Andreas Spillner • Tilo Linz

20 JAHRE

Open Book »Basiswissen Softwaretest« in der ersten Auflage 2002

# Basiswissen Softwaretest

Aus- und Weiterbildung zum Certified Tester

- Foundation Level
- nach ASQF- und ISEB-Standard

dpunkt.verlag

**Basiswissen Softwaretest**

## Über die Autoren

**Andreas Spillner** ist Professor für Informatik an der Hochschule Bremen, Fachbereich Elektrotechnik und Informatik. Er ist Sprecher der Fachgruppe TAV »Test, Analyse und Verifikation von Software« der Gesellschaft für Informatik e.V. und Mitglied im German Testing Board. Seine Arbeitsschwerpunkte liegen im Bereich Softwaretechnik, Qualitätssicherung, Testen und objektorientierte Softwareentwicklung.

**Tilo Linz** ist Vorstand der imbus AG, einem führenden Dienstleister für Softwaretest. Er ist Vizepräsident des ASQF und Leiter des German Testing Board. Seine Arbeitsschwerpunkte sind Beratungs- und Coaching-Projekte zu den Themen Softwarequalitätsmanagement und Optimierung von Softwareentwicklungs- und Softwaretestprozessen.

Andreas Spillner • Tilo Linz

# Basiswissen Softwaretest

Aus- und Weiterbildung zum Certified Tester

Foundation Level nach ASQF- und ISEB-Standard

dpunkt.verlag

spillner@informatik.hs-bremen.de
tilo.linz@imbus.de

Lektorat: Christa Preisendanz
Copy-Editing: Jörg Gottschalk, Hannover
Herstellung: Verlagsservice Hegele, Dossenheim
Umschlaggestaltung: Helmut Kraus, Düsseldorf
Druck und Bindung: Koninklijke Wöhrmann B.V., Zutphen, Niederlande

Die Deutsche Bibliothek – CIP-Einheitsaufnahme

Spillner, Andreas:
Basiswissen Softwaretest : Aus- und Weiterbildung zum Certified-Tester / Andreas Spillner; Tilo Linz –
1. Aufl. – Heidelberg : dpunkt-Verl., 2003
ISBN 3-89864-178-3

1. Auflage 2003
Copyright © 2003 dpunkt.verlag GmbH
Ringstraße 19 b
69115 Heidelberg

Die vorliegende Publikation ist urheberrechtlich geschützt. Alle Rechte vorbehalten.
Die Verwendung der Texte und Abbildungen, auch auszugsweise, ist ohne die schriftliche Zustimmung des Verlags urheberrechtswidrig und daher strafbar. Dies gilt insbesondere für die Vervielfältigung, Übersetzung oder die Verwendung in elektronischen Systemen.

Alle Informationen in diesem Buch wurden mit größter Sorgfalt kontrolliert.
Weder Autoren noch Verlag können jedoch für Schäden haftbar gemacht werden, die in Zusammenhang mit der Verwendung dieses Buches stehen.

In diesem Buch werden eingetragene Warenzeichen, Handelsnamen und Gebrauchsnamen verwendet. Auch wenn diese nicht als solche gekennzeichnet sind, gelten die entsprechenden Schutzbestimmungen.

# Vorwort

*Motivation für ein weiteres Testbuch*

Warum haben wir ein Buch zum Thema Softwaretest geschrieben, wo es doch eine ganze Reihe von Büchern in diesem Bereich bereits gibt?

Wir beide beschäftigen uns seit vielen Jahren mit Themen und Fragestellungen im Bereich Softwaretest. Der eine mit einer eher wissenschaftlichen Ausrichtung, der andere mit der täglichen Anwendung der Prüfverfahren in der Praxis. Beide sind wir der Auffassung, dass der systematische Einsatz von Prüfverfahren in der Praxis in vielen Firmen verbesserungsfähig ist und es einen großen Mangel an qualifiziertem Personal gibt.

Die ISEB-Initiative, ein dreistufiges Qualifizierungsprogramm im Bereich Softwaretest zu definieren, Kursanbieter in diesem Bereich zu akkreditieren und den Teilnehmern die Möglichkeit zu bieten, sich einer unabhängigen Prüfinstanz zu stellen und ein Zertifikat zu erhalten, ist von uns von Beginn an mit starkem Interesse verfolgt worden. Die Initiative ist vom ASQF und der Fachgruppe TAV der Gesellschaft für Informatik aufgegriffen worden. Eine ausführliche Beschreibung der Zusammenhänge findet sich in der Einleitung des Buches.

Die Lehrpläne für das Qualifizierungsprogramm sind in Gremien von mehreren europäischen Fachexperten erarbeitet worden und bündeln das derzeitige Wissen im Bereich Softwaretest. Die Inhalte sind in vielen Büchern zu finden, aber eben nicht in einem einzigen Buch. Die Lehrpläne listen die einzelnen Punkte auf, die zu behandeln sind, enthalten aber weder detaillierte Beschreibungen der einzelnen Verfahren und Vorgehensweisen beim Testen von Software noch erläuternde Beispiele. Die Idee ein »passendes« Buch zu schreiben, fanden wir sehr nahe liegend.

