Testautomatisierungs-Frameworks wie JUnit als Testsystematik gesehen.

Das Thema ist bekannt, Theorien und Verfahren gibt es seit Jahrzehnten, und Begriffe wie Grenzwertanalyse und Pfadabdeckung sind geläufig. In den letzten Jahren hinzugekommen oder populärer geworden sind beispielsweise Testautomatisierung und eine Reihe durchaus cooler Tools.

Damit sind auch die Anforderungen gestiegen. Moderne Entwurfsmuster führen zu neuen vertrackten Problemen, auch Objektorientierung an sich bringt schon einige besondere qualitätsrelevante Phänomene mit. Und nicht zuletzt führt die suggerierte trügerische Sicherheit von Unit-Tests auch zu weiteren sozialen, psychologischen und organisatorischen Problemen. Gestiegene Komplexität und gestiegener Termin- und Kostendruck führen dazu, dass Testen ein schwieriges Thema bleibt.

Uwe Vigenschow stellt sich dieser Situation, indem er nicht noch ein weiteres Buch mit tollen oder teuren Testtheorien und -verfahren vorstellt, sondern als Ausgangspunkt hat, »gut genug« zu sein. In diesem Sinne ist das vorliegende Buch pragmatisch.

Es greift die aktuellen Probleme und Herausforderungen auf, spricht eine einfache und klare Sprache und konzentriert sich auf wichtige Elemente. Und der Autor macht dies aus einer Position heraus, aus der er weiß, wie sich diese wichtigsten Elemente in die umfassenden gesamten Grundlagen der Disziplin Testen einfügen, denn das Thema verfolgt Uwe Vigenschow schon seit acht Jahren. Er schafft also die Reduktion aufs Wesent-

liche, gut genug sein, mit wenig Aufwand, aber systematisch ausreichend, um gute Effekte zu erzielen. Vielen Dank, Uwe!

Bernd Oestereich

Amrum, Juli 2004

Warum nur und für wen?

Seltsamer Titel, aber ich wollte diesen Abschnitt eben nicht *Einleitung* nennen. Aber warum überhaupt noch ein Buch zum Thema *Testen*? Es gibt doch schon genug. Und gebracht haben die auch nicht viel.

Eben genau darum! Oder um es mit den Worten von Tom DeMarco zu sagen[1]:

> »Ich kritisiere die Qualitätsbewegung nicht, weil sie teuer wäre oder unseren Organisationen zu viel Energie abverlangen würde. Mir macht vielmehr zu schaffen, dass hinter der Qualitätsbewegung mehr Lippenbekenntnis als ernst zu nehmende Hilfe steckt. Darüber hinaus werden echte Qualitätsverbesserungen durch die allgegenwärtigen Qualitätsverbesserungsprogramme [eher] unnötig erschwert.«

Mit diesem Buch möchte ich weg von der Perfektion theoretischer Gedankengerüste, die der Härte der Realität nicht standhalten, hin zu einfachen, wirkungsvollen Verbesserungen im Rahmen unserer täglichen Softwareentwicklung. So wie wir im Alltag nur selten das Optimum anstreben, sondern einer *Gut-genug-Strategie* folgen.

Wir, das sind Software-Entwickler und Projektleiter, die langfristig die Qualität der Software, die wir schreiben, verbessern wollen. Nicht aus Selbstzweck, sondern aus reinem Eigennutz. Wir wollen unter dem hohen Zeitdruck, dem wir ausgesetzt sind, effizient arbeiten, später Änderungen mit einem besseren Gefühl der Sicherheit einbauen können und im weiteren Projektverlauf sowie in der späteren Wartung einfach mehr Spaß an der Arbeit haben. Und da sind wir schon bei dem Zitat von Tom DeMarco. Wir wollen gemeinsam Wege und Optionen diskutieren. Ich werde Ihnen dazu eine Reihe von Optionen vorschlagen, die sich in meiner Praxis bewährt haben und Sie beurteilen das Ganze realistisch, auf die Machbarkeit in Ihrem konkreten Umfeld. Ich bin überzeugt, dass so manches davon Ihren Alltag verbessern wird.

Wir Entwickler leben in dem Luxus, Spaß an unserer Arbeit zu haben. Das motiviert uns [18, 19]. Wir werden für Arbeiten bezahlt, die

[1] Tom DeMarco, Berater und Autor, geb. 1940, aus: Spielräume, 2001 [22]

wir vielleicht auch sonst machen würden. Die ganzen hochwertigen Open-Source-Projekte sind ein Beispiel dafür. Eine bessere Qualität unserer Arbeit kann dazu beitragen, mehr Spaß zu haben. Große bürokratische Regelwerke helfen uns da nicht weiter, obwohl sie prinzipiell in sich stimmig und zielführend sind. Der *Human Factor* wird dabei aber ebenso außer Acht gelassen wie der tägliche Druck im Projektgeschäft.

Vieles von dem, was wir in diesem Buch diskutieren, ist nicht neu, aber in einen anderen Kontext verschoben oder unter einem anderen Licht betrachtet. Bewusst habe ich mich dabei auf wenige grundlegende Qualitätssicherungsmethoden reduziert, die ich für besonders effizient halte. Genauso finden wir diverse technische Themen, die auf den ersten Blick gar nichts mit Qualitätsverbesserung zu tun haben. Eine erfolgreich umgesetzte Gutgenug-Strategie durchdringt eben alle Bereiche und nicht nur die, wo es bequem ist. Und dass eine solche Strategie äußerst erfolgreich sein kann, hat nicht zuletzt die Firma Microsoft seit Jahrzehnten gezeigt.

Wenn wir die hier beschriebenen Wege alle ausgereizt haben, treffen wir uns gerne wieder und diskutieren ein paar neue Ideen. Bis dahin viel Erfolg auf Ihrem Weg. Ach ja, Downloads der Beispiele, ein Errata und weitere Informationen zu diesem Buch finden Sie unter `www.oo-testen.de`.

Uwe Vigenschow

Hamburg, August 2004

Inhaltsverzeichnis

1 Komplexe Systeme führen zu Fehlern

»Das einzige Mittel, den Irrtum zu vermeiden, ist die Unwissenheit.«[1] Für uns bietet diese Erkenntnis kaum eine Möglichkeit, Fehler zu vermeiden. Also wollen wir uns der Problematik von Fehler und Test aktiv stellen. Wir sind beileibe nicht chancenlos bei unserem Kampf für bessere Software.