Grundlage für unser Buch ist der Lehrplan »Grundlagen des Testens« ASQF Certified Tester, Foundation Level [URL: GTB CTF 01]. Die ersten Erfahrungen mit dem Lehrplan zum Foundation Level führten zu dem Wunsch einer leichten Überarbeitung, die derzeit von einem Gremium des ISTQB (International Software Testing Qualification Board) in Angriff genommen wird. Wir beide haben durch die Arbeit am Buch kleinere Schwachstellen im Lehrplan erkannt und beteiligen uns aktiv an dessen Überarbeitung. Die nach Beendigung der Arbeit beschlossenen Änderungen, die sich auf den Inhalt des Buches auswirken, werden wir auf der WWW-Seite zum Buch [URL: dPunkt] aktuell dokumentieren.

Der Titel unseres Buches »Basiswissen Softwaretest« sagt eigentlich schon alles. Wir sind davon überzeugt, dass jeder im Bereich der Softwareentwicklung Tätige – sei es beispielsweise in der Programmierung, in der Qualitätssicherung, in der Projektleitung oder im Management – ein Grundwissen im Softwaretest besitzen sollte, um seine tägliche Arbeit besser verrichten und den oft vernachlässigten Bereich Testen besser einschätzen oder überhaupt verstehen zu können. Die Aussage trifft selbstverständlich auch für die zukünftig Tätigen, die Studierenden und Auszubildenden im IT-Bereich, zu. Auch so mancher Lehrende wird sein Wissen mithilfe des Buches vervollständigen können.

Wir haben uns bemüht, ein leicht verständliches und kompaktes Buch zu schreiben, um dem großen Kreis an möglichen Lesern gerecht zu werden. Vorwissen im Bereich Testen oder Qualitätssicherung ist nicht erforderlich, Grundkenntnisse der Softwareentwicklung sind sicherlich hilfreich. Unser Buch hat zwar den Lehrplan zum »Foundation Level Certified Tester« als Grundlage, es besteht aber keine zwanghafte Kopplung, auch das entsprechende Zertifikat zu erwerben.

*Drei Geleitworte*

Warum gibt es drei Geleitworte in Englisch und Deutsch in unserem Buch?

Durch unsere langjährigen Aktivitäten im Bereich Softwaretest haben wir zahlreiche internationale Kontakte aufbauen können. Als unser Buchprojekt konkrete Formen annahm, haben wir drei Personen um Geleitworte gebeten. Da alle drei viel gefragte und beschäftigte Persönlichkeiten sind, haben wir eher mit zurückhaltenden Reaktionen gerechnet. Um so erfreuter waren wir, dass wir tatsächlich drei Geleitworte innerhalb der Termine erhalten haben. Unsere »Risko«-Planung war somit völlig unnötig. Da jedes Geleitwort einen anderen Aspekt vertieft, haben wir uns entschlossen, alle drei aufzunehmen.

*Prof. David Parnas Ph.d, Dr. h.c., Dr. h.c., FRSC, P.Eng.*

David Parnas, der Begründer der Modularisierung und des Geheimnisprinzips (*information hiding*), engagiert sich seit langem in Fragen der Ausbildung im Bereich Informatik. Für ihn muss die Informatik-Ausbildung eine Professionalität erreichen wie bei den Medizinern oder in den Rechts- und Ingenieurwissenschaften. Die allgemein anerkannte Festlegung von Lehrinhalten und der Nachweis des erworbenen Fachwissens, beispielsweise im Bereich Softwaretest, sieht er als einen Schritt in die richtige Richtung an. Teile des Buches sind während des Forschungsaufenthalts von Andreas Spillner auf Einladung von David Parnas an der McMaster University in Hamilton, Ontario in Kanada entstanden. Ausführliche Informationen zu David Parnas sind zu finden unter [URL: Parnas].

*Martin Pol*

Martin Pol gehört wohl zu den bekanntesten Persönlichkeiten im Testbereich in Europa, wenn nicht gar weltweit. Er ist Mitautor von

T-Map (Test Management Approach) und TPI (Test Process Improvement). 1998 erhielt er den »European Testing Excellence Award«. In seinem Geleitwort hebt er die Bedeutung des Testens im Softwareentwicklungsprozess hervor und wie wichtig Basiswissen im Softwaretest für alle im IT-Bereich arbeitenden Personen ist. Ausführliche Informationen zu Martin Pol sind abrufbar unter [URL: Pol].

*Dorothy Graham A.B., M.Sc.*

Dorothy Graham, seit vielen Jahren Beraterin und Trainerin im Softwarequalitäts- und Testbereich, hat das ISEB Software Testing Board mitgegründet und gehört ihm seitdem an. Ohne das große Engagement von Dorothy Graham wäre die ISEB-Initiative wohl nicht so weit vorangeschritten und die erfolgte Internationalisierung noch in weiter Ferne. Sie ist Mitautorin von Büchern zu den Themen »Software-Inspektionen« und »Softwaretest-Automatisierung«. Der »European Testing Excellence Award« wurde ihr 1999 zuerkannt. In ihrem Geleitwort macht Dorothy Graham die Vorteile einer Akkreditierung und Zertifizierung durch eine unabhängige Instanz deutlich. Ausführliche Informationen zu ihrer Person sind unter [URL: Graham] zu finden.

Wir möchten Dorothy Graham, Martin Pol und David Parnas für ihre Unterstützung und die geopferte Zeit recht herzlich danken.

*Danksagung*

Wie es sich für ein Projekt mit einem Qualitätsanspruch gehört, haben wir unsere Arbeit externen Reviewern vorgelegt. Wir haben viele Anregungen und Hinweise zur Verbesserung des Textes und der Beispiele erhalten. Wir möchten uns für die vielen geopferten Stunden Freizeit ganz herzlich bedanken bei Uwe Hehn, Ruth Keys, Karin Vosseberg und Mario Winter. Darüber hinaus gilt unser Dank den Kolleginnen und Kollegen der imbus AG, die ebenfalls durch zahlreiche Anmerkungen und konstruktive Diskussionen zum Gelingen des Buches beigetragen haben. Besonders erwähnen möchten wir Matthias Daigl und Thomas Roßner.

Bedanken möchten wir uns auch bei den Mitarbeiterinnen und Mitarbeitern des dpunkt.verlags. Zu Beginn unseres Projekts hatten wir uns eine Deadline gesetzt, was bei jedem Projekt erfolgen sollte. Auch Dank der großen Unterstützung des Verlags ist es gelungen, den von uns geplanten Fertigstellungstermin zu halten.

Wir wünschen alle Lesern des Buches erfolgreiche Stunden in dem Sinne, dass Testen nicht mehr als notwendiges Übel der Softwareentwicklung am Projektende betrachtet wird, sondern als herausfordernde, kreative Tätigkeit, die projektbegleitend erfolgt und neben der Aufdeckung von Fehlern auch zum Nachweis der Qualität der erstellten oder geänderten Software dient. Testen kann und soll auch Spaß machen.

Bremen, Möhrendorf, September 2002
*Andreas Spillner und Tilo Linz*

## Geleitwort von David Parnas*

One of the biggest problems in the computer software field is the ability of inadequately qualified people to enter the profession and practice software development without limits. Software has become critical to our society; it is embedded in many devices that we count on, devices such as telephones and banking machines. Nonetheless, many of the people who write that software have not been educated for the job and have never demonstrated their qualifications to any objective professional body. There is no other field in the world where people without approved education can decide to identify themselves as »Engineers« and produce products that are essential to the safety and well-being of the public.

Software is well known for low reliability and lack of trustworthiness. In part this is attributable to the difficulty of dealing with the complexity of today's software systems, but the inadequate knowledge, skills, and professionalism of many of the practitioners also contributes to this problem. Moreover, we can thank the inadequately qualified people who produced today's software for the unreliability and complexity of the products that serve as support software for new products.

Our educational institutions have failed the public in this field. They have not recognized that those who study Computer Science require a professional education, one similar in style to the education provided to those who study medicine, law, or engineering. In those fields, the curriculum is designed around a set of professional requirements. Students are told what they must learn, rather than allowed to learn what they feel like learning. In the software field, universities have allowed the contents of courses to depend on the whim of the instructor, and the choice of courses to be largely up to the student. As a result, when an employer or client meets a graduate of a Computer Science programme, only experienced software developers are able to judge whether or not a graduate has the knowledge and skills needed for the job. Often, we cannot even find a graduate who has the appropriate body of knowledge and experience.

This problem is quite clear in the area of software testing. Many new employees are assigned testing duties without any knowledge how to design tests, how to evaluate test results, and how to draw valid conclusions can be drawn from the test results. Fortunately, there is now a useful international initiative to establish national Testing Boards,

* Eine deutschsprachige Übersetzung bzw. die engl. Originale sind auf der Webseite des Buches [URL: dPunkt] zu finden.

which will approve courses of instruction and issue certificates to those who are able to demonstrate their understanding of basic terms and procedures in testing.

Unfortunately, this is still a shortage of appropriate study material for people who would like to become better software testers and pass the national tests. This book, »Basiswissen Softwaretest« by Spillner and Linz, fills this gap by providing a well organized and complete view of what is known about how to test software. It provides an essential component of the international effort to establish standards for software professionals and then help people to become fully qualified software developers.