Komplexe Systeme sind analytisch in begrenzter Zeit nur unvollständig erfassbar. Fehler sind damit zwangsläufig die Folge. Ganz allgemein betrachtet gibt es dafür vier Gründe:

- Kommunikationsprobleme
  - extern, also zwischen Menschen
  - intern, also im internen Kommunikationsprozess bei der Transformation von Sprache in Verständnis
- Gedächtnisprobleme
- Fachliches Problemverständnis
- Hohe Komplexität der Softwarelösung

Betrachten wir die vier Bereiche ruhig noch etwas genauer.

1.1 Kommunikation

Ein einfaches Kommunikationsmodell beschreibt Kommunikation als Folge von Transformationen [79]. Dabei kann es bei jeder Transformation zu Verlusten im Informationsgehalt kommen. Dies erfolgt sowohl zwischen Personen wie auch innerhalb eines Menschen (Abb. 1.1).

Jede Wahrnehmung ist subjektiv. Dazu kommt das Ausfiltern von Informationen aufgrund der physikalischen Einschränkungen unserer Wahrnehmung. Das Wahrgenommene wird dann transformiert und als Erinnerung im Gehirn abgelegt. Bei dieser Transformation helfen uns unsere bisherigen Erfahrungen. Außerdem sind wir begrenzt durch unsere individuelle

[1] Jean-Jacques Rousseau (1712–1778), Schriftsteller und Kulturphilosoph, aus: *Emilie*, ca. 1762

Auffassungsgabe. Beides hilft uns beim schnellen Erfassen, filtert aber erneut Informationsgehalt aus.

Bei der Retransformation aus unserem Gehirn in extern Kommunizierbares, also z. B. Sprache, bleibt erneut einiges auf der Strecke. Besonders bewusst wird uns dies, wenn wir nicht in unserer Muttersprache, sondern in einer Fremdsprache kommunizieren müssen. Wir spüren förmlich, wie Informationsgehalt liegen bleibt. Bei unserem Gesprächspartner spielen sich natürlich dieselben Prozesse ab.

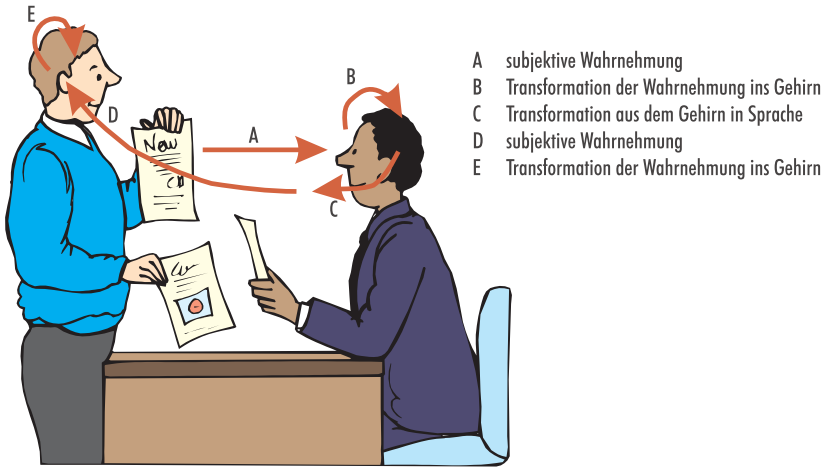

Abbildung 1.1: Ein einfaches Kommunikationsmodell nach Shannon und Weaver [79].

Wir technisch geprägten Menschen reduzieren Kommunikation häufig auf den reinen Informationsgehalt. Dies wird als inhaltlich-sachliche Ebene der Kommunikation bezeichnet. Daneben gibt es aber noch drei weitere Ebenen der Kommunikation, die der Geschäftsordnung, der sozialen Beziehungen und des Unbewussten (Abb. 1.2) [2, 70]. Dieses Kommunikationsmodell wird grafisch in Form eines Eisbergs dargestellt und entsprechend genannt. Schauen wir uns die Ebenen des Eisbergmodells genauer an.

Sachebene: fachlicher Inhalt, Ziele, Verstand, Aufgaben... Wir transportieren hier die Antworten nach dem *was*.

Geschäftsordnung: Befugnisse, Entscheidungsverfahren, Standards, Regeln... Mit der Geschäftsordnung beantworten wir die Frage nach dem *womit*.

Soziale Beziehungen: Gefühle, Erwartungen, Ängste, Anerkennung, Offenheit, Vertrauen...

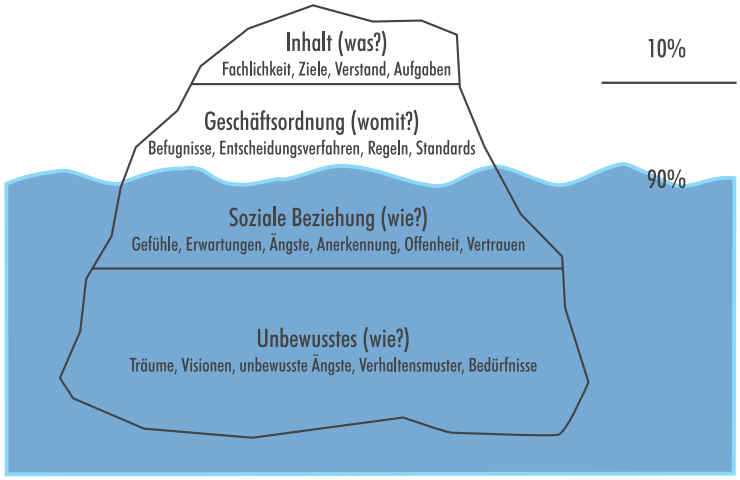

Abbildung 1.2: Eisbergmodell: Kommunikation spielt sich parallel auf vier Ebenen ab [2, 70].

Unbewusstes: Träume, Visionen, unbewusste Ängste oder Verhaltensmuster...

Das Besondere am Eisbergmodell ist weniger die Tatsache der Existenz der vier Ebenen, sondern deren Anteil an der Wichtigkeit für den Transport von Information. Die Sachebene spielt mit ca. 10% nur eine untergeordnete Rolle, der Anteil unterhalb der Wasserlinie macht dagegen über die Hälfte aus. In diesem Zusammenhang gibt es zwei Regeln, die uns im täglichen Leben von Nutzen sein können.

1. Offensichtliche Kommunikationsprobleme auf einer Ebene haben ihre versteckte Ursache häufig in der darunter liegenden Ebene.
2. Kommunikationsprobleme müssen auf der Ebene gelöst werden, auf der sie verursacht werden.