*David Lorge Parnas*,
Ph.d, Dr.h.c., Dr.h.c., FRSC, P.Eng.
McMaster University, Hamilton, Ontario, Canada
June 2002

## Geleitwort von Martin Pol

Für viele Organisationen spielt die Qualität von Softwaresystemen eine immer größere Rolle. Es werden zunehmend Maßnahmen ergriffen, um eine höhere Qualität zu erreichen. Trotz ermutigender Resultate mit verschiedenen Ansätzen zur Qualitätssicherung ist die IT-Branche weit davon entfernt, fehlerfreie Software entwickeln zu können. Ein solches Ziel wird leider noch für geraume Zeit eine Utopie bleiben. Die Entwicklung von Softwaresystemen ist nach wie vor ein schwieriges »Handwerk« und auf absehbare Zeit kaum ohne Fehler zu meistern. Die Ursachen für Fehler sind vielfältig und schwer vorhersehbar. Man wird weiterhin nicht umhin kommen, viel Energie darauf zu verwenden, die Fehler ausfindig zu machen. Das Testen wird ein wichtiger Bestandteil der Entwicklung und Wartung bleiben und oft mehr als 30–40 % des Gesamtbudgets aufzehren.

Das Testen ist nicht mehr nur eine Phase, die nach der Implementierung kommt, sondern ist ebenso ernst zu nehmen wie andere Aktivitäten der Entwicklung von Systemen. Testen ist zu einer eigenständigen Aufgabe und Tätigkeit geworden. Die Informatik hat sukzessive die Grundideen des Testens akzeptiert: Testen von Software ist ein Prozess, bestehend aus Planung und Vorbereitung einerseits und Messen und Prüfen andererseits. Testen dient dazu, die Charakteristika eines Systems festzustellen und die Unterschiede zwischen dem Ist- und dem Sollverhalten aufzuzeigen. Gute Qualität kann als Erfüllung der Anforderungen gesehen werden. Somit ist das Ergebnis des Testens, die vorhandene Qualität aufzuzeigen und Verbesserungen zu ermöglichen. Es gibt einen Ein-

blick darin, welche Risiken bestehen, wenn eine geringere Qualität akzeptiert wird. Dies ist ebenfalls ein wesentliches Ziel des Testens.

Da Time-to-Market, Wettbewerb, Globalisierung und Quality of Service, einschließlich der Qualität von Softwaresystemen, für viele Firmen zu einem wichtigen Überlebensfaktor geworden sind, steigt der Bedarf für einen angemessenen Testprozess immer weiter an. Sowohl die immer größer werdende Bedeutung von Software in unserer Gesellschaft als auch das Budget, das für das Testen ausgegeben wird, bestätigen den Bedarf an einem gut strukturierten und verlässlichen Testprozess innerhalb der Softwareentwicklung. Hierbei sind ein strukturiertes Vorgehen, eine angemessene Organisationsstruktur und die entsprechende Infrastruktur notwendig.

Dieses Buch schafft die Grundlage für einen gut strukturierten Testprozess. Es beschreibt die Grundsätze des Testens, alle Aktivitäten zum Testen innerhalb des Entwicklungsprozesses, Techniken des Testens, Testmanagement und Testwerkzeuge. Daher ist das Buch für jeden von Interesse, der im IT-Bereich arbeitet: Entwickler, Tester, Anwender, Projekt- und Abteilungsleiter etc. Darüber hinaus ist es ein idealer Begleiter, um sich auf das ISEB-Zertifizierungsprogramm vorzubereiten. Egal für welchen Einsatz – ich bin mir sicher, dass dieses umfassende Buch von sehr großem Nutzen sein wird.

*Martin Pol*
Polteq IT Services B.V.
Amersfoort, The Netherlands
June 2002

## Geleitwort von Dorothy Graham

Was macht ein gutes Zertifizierungsprogramm für Fachleute aus? Ein erfolgreiches Programm muss vor allem drei Hauptkriterien erfüllen:

- Es muss ein Grundkonsens über das Thema vorhanden sein
- Es muss denen, die das Zertifikat erhalten (und ihren Arbeitgebern), einen Mehrwert bieten
- Es muss allen Beteiligten gegenüber fair sein und darf niemanden bevorzugen

Im vorliegenden Buch geht es um ein Programm, das diese Kriterien erfüllt. Beim Thema Softwaretesten ist ein ausreichender Konsens vorhanden, der als Basis für ein Zertifikat dienen kann. Das neue deutsche ASQF-Programm und das erfolgreiche ISEB-Programm aus Großbritannien werden sich bald international durchsetzen. Der Hauptgrund

für den Erfolg des britischen Programms ist meiner Meinung nach die Unabhängigkeit des akkreditierenden und prüfenden Gremiums. Ich glaube auch, dass diese Philosophie den Erfolg des internationalen Programms für zertifizierte Tester sicherstellen wird, bei dem Deutschland momentan eine Vorreiterrolle spielt.

Bei dieser Unabhängigkeit spielen zwei Aspekte eine Rolle: zum einen die Akkreditierung von Ausbildungseinrichtungen und zum anderen die Prüfung der einzelnen Kandidaten.

Worin liegt der Unterschied zwischen Zertifizierung und Akkreditierung? Bei der Zertifizierung geht es um den einzelnen Teilnehmer, während die Akkreditierung Ausbildungseinrichtungen betrifft. Der Teilnehmer erhält ein Zertifikat, wenn er sein Wissen unter Beweis stellt. Das Wissen wird durch gewonnene Arbeitserfahrung und durch Fortbildungsveranstaltungen erworben. Wenn ein Grundkonsens über das Thema vorhanden ist (und damit eine Art »Lehrplan«) kann ein unabhängiges Gremium Kurse von Ausbildungseinrichtungen prüfen – dabei handelt es sich um die Akkreditierung. Die Akkreditierung bescheinigt, dass die Ausbildung einen bestimmten Standard im Hinblick auf Inhalt und Organisation erfüllt. Wenn sich Ausbildungseinrichtungen selbst akkreditieren, ist dies nur wenig aussagekräftig; die Akkreditierung durch ein unabhängiges Gremium ist sehr viel glaubwürdiger.

Um ein Zertifikat zu erhalten, muss der Kandidat eine Prüfung bestehen, die auf einem anerkannten Lehrplan beruht. Wenn die Prüfung durch ein unabhängiges Gremium gestellt, überwacht und benotet wird, bestehen gleiche Chancen für alle Kandidaten (und Ausbildungseinrichtungen). Durch die Unabhängigkeit des überwachenden Gremiums steigt der Wert des vergebenen Zertifikats.

Es gibt Programme, bei denen sich die Ausbildungseinrichtungen selbst akkreditieren und ihre eigenen Prüfungen stellen. Obwohl diese Programme durchaus einen gewissen Wert haben, bieten unabhängige Programme mehr. Diejenigen, die Zertifizierungsprogramme generell ablehnen, betonen, dass es bei allen Programmen Probleme geben kann. Dies ist durchaus richtig – das perfekte Programm gibt es nicht. Dennoch bin ich der Überzeugung, dass unabhängige Programme wie ISEB, ASQF oder der geplante internationale Standard für die Teilnehmer und ihre Arbeitgeber von erheblichem Wert sein werden.

Woher wissen wir, dass ein unabhängiges Programm funktionieren wird? Weil ISEB bereits in vielen Ländern funktioniert. ISEB steht für Information Systems Examination Board. Das ISEB wurde ursprünglich als unabhängiges Gremium für Qualifizierungen im Bereich Systemanalyse und -design gegründet. Es gehört nun zu der British Computer Society und bietet Qualifikationen in zehn Fachgebieten an,

darunter Projektmanagement, IT-Service-Management, Geschäftssystementwicklung, Datenschutz und -sicherheit – und seit 1998 auch Softwaretesten. An der Spitze jedes Fachgebiets bilden Vertreter aus Industrie und Hochschule einen Ausschuss. Arbeitsgruppen kümmern sich um Einzelfragen. So gibt es im Bereich Softwaretesten eine Arbeitsgruppe für die Akkreditierung und eine für die Prüfung. ISEB-Mitarbeiter sorgen für die Verwaltung der Ausschüsse und der Arbeitsgruppen, bei den Prüfungen stellen sie das Aufsichtspersonal, verteilen die von der Prüfungs-AG erarbeiteten Prüfungsfragen und werten die Antworten der Kandidaten aus.

Der Ausschuss Softwaretesten des ISEB wurde 1997 gegründet. Der Lehrplan für den Foundation Level wurde von Ausschussmitgliedern aufgestellt und der erste Kurs im Oktober 1998 abgehalten. Seither läuft das Programm mit großem Erfolg und übertrifft alle Erwartungen. Es ist inzwischen – nach IT-Service-Management – das erfolgreichste Programm des ISEB. Bis Mitte 2002 waren bereits 23 Ausbildungseinrichtungen für den Kurs Softwaretesten (Foundation Level) akkreditiert, darunter Organisationen aus Deutschland, den Niederlanden, Schweden, Irland und Australien sowie Großbritannien. Über 8000 Kandidaten haben in weniger als vier Jahren an der Prüfung zum Softwaretester teilgenommen. Das Programm hat den Status des Softwaretestens und die Anerkennung, die man Softwaretestern entgegenbringt, deutlich gehoben. Auf dem britischen Arbeitsmarkt wird man inzwischen ohne das Zertifikat in der Regel gar nicht erst zu einem Vorstellungsgespräch als Softwaretester eingeladen.