Natürlich könnte man ein eigenes Buch über das Eisbergmodell schreiben. Wichtig ist mir hier, dass wir ein erstes Gefühl bekommen, warum es so schnell zu Kommunikationsproblemen kommen kann, obwohl inhaltlich doch eigentlich alles gesagt wurde. Durch das Verletzen von Konventionen und Regeln oder z. B. durch arrogantes Verhalten verderben wir uns die Möglichkeit, andere Menschen zu überzeugen. Der unbewusste Anteil ist dabei erheblich; manchmal können wir eine bestimmte Person einfach nicht erreichen und nur, weil wir sie durch unser Aussehen und unseren Habitus an ihren alten Chef erinnern, von dem sie im Streit entlassen wurde...

1.2 Gedächtnis

Auch wenn Sie nicht an »Pre-Alzheimer« leiden, können Sie sich nicht alles merken. Ich bin sogar ein sehr vergesslicher Mensch, was sowohl bei der Arbeit als auch in anderen persönlichen Beziehungen immer wieder zu Irritationen führt. Es gibt Techniken, das Gedächtnis zu stärken oder durch Mnemotechniken zu verbessern. Trotzdem können wir uns nicht alles merken. Leider gelingt es uns aber auch nur begrenzt, alles aufzuschreiben und so vor dem Vergessen zu retten.

Wir können nur die wichtigsten Dinge dokumentieren. Dabei sollten wir auch immer einen pragmatischen Kompromiss finden zwischen Aufwand, Umfang und Inhalten, wobei ein besonderes Augenmerk auf der Wartung von Dokumenten liegen muss, um nicht entweder andauernd unsere Dokumente ändern zu müssen oder aber schnell veraltende Texte vorzufinden, die keinen relevanten Praxiswert mehr haben. Die Kunst besteht also im Rahmen der Softwareentwicklung darin, durch den Code, im Code und durch kodierte Testfälle die lokale Dokumentation durch die Arbeitsergebnisse selbst zu erreichen und in externen Dokumenten die übergreifenden Zusammenhänge und langfristig gültigen Aspekte zu behandeln.

Selbst wenn uns das gelingt, brauchen wir die Detailinformationen in unseren Köpfen. Und dort verhält es sich wie mit einer Festplatte, auf der von jemand anderem Dateien gelöscht werden und wir kein Backup haben. Es passieren Fehler.

1.3 Fachlichkeit

Wir sind technische Experten, stecken in den Tiefen unserer Programmiersprachen, Tools und Klassenbibliotheken. Methodisches Arbeiten in den Bereichen der Softwarearchitektur und des Designs sind uns geläufig, wir modellieren mit der Unified Modeling Language (UML) und können Tage bzw. Nächte damit verbringen, Code zu optimieren, die Performance eines Systems zu verdoppeln oder einen vertrackten Fehler durch Analyse des Stackdumps zu finden. Dafür sind wir ausgebildet, und darin haben wir Erfahrung.

Programmierung und die Erstellung von Software sind kein Selbstzweck. Auftraggeber verfolgen fachliche Ziele, und die Anwender und Fachabteilungen, mit denen wir es zu tun haben, um an die Anforderungen zu kommen, die wir umsetzen sollen, haben ganz andere Sorgen. Auch sie sind Fachexperten, nur leider auf ganz anderen Gebieten. So lange, wie wir uns bereits mit der Umsetzung von Entwurfsmustern oder der Implementierung von Mehrschichtarchitekturen befasst haben, haben sie in ihrer eigenen fachlichen Welt gearbeitet.

Um zu verstehen, was wir eigentlich tun sollen, müssten wir eigentlich selbst eine Banklehre gemacht, Versicherungs- oder Speditionskaufmann gelernt, ein Baustatik-, Jura- oder BWL-Studium absolviert haben. Haben wir aber nicht! Entwickler, die lange im selben Umfeld arbeiten, erarbeiten sich nebenbei einen Großteil dieses Wissens, aber eben nicht alles.

Warum ist das überhaupt ein Problem? Wir reden doch miteinander und die Anforderungsgeber sagen uns schon, was sie wollen. In der Praxis treffen wir dabei primär auf drei Probleme:

- Die Anforderungsgeber können nur schwer vermitteln, was sie wollen, da sie sich selbst darüber nur diffus im Klaren sind. Die Abstraktions- und Analysefähigkeit ist leider sehr unterschiedlich verteilt. Häufig muss dies auf Seiten der Entwicklung im Rahmen der Analyse geleistet werden. Jetzt ist aber die Gefahr groß, dass wir aus Unkenntnis fachlicher Zusammenhänge in der Abstraktion wichtige Details übersehen bzw. Zusammenhänge zu einfach sehen.
- Wir sprechen nicht die gleiche Fachsprache (Abb. 1.3). Es ist zwar immer noch Deutsch, aber wir verstehen es trotzdem nur begrenzt. Es findet also ein Transfer statt, bei dem Informationsgehalt verloren gehen kann. Diese Transferverluste können sich später als Fehler rächen. Gemindert werden kann dieser Verlust nur durch einen expliziten Übersetzer. Zusätzlich erfolgt im Rahmen der Analyse eine Abstraktion, die auch zu Fehlern führen kann.
- Die Anforderungsgeber wissen nicht, was technisch möglich ist, und können daher nur im Rahmen ihres technischen Horizonts Vorschläge machen. Um aber von unserer Seite aus Alternativen vorschlagen zu können, müssen wir erstmal verstanden haben, was die Anforderungsseite eigentlich will bzw. braucht. Da wir das aber nur schwer verstehen, drehen wir uns leicht im Kreis.

1.4 Komplexität

Selbst wenn wir die drei zuvor behandelten Bereiche in den Griff bekommen, spielen uns unsere Aufgabenstellungen immer noch einen Streich. Softwareprojekte gehören zu den komplexesten Dingen, die Menschen versuchen. Leider sind wir nur begrenzt in der Lage, Komplexität zu überblicken. Wir versuchen unsere Aufgaben zu zerlegen und so die Gesamtkomplexität zu reduzieren. Dennoch werden uns die Vielfalt und die Abhängigkeiten der Anforderungen wie auch unsere technischen Möglichkeiten immer wieder Fehler bescheren. Durch die Zerlegung schaffen wir eben nur den Anschein, die Komplexität im Griff zu haben (Abb. 3.3 auf Seite 24).