Wie sieht die zukünftige Entwicklung aus? Das International Software Testing Qualification Board (ISTQB) wurde 2002 gegründet, so dass Akkreditierung und Prüfung auf Foundation Level bald in allen angeschlossenen Ländern durchweg von unabhängigen Gremien vorgenommen werden. Softwaretester auf der ganzen Welt werden sich besser verständigen können und anerkannter sein. Dieses Buch wird für deutschsprachige Länder einen wichtigen Beitrag dazu leisten. Das internationale Zertifikat wird weltweit Grundkenntnisse im Bereich Softwaretesten sichern. Dies ist eine spannende Zeit für das Softwaretesten!

*Dorothy Graham*
Macclesfield, UK
June 2002

# Inhalt


**1 Einleitung 1**

**2 Grundlagen des Softwaretestens 7**
2.1 Begriffe und Motivation . . . . . 8
2.1.1 Fehlerbegriff . . . . . 9
2.1.2 Testbegriff . . . . . 10
2.1.3 Softwarequalität . . . . . 12
2.1.4 Testaufwand . . . . . 14
2.2 Fundamentaler Testprozess . . . . . 18
2.2.1 Testplanung . . . . . 21
2.2.2 Testspezifikation . . . . . 23
2.2.3 Testdurchführung . . . . . 25
2.2.4 Testprotokollierung . . . . . 26
2.2.5 Auswertung und Bewertung des Testendes . . . . . 26
2.3 Sollwerte und Testorakel . . . . . 30
2.4 Priorisierung des Tests . . . . . 32
2.5 Psychologie des Testens . . . . . 34
2.6 Zusammenfassung . . . . . 36

**3 Testen im Softwarelebenszyklus 39**
3.1 Das allgemeine V-Modell . . . . . 39
3.2 Komponententest . . . . . 42
3.2.1 Begriffsklärung . . . . . 42
3.2.2 Testobjekte . . . . . 42
3.2.3 Testumgebung . . . . . 43
3.2.4 Testziele . . . . . 45
3.2.5 Teststrategie . . . . . 48
3.3 Integrationstest . . . . . 49
3.3.1 Begriffsklärung . . . . . 49
3.3.2 Testobjekte . . . . . 51
3.3.3 Testumgebung . . . . . 52
3.3.4 Testziele . . . . . 52
3.3.5 Integrationsstrategien . . . . . 54
3.4 Systemtest . . . . . 57
3.4.1 Begriffsklärung . . . . . 57
3.4.2 Testobjekt und Testumgebung . . . . . 57
3.4.3 Testziele . . . . . 59
3.4.4 Test funktionaler Anforderungen . . . . . 59
3.4.5 Test nicht funktionaler Anforderungen . . . . . 62
3.4.6 Probleme in der Systemtestpraxis . . . . . 64

3.5 Abnahmetest . . . . . 64
3.5.1 Test auf vertragliche Akzeptanz . . . . . 65
3.5.2 Test auf Benutzerakzeptanz . . . . . 65
3.5.3 Feldtest . . . . . 66
3.6 Test neuer Produktversionen . . . . . 67
3.6.1 Softwarewartung . . . . . 67
3.6.2 Geplante Weiterentwicklung . . . . . 69
3.6.3 Regressionstest . . . . . 70
3.7 Kosten- und Wirtschaftlichkeitsaspekte . . . . . 72
3.7.1 Fehlerkosten . . . . . 72
3.7.2 Testkosten . . . . . 73
3.7.3 Konsequenzen für die Teststrategie . . . . . 75
3.8 Testplanung . . . . . 76
3.8.1 Qualitätssicherungsplan . . . . . 76
3.8.2 Testkonzept . . . . . 77
3.9 Zusammenfassung . . . . . 77

**4 Statischer Test 79**
4.1 Strukturierte Gruppenprüfungen . . . . . 79
4.1.1 Grundlagen . . . . . 79
4.1.2 Reviews . . . . . 80
4.1.3 Grundlegende Vorgehensweise . . . . . 82
4.1.4 Rollen und Verantwortlichkeiten . . . . . 85
4.1.5 Reviewarten . . . . . 87
4.2 Statische Analyse . . . . . 91
4.2.1 Compiler als statisches Analysewerkzeug . . . . . 93
4.2.2 Prüfung der Einhaltung von Konventionen und Standards . . . . . 93
4.2.3 Durchführung der Datenflussanalyse . . . . . 94
4.2.4 Durchführung der Kontrollflussanalyse . . . . . 96
4.2.5 Ermittlung von Metriken . . . . . 96
4.3 Zusammenfassung . . . . . 99