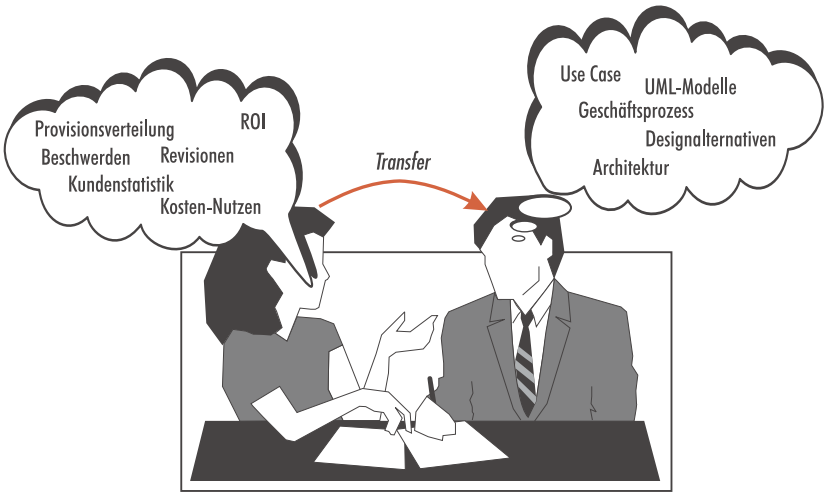

Abbildung 1.3: Bei der Kommunikation zwischen Fachabteilung und Entwicklung findet ein Transfer aus einer Fachsprache in eine andere statt.

1.5 Erstes Fazit

Es bleibt uns also nichts anderes übrig, als zu akzeptieren, dass Fehler immer wieder gemacht werden, ja geradezu gemacht werden müssen! Oder anders formuliert: Wenn wir keine Fehler machen, treten wir auf der Stelle und kommen inhaltlich nicht voran.

Akzeptieren wir Fehler als notwendig in unserem Projekt-Lern-Prozess, so können wir sie bewusst dazu nutzen:

- Über Fehler können wir die kritischen Bereiche identifizieren, um sie dann genauer zu betrachten.
- Unser Entwicklungsprozess sollte sich aktiv und zu jeder Zeit den Fehlern stellen, sie suchen, finden und beheben.

Leider stellt sich uns das Problem der Fehler und ihrer Entdeckung noch wesentlich differenzierter dar. Fehler weisen Strukturen auf. Nicht jeden Fehler können wir zu jedem Zeitpunkt finden. Es muss daher differenzierte Testarten geben. Die grundsätzliche Einstellung ist dabei aber der Schlüssel zum Erfolg!

Wir akzeptieren Fehler und nutzen sie. Fehler sind nicht notwendigerweise ein Zeichen von Schwäche, sondern systemimmanent. Wenn wir das leugnen, werden wir scheitern. Die Einstellung gegenüber Fehlern, die *Fehlerkultur*, ist so wichtig, dass sie später noch differenzierter betrachtet wird.

2 Programmiersprachen sind fehleranfällig

Unser wesentliches Werkzeug ist die Programmiersprache selbst. Und wie jedes Tool, das wir sonst noch so einsetzen, hat sie Stärken und Schwächen. Schauen wir uns zwei konkrete Beispiele näher an.

2.1 Die Venussonde Mariner 1

Im Jahr 1962 wurde die Trägerrakete der Mariner 1-Venussonde 290 Sekunden nach dem Start kontrolliert zerstört, da sie von der vorgesehenen Flugbahn abwich. Der Schaden belief sich auf ca. 18,5 Mio. $. Was war los? Die Steuerungssoftware der Atlas-Agena B-Trägerrakete ist in FORTRAN programmiert worden. Die entscheidende Schleife sieht so aus [37]:

```
IF (TVAL .LT. 0.2E-2) GOTO 40
DO 40 M = 1, 3
W0 = (M-1)*0.5
X = H*1.74533E-2*W0
DO 20 N0 = 1, 8
EPS = 5.0*10.0**(N0-7)
CALL BESJ(X, 0, B0, EPS, IER)
IF (IER .EQ. 0) GOTO 10
20 CONTINUE
DO 5 K = 1. 3
T(K) = W0
Z = 1.0/(X**2)*B1**2+3.0977E-4*B0**2
D(K) = 3.076E-2*2.0*(1.0/X*B0*B1+3.0977E-4**(B0**2-X*B0*B1))/Z
E(K) = H**2*93.2943*W0/SIN(W0)*Z
H = D(K)-E(K)
5 CONTINUE
10 CONTINUE
Y = H/W0-1
40 CONTINUE
```

Codebeispiel 2.1: Ausschnitt der FORTRAN-Steuerungssoftware der Atlas-Agena B-Trägerrakete aus dem Jahr 1962.

FORTRAN-Cracks können ja mal den Fehler suchen. Während meines Physikstudiums habe ich diese Sprache gut kennen gelernt und auch später noch in FORTRAN programmiert. Ich darf Sie daher bitte kurz durch die relevante Stelle des Codes führen? Es ist die kleine Schleifenanweisung in der Mitte:

```
DO 5 K = 1. 3
```

Richtig wäre gewesen:

```
DO 5 K = 1, 3
```

Ja wirklich: Ein Punkt anstatt eines Kommas macht den Unterschied. Und der Compiler bemerkt es auch nicht! In FORTRAN gibt es eine Reihe von Default-Regeln, die den Entwicklern das Leben vereinfachen sollen, aber leider seltsame Nebenwirkungen haben. So werden unter gewissen Umständen Blanks ignoriert. In diesem Fall denkt der Compiler, es soll sich um eine Wertzuweisung handeln, da nach dem Gleichheitszeichen wohl eine Zahl steht und dabei das Blank ignoriert wird. Davor muss dann eine Variable stehen. Auch hier werden die Blanks ignoriert. Da diese Variable nicht vorher deklariert wurde, gelten die Defaults, in diesem Fall wird eine FLOAT-Variable mit dem Namen DO5K angelegt. Zusammengefasst sieht der Compiler die Zeile

```
DO5K = 1.3
```

und führt keine Schleife von 1 bis 3 aus, wie eigentlich gewünscht. Kleine Ursache, große Wirkung!

Sie werden einwenden, dass sich dieser Fehler vor über 40 Jahren zugetragen hat, in der ältesten Hochsprache. Heute sind wir doch viel weiter. Sind wir das wirklich? Auch in den modernen, typisierten Sprachen ist ein solcher Fehler möglich, z. B. in C++:

```
while (x > 0,1) {...}
```

Dieses Statement führt zu einer Endlosschleife, weil auch hier ein Punkt mit einem Komma verwechselt wurde. Ein C++-Compiler sieht zwei Terme in der Bedingung, $x > 0$ und 1, wobei Letzteres immer TRUE ist.

Erst in Java würde die `while`-Bedingung angemeckert werden, da dort keine implizite Wandlung von 1 in TRUE erfolgt. Sind also typsichere Sprachen wie Java oder Ada die Lösung?

2.2 Der Jungfernflug der Ariane 5

Speziell für komplexe, sicherheitsrelevante Software wurde im Auftrag des US-Verteidigungsministeriums die Sprache *Ada* entwickelt. Hilft uns

das weiter? Ich befürchte nicht, denn z. B. die Steuersoftware der Ariane-Raketen ist in Ada geschrieben.

Die Gesamtentwicklungskosten in zehn Jahren der Ariane 5 im Jahr 1996 belief sich auf ca. 5,5 Mrd. €. Werfen wir einen Blick in die Programmierung des Trägheitsnavigationssystems [37]. Ich habe dabei nur die wesentlichen Teile herausgepickt und die für unsere Betrachtungen irrelevanten Zwischenteile, die durch ... angedeutet sind, übersprungen.

```
declare
  vertical_veloc_sensor: float;
  horizontal_veloc_sensor: float;
  vertical_veloc_bias: integer;
  horizontal_veloc_bias: integer;
...
begin
  declare pragma suppress(numeric_error, horizontal_veloc_bias);
  begin
    sensor_get(vertical_veloc_sensor);
    sensor_get(horizontal_veloc_sensor);
    vertical_veloc_bias := integer(vertical_veloc_sensor);
    horizontal_veloc_bias := integer(horizontal_veloc_sensor);
...
    exception
      when numeric_error => calculate_vertical_veloc();
      when others => use_irs1();
  end;
end irs2;
```

Codebeispiel 2.2: Ausschnitt der ADA-Steuerungssoftware der Ariane 5-Trägerrakete, 1996

Was ist hier passiert? Standardmäßig werden die Gültigkeitsbereiche zur Laufzeit geprüft, was jedoch unterdrückt werden kann. Genau dies erfolgt in der Zeile

```
  declare pragma suppress(numeric_error, horizontal_veloc_bias);
```

Von einem Sensor für die horizontale Geschwindigkeitsermittlung werden interne Einheiten an die Steuerungssoftware gegeben. Diese werden mit der Zeile

```
    horizontal_veloc_bias := integer(horizontal_veloc_sensor);
```

einer Ganzzahl-Variablen zugewiesen. Genau hier erfolgt 30 Sekunden nach dem Abheben ein Überlauf, ein Integer-Overflow, der nicht abgefangen wurde, da die Laufzeitprüfung etwas weiter oben ausgeschaltet wurde.

Wie nur konnte es dazu kommen? Nun, die betroffene Software lief seit Jahren problemlos und unverändert in der schubschwächeren Vorgängerversion Ariane 4. Auf einen intensiven Test inkl. einer Simulation wurde daher verzichtet, insbesondere weil der Test mit ca. einer halben Mio. € auch recht teuer war.

37 Sekunden nach dem Zünden der Rakete bzw. 30 Sekunden nach dem Abheben erreichte die Ariane 5 in 3700 m Flughöhe eine Horizontalgeschwindigkeit von 32768,0 internen Einheiten des Sensors. Dieser Wert lag etwa fünfmal höher als beim Vorgängermodell Ariane 4. Wie gesehen führte die Umwandlung in eine ganze Zahl zu einem Überlauf, der nicht abgefangen wurde.

Der redundant ausgelegte Ersatzrechner hatte das gleiche Problem bereits 72 ms vorher und schaltete sich gemäß seiner Spezifikation sofort ab. Die Diagnosedaten, die zum Hauptrechner geschickt wurden, interpretierte dieser als Flugbahndaten, die zu unsinnigen Steuerbefehlen für die Feststofftriebwerke wie für das Haupttriebwerk führten. So sollte die berechnete Flugabweichung von über 20° korrigiert werden. Daraufhin drohte die Rakete auseinander zu brechen und sprengte sich 39 Sekunden nach dem Zünden der Triebwerke selbst.

Das sollten wir uns etwas genauer anschauen: Der problematische Programmteil wird nur für die Startvorbereitungen und den Start eingesetzt. Es ist aus Sicherheitsgründen während der ersten 50 Sekunden aktiv, bis die Bodenstation die vollständige Kontrolle übernommen hat.

Im Code wird nur bei drei der sieben Variablen auf einen Überlauf geprüft. Für die anderen vier Variablen wurde diese Prüfung ausgeschaltet, da Beweise existieren, dass die Werte bei der Ariane 4 stets klein genug bleiben würden. Die Beweise gelten jedoch nicht für die wesentlich stärkere Ariane 5 und wurden auch nicht für sie nachgezogen oder geprüft. Die Ursache war also die Wiederverwendung scheinbar unproblematischen Codes!

Wieso lag ein so fester Glauben an die Software der Ariane 4 bzw. 5 vor? Es wurde beim Programmdesign davon ausgegangen, dass nur Hardwarefehler auftreten können! Das Risikomanagement hat Softwarefehler gar nicht in Betracht gezogen. Deshalb wurden auch die Ersatzrechner mit identischer Software ausgestattet. Daher wurde auch in der Systemspezifikation festgelegt, dass sich im Fehlerfall eines Rechners dieser abschalten soll und der andere einspringt. Ein Restart des Systems dauert viel zu lange, da die Flughöhenbestimmung recht aufwendig ist.

2.3 Zweites Fazit

Beide Beispiele habe ich nicht aus Schadenfreude ausgewählt, sondern weil wir die grundsätzliche Problematik daran gut erkennen können. Wie bereits

eingangs bemerkt, gehe ich davon aus, dass in der jeweiligen Entwicklung eher überdurchschnittlich gute Programmierer und Projektleiter zu finden waren. Wenn schon denen solche Fehler unterlaufen, wie sieht es dann bei uns »Normalos« aus?

Der Glaube an die Fähigkeiten des Compilers bzw. an die Sicherheit wiederverwendeten Codes verstellt für unsere Risikobetrachtungen den Blick auf die weiterhin problematischen Teile. Gerade typisierte Sprachen geben uns so eine trügerische Sicherheit [28].