**5 Dynamischer Test 101**
5.1 Black-box-Verfahren . . . . . 104
5.1.1 Äquivalenzklassenbildung . . . . . 104
5.1.2 Grenzwertanalyse . . . . . 115
5.1.3 Zustandsbezogener Test . . . . . 119
5.1.4 Allgemeine Bewertung der Black-box-Verfahren . . . 126
5.2 White-box-Verfahren . . . . . 128
5.2.1 Anweisungsüberdeckung . . . . . 129
5.2.2 Zweigüberdeckung . . . . . 131
5.2.3 Test der Bedingungen . . . . . 134
5.2.4 Pfadüberdeckung . . . . . 138
5.2.5 Allgemeine Bewertung der White-box-Verfahren . . 142
5.2.6 Instrumentierung und Werkzeugunterstützung . . . . 144

5.3 Intuitive Testfallermittlung . . . . . 144
5.4 Zusammenfassung . . . . . 146

**6 Testmanagement 149**
6.1 Organisation von Testteams . . . . . 149
6.2 Mitarbeiterqualifikation . . . . . 151
6.3 Management der Testarbeiten . . . . . 153
6.3.2 Testzyklusplanung . . . . . 153
6.3.3 Testzyklusüberwachung . . . . . 154
6.3.3 Testzyklussteuerung . . . . . 155
6.4 Fehlermanagement . . . . . 156
6.4.1 Testprotokoll . . . . . 156
6.4.2 Fehlermeldung . . . . . 157
6.4.3 Fehlerklassifikation . . . . . 159
6.4.4 Fehlerstatus . . . . . 161
6.5 Anforderungen an das Konfigurationsmanagement . . . . . 163
6.6 Relevante Normen und Standards . . . . . 165
6.7 Zusammenfassung . . . . . 166

**7 Testwerkzeuge 169**
7.1 Typen von Testwerkzeugen . . . . . 169
7.1.1 Werkzeuge für Testplanung und Testmanagement . 169
7.1.2 Werkzeuge zur Testspezifikation . . . . . 170
7.1.3 Werkzeuge zur Testdurchführung . . . . . 171
7.1.4 Werkzeuge zur Test- und Testobjektanalyse . . . . . 175
7.2 Auswahl und Einführung von Testwerkzeugen . . . . . 176
7.2.1 Wirtschaftlichkeit der Werkzeugeinführung . . . . . 177
7.2.2 Werkzeugauswahl . . . . . 179
7.2.3 Werkzeugeinführung . . . . . 180
7.3 Zusammenfassung . . . . . 181

**A Testkonzept nach IEEE 829 183**

**B Wichtige Hinweise zum Lehrstoff und zur Prüfung zum Certified Tester 189**

**C Übungsaufgaben 191**

**Glossar 195**

**Quellenverzeichnis 217**

**Index 223**

# 1 Einleitung

Software hat in den letzten Jahren eine enorme Verbreitung gefunden. Es gibt kaum noch Geräte, Maschinen oder Anlagen, in denen die Steuerung nicht über Software bzw. Softwareanteile realisiert wird. Beispielsweise werden im Automobil, vom Motormanagement über die Getriebesteuerung bis zum Bremsassistenten, immer mehr Steuerungsfunktionen von Mikroprozessoren übernommen und durch entsprechende Software umgesetzt. Software trägt somit ganz entscheidend zum Funktionieren der Geräte und Anlagen bei. Ebenso ist der reibungslose Ablauf eines Betriebs oder einer Organisation weitgehend von der Zuverlässigkeit der Softwaresysteme abhängig, die zur Abwicklung der Geschäftsprozesse oder einzelner Aufgaben eingesetzt werden. Wie schnell beispielsweise ein Versicherungskonzern ein neues Produkt oder auch nur einen neuen Tarif am Markt einführen kann, ist heutzutage zum großen Teil davon abhängig, wie schnell die konzerneigenen DV-Systeme entsprechend angepasst oder ausgebaut werden können.

*Hohe Abhängigkeit vom reibungslosen Funktionieren der Software*

In beiden Bereichen (technische und kommerzielle Softwaresysteme) ist die Qualität der Software zum entscheidenden Faktor für den Erfolg von Produkten oder Unternehmen geworden.

Viele Unternehmen haben diese Abhängigkeit von der Software erkannt und streben eine verbesserte Qualität ihrer Softwaresysteme und ihres Softwareentwicklungsprozesses an. Ein Mittel dies zu erreichen, ist das systematische Prüfen und Testen der entwickelten Software. Teilweise haben entsprechende Verfahren Einzug in die tägliche Praxis der Softwareentwicklung gefunden. In vielen Bereichen ist aber weiterhin ein erheblicher Bedarf an Wissensvermittlung der Prüf- und Testverfahren erforderlich.

*Grundlagenwissen zum strukturierten Prüfen und Testen*

Mit diesem Buch stellen wir Grundlagenwissen bereit, das bei entsprechender Umsetzung zu einem strukturierten, systematischen Vorgehen beim Prüfen und Testen führt und somit zur Qualitätsverbesserung der Software beiträgt. Der Inhalt des Buches ist so abgefasst, dass kein Vorwissen im Bereich der Softwarequalitätssicherung vorausgesetzt wird. Das Buch ist als Lehrbuch konzipiert und auch zum Selbststudium geeignet. Ein durchgängiges Fallbeispiel hilft, jedes dargestellte Thema und seine praktische Umsetzung schnell zu verstehen.