Die strenge Typisierung ist ein überschätzter Sicherheitsmechanismus. Sprachen wie Smalltalk oder Python bieten gar nicht die Möglichkeiten dazu. Es sind nicht-typisierte Sprachen, und die Smalltalk- oder Python-Programmierer vermissen die Typisierung auch nicht! Generell können wir uns fragen, wie in Smalltalk oder Python erfolgreich Software entwickelt werden kann, wenn die Typprüfungen doch so wichtig sein sollen?

Laufzeitfehler erfolgen eben trotz der statischen, strengen Typüberprüfungen. Ein kompilierfähiges Programm in einer streng typisierten Sprache wie z. B. Java hat eben nur die rudimentären, syntaktischen Tests bestanden. Diese sind notwendig, aber bei weitem nicht hinreichend!

Deutlich wird dies am Beispiel einer Interface-Realisierung aus Abb. 2.1 auf Seite 14. Das Interface `Sortierbar` deklariert die zu implementierenden Methoden für alle sortierbaren Klassen. Eine Client-Klasse kann nun über eine Menge (Collection) sortierbarer Objekte gehen und z. B. deren Maximum bestimmen. Die implizite Annahme, die dazu getroffen wird, lautet, dass die Menge nur aus Objekten derselben Klasse besteht.

Warum ist das so? Die von den konkreten, sortierbaren Klassen implementierten Vergleichsmethoden wie `istGroesser()` beruhen auf klassenspezifischer Logik. Eine Reservierung wird eben nach anderen Kriterien sortiert werden als eine Person oder ein Buch. Da bei einem Vergleich stets mindestens zwei Objekte betrachtet werden müssen, erfolgt zur Laufzeit ein Cast vom Basistyp `Sortierbar` herunter in die konkrete Klasse, also z. B. `Reservierung`. Damit das auch funktioniert, darf die betrachtete Menge nur homogen aus Reservierungsobjekten bestehen.

O.k., in Java können wir durch explizite Typabfragen auf die Klasse eines Objekts eine zusätzliche Sicherheit programmieren. Es geht in diesem Beispiel auch nur um die Illustration des grundsätzlichen Problems. Außerdem ist dies nicht in allen Sprachen möglich. In nicht streng typisierten Sprachen wie Smalltalk oder Python ist dies weder möglich noch gewünscht!

Unabhängig von der Sprache müssen alle Tests erfolgreich durchlaufen werden, die definiert worden sind, um den korrekten Ablauf sicherzustellen. Und hier spielen Smalltalk oder Python ihre Stärken aus: Der Code kann schneller geschrieben werden, und somit können die Tests früher beginnen!

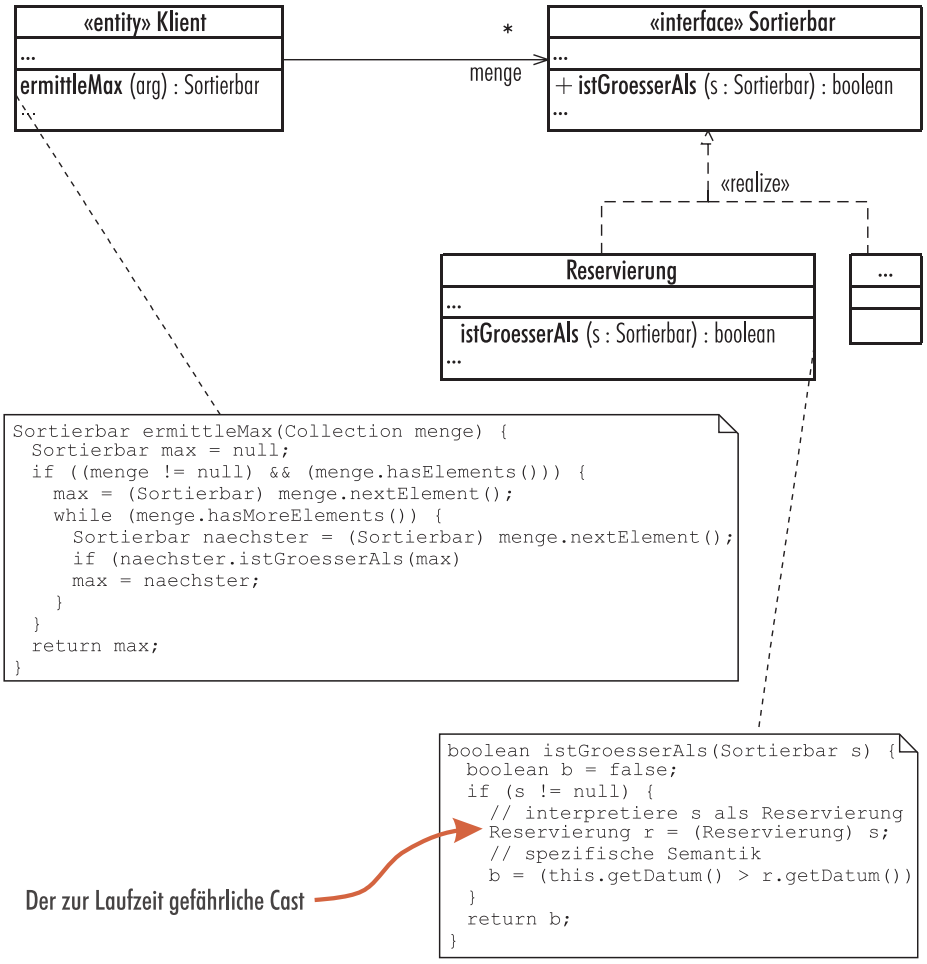

Abbildung 2.1: Das Interface-Pattern wird gerne und häufig zum Entkoppeln von Abhängigkeiten angewendet. Bei der Betrachtung von Mengen von Objekten kann es zu Problemen kommen.

8 Lösungen für fortgeschrittene Probleme

8.1 Zustandsraumbasiertes Testen

Ein beliebter Tummelplatz für Fehler sind Zustandsübergänge. Ein Zustand ist eine fachlich motivierte Abstraktion einer Menge möglicher Werte eines Modellelements wie z. B. einer Klasse, wobei deren Werte in den Attributen abgelegt werden. Ein Zustandsübergang, also eine Transition von einem Zustand in einen anderen, wird durch ein Ereignis ausgelöst.

Zustandsraumbasiertes Testen können wir unabhängig von prozeduraler oder objektorientierter Programmierung immer dann einsetzen, wenn wir ein zustandsbasiertes Regelwerk implementieren. Ein solches System bezeichnen wir auch als *Zustandsautomaten*. Er besteht aus einer endlichen Anzahl interner Zustände [3]. Wir werden im Zusammenhang mit objektorientierten Testmustern in Abschnitt 9.3.1 ab Seite 112 noch einmal auf dieses Thema zurückkommen.

Zustände können mit der UML modelliert werden. Die entsprechenden Elemente sind Abb. 8.1 zu entnehmen. Im Wesentlichen basiert die Zustandsmodellierung immer noch auf einem Artikel von David Harrel von 1987 [39].

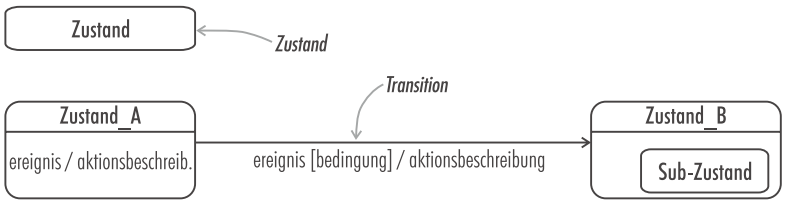

Abbildung 8.1: Die zentralen Elemente der Zustandsmodellierung mit der UML.

Ein Zustand kann Unterzustände haben und Aktionen auslösen, die an bestimmte Ereignisse geknüpft sind. Ein Zustandsübergang (Transition) ist ebenfalls mit einem Ereignis verbunden, welches zusätzlich durch eine Be-

dingung geschützt sein kann. Transitionen können auch Aktionen auslösen. Aktionen sind z. B. das Senden eines Signals oder der Aufruf von Methoden.

Ein Zustand kann also für definierte Ereignisse bestimmte Aktionen auslösen. Drei solcher zustandsinternen Ereignisse sind bereits vordefiniert:

do Während der Zustand aktiv ist, wird die beschriebene Aktion ausgeführt.

entry Beim Eintreten in diesen Zustand wird die beschriebene Aktion ausgeführt.

exit Beim Verlassen des Zustands wird die beschriebene Aktion ausgeführt.

Die Transitionen werden von Ereignissen ausgelöst und können an Bedingungen geknüpft sein. Diese Bedingung wird auch als *Guard* bezeichnet. Eine Transition kann eine Aktion auslösen, z. B. das Senden eines Signals.

Die möglichen Werte eines Modellelements wie z. B. Klasse spannen einen mehrdimensionalen Raum auf, den Zustandsraum (Abb. 8.2). Eine Klasse legt ihre Zustände in ihren Attributen ab, wobei sich ein bestimmter Zustand aus einer Kombination verschiedener Attribute zusammensetzen kann. Im Zustandsraum gibt es erlaubte Übergänge von einem Zustand in den anderen und verbotene Wege, die mögliche Fehler im späteren Programm bilden können, die Schleichpfade. Im Zustandsraumbasierten Test prüfen wir die erlaubten und verbotenen Übergänge. Gerade die möglichen Schleichpfade werden nur zu gerne beim Test vergessen.

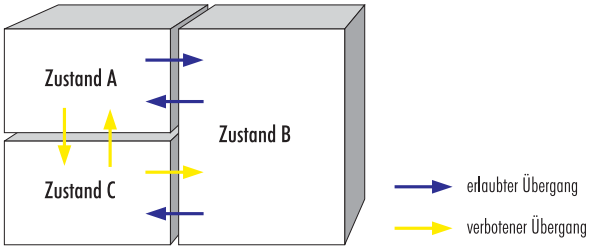

Abbildung 8.2: Schematische Darstellung eines Zustandsraums mit erlaubten und verbotenen Zustandsübergängen. Zwischen Zustand A und B kann hin und her gewechselt werden, aber nur aus B heraus können wir in Zustand C gelangen, der den Endzustand darstellt.

Ein einfaches Beispiel für ein Zustandsmodell finden wir in Abb. 8.3. Dort ist das Modell für einen Kassettenrecorder dargestellt. O.k., Kassettenrecorder sind im Zeitalter der CD-Brenner etwas aus der Mode gekommen, aber das Beispiel ist aussagekräftig und dennoch einfach zu durch-

dringen. Außerdem handelt es sich um einen modernen Kassettenrecorder mit Sensortasten ähnlich einem CD-Player.

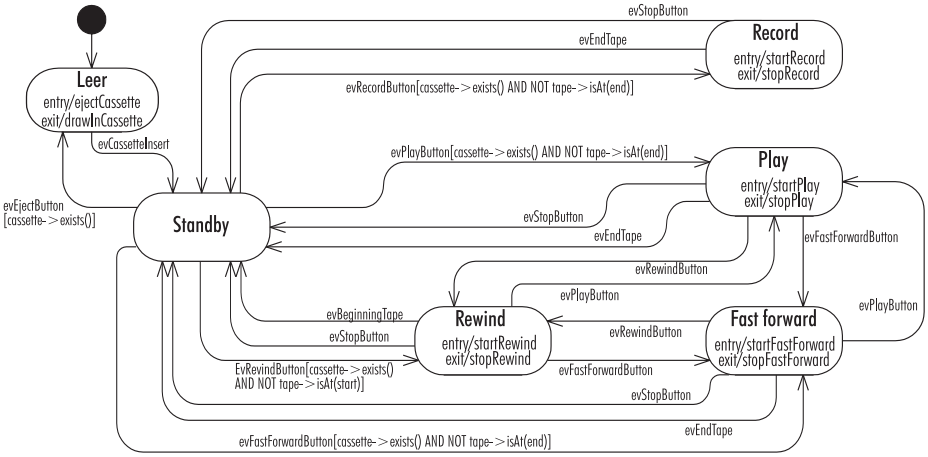

Abbildung 8.3: Zustandsmodell für einen Kasettenrecorder

Nach einem Zustandsmodell können wir nur schwer Tests definieren. Es bietet dafür keine gute Basis. Besser ist es, das Modell in einen Zustandsbaum zu überführen und eine Zustandsübergangstabelle abzuleiten.

Eine Zustandsübergangstabelle erzeugen wir, indem wir alle möglichen Zustände nebeneinander schreiben und an den Kopf jeder darunter liegenden Zeile die möglichen Ereignisse. In die Tabellenelemente können wir jetzt die resultierenden Zustände eintragen. Ein Beispiel für eine vereinfachte Form einer Zustandsübergangstabelle ist Tab. 9.1 auf Seite 115 zu entnehmen. Wie das für unser Kassettenrecorder-Beispiel aussehen kann, ist in Tab. 8.1 zu sehen.

Um sie später besser referenzieren zu können, sind in der Tabelle die Zustandsübergänge durchnummeriert. Übergänge, die an spezielle Bedingungen geknüpft sind, also durch Bedingungsprüfungen geschützt sind, haben wir dabei mit einem *g* für *Guard* markiert. Für die gültigen Übergänge sind die auslösenden Ereignisse angegeben, wobei die Tabelle von der Kopfzeile aus zu den einzelnen auslösenden Ereignissen in der ersten Spalte zu lesen ist. Der resultierende Zustand steht dann in dem Tabellenschnittpunkt. Die Nummern identifizieren die Übergänge eindeutig für den folgenden Zustandsübergangsbaum (Abb. 8.4).

Auf Basis einer Zustandsübergangstabelle kann versucht werden, diese als Tabelle zu kodieren und im Code bei der Behandlung eintreffender Ereignisse gegen diese Tabelle zu prüfen. Die korrekte Implementierung und Anpassungen im weiteren Projektverlauf sollten natürlich getestet werden.

	Leer	Standby	Rewind	Play	Fast Forward	Record
evEject-Button	×	2 (g) Leer	×	×	×	×
evRewind-Button	×	3 (g) Rewind	×	11 Rewind	15 Rewind	×
evPlayButton	×	4 (g) Play	7 Play	×	16 Play	×
evFastForward-Button	×	5 (g) FFwd	8 FFwd	12 FFwd	×	×
evRecord-Button	×	6 (g) Record	×	×	×	×
evStopButton	×	×	9 Standby	13 Standby	17 Standby	19 Standby
evEndTape	×	×	×	14 Standby	18 Standby	20 Standby
evBeginning-Tape	×	×	10 Standby	×	×	×
evInsert-Cassette	1 Standby	×	×	×	×	×

Tabelle 8.1: Die Zustandsübergänge aus dem Zustandsmodell eines Kassettenrecorders aus Abb. 8.3 als Wahrheitstabelle. Durch Bedingungen geschützte Übergänge sind durch (g) gekennzeichnet, ungültige durch ×.

Wir können jetzt beginnend mit dem initialen Zustand *Leer* jeden erlaubten Übergang als Transition zeichnen. So entsteht ein *Zustandsbaum*. Wir beenden eine Transitionsfolge, also einen Ast unseres Baums, wenn wir auf einen Zustand stoßen, den wir vorher bereits schon einmal hatten. Die einzelnen Transitionen werden mit den Nummern aus der Tab. 8.1 benannt. Für unser Kassettenrecorder-Beispiel ist der Zustandsbaum in Abb. 8.4 dargestellt. Ein Zustandsbaum ist kein eigenes UML-Diagramm. Er kann aber mit den Hilfsmitteln der UML beschrieben werden, wie in Abb. 8.4 geschehen. Auf Grundlage der Tabelle und des Baums können wir jetzt das Zustandsmodell vollständig testen. Wir gehen dabei in zwei Schritten vor.

1. Wir testen alle erlaubten Übergänge, indem wir mit unseren Tests jeden Ast unseres Zustandsbaums durchlaufen.
2. Aus der Tabelle entnehmen wir alle verbotenen Übergänge und prüfen, dass diese auch wirklich nicht möglich sind. Ansonsten haben wir einen verbotenen *Schleichpfad* gefunden.

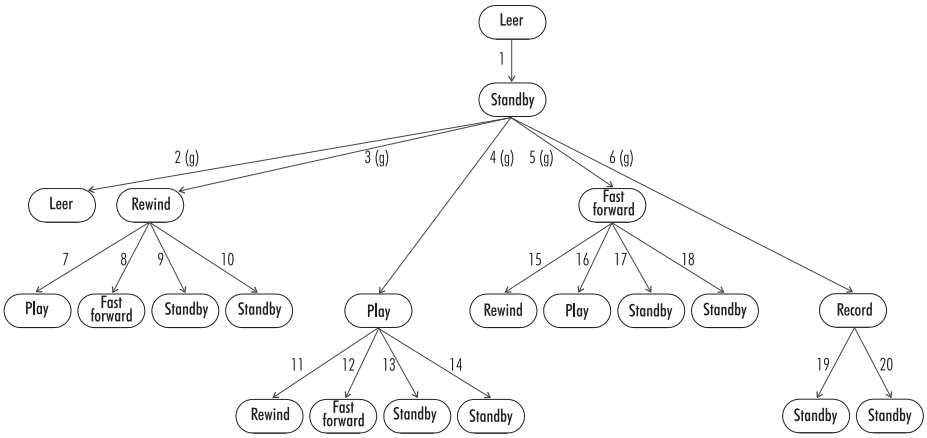

Abbildung 8.4: Aus dem Zustandsmodell aus Abb. 8.3 abgeleiteter Zustandsbaum für einen Kassettenrecorder.

Leider gibt es keine sinnvolle Vereinfachung für diese beiden Testschritte. Wir sollten sie stets vollständig ausführen. Anderenfalls kann uns hier ein schwerer Fehler durch unsere Tests rutschen!

8.2 Rekursion und Nebenläufigkeit

Besondere Probleme für den Test können rekursive Algorithmen oder Nebenläufigkeiten aufwerfen. Bei der Nebenläufigkeit ist das offensichtlich: Parallele Prozesse oder Threads sind komplex und damit per se fehleranfällig. Rekursive Algorithmen sind dagegen mathematisch klar, elegant und eindeutig und von daher eigentlich recht robust. Aber auch hier kann der Teufel im Detail stecken.

8.2.1 Rekursive und iterative Algorithmen

Eine Prozedur, die sich selbst aufruft, heißt *rekursiv*. Der Aufruf erfolgt dabei entweder direkt oder über andere Prozeduren bzw. Methoden indirekt (Abb. 8.5).

Ein rekursiver Algorithmus kann auch nicht-rekursiv programmiert werden. Generell sollten wir uns fragen, ob wir überhaupt eine rekursive Implementierung haben wollen oder nicht lieber den Algorithmus in eine Iteration umformulieren. Als Beispiel, wie die beiden unterschiedlichen Implementierungen aussehen, habe ich einen Klassiker ausgewählt, den Quicksort-Algorithmus nach C. A. R. Hoares [55, 91].