Weitere Beispiele und Übungsaufgaben (zu finden auch unter [URL: dPunkt]) helfen, das Gelernte zu vertiefen.

Ansprechen möchten wir Softwaretester[1] in Software- und Industrieunternehmen, die ihre Kenntnisse auf eine fundierte Grundlage stellen wollen, sowie Programmierer und Entwickler, die Testaufgaben übernommen haben bzw. übernehmen werden. Das Buch wird Projekt- und Teamleitern helfen, die Softwaretests effizienter zu gestalten. Ebenfalls werden Quereinsteiger in entwicklungsnahen IT-Berufen und Mitarbeiter in Fachabteilungen, die an der Abnahme, Einführung oder Weiterentwicklung von IT-Anwendungen beteiligt sind, Hilfestellung für ihre tägliche Arbeit finden.

Obwohl das Prüfen und Testen in der Praxis einen hohen Stellenwert hat (Aufwendungen in diesem Bereich werden mit 25 % bis 50 % der Entwicklungszeit und -kosten beziffert), gibt es an Universitäten, Fachhochschulen und Berufsakademien im Bereich Informatik nur wenige Lehrveranstaltungen, die sich intensiv mit dieser Problematik auseinander setzen. Das Buch soll Lernende und Lehrende im gleichen Maße ansprechen.

Das lebenslange Lernen ist besonders im IT-Bereich unverzichtbar. Weiterbildungsmaßnahmen werden von vielen Firmen angeboten. Oft besteht Unsicherheit bei den Interessenten über die Qualität der angebotenen Kurse. Um im Weiterbildungsbereich für bessere Vergleichbarkeit der Kursanbieter untereinander und für einen allseits anerkannten Nachweis des erworbenen Wissens der Kursteilnehmer zu sorgen, ist in England das *Information Systems Examinations Board* [URL: ISEB] ins Leben gerufen und der *British Computer Society* [URL: BCS] angegliedert worden.

*Dreistufiges Qualifizierungsprogramm*

ISEB ist in unterschiedlichen Bereichen tätig, unter anderem auch im Bereich Softwaretest. 1998 wurde ein dreistufiges Qualifizierungsprogramm definiert. Die Grundlagen zum Softwaretest sind im Lehrplan zum Erhalt des *Foundation Certificate* [URL: ISEB CTF 99] beschrieben. Darauf aufbauend kann seit 2002 das *Practitioner Certificate* [URL: ISEB CTP 01] erworben werden, um vertiefte Kenntnisse im Prüfen und Testen während der Softwareentwicklung nachzuweisen. Ein drittes weiterführendes Zertifikat ist geplant.

*Internationale Initiative*

Die ISEB-Aktivitäten wurden von anderen Ländern aufgegriffen, und ähnliche Initiativen wurden gestartet. Es sind landesspezifische *Testing Boards* etabliert worden, die gemeinsam das *International*

1. Wir verwenden im Buch die männliche Form und wollen damit Frauen selbstverständlich nicht ausschließen bzw. ausgrenzen.

*Software Testing Qualification Board* [URL: ISTQB] bilden. Die Zusammenhänge werden in Abbildung 1-1 verdeutlicht.

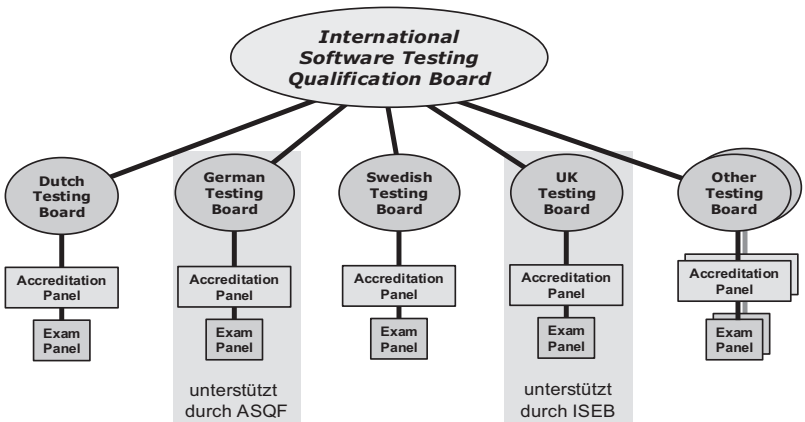



***Abb. 1–1***
*International Software Testing Qualification Board*

Die landesspezifischen *Testing Boards* sind zuständig für die Akkreditierung von Trainingsanbietern und die Durchführung von Prüfungen in ihren jeweiligen Ländern. Das *International Software Testing Qualification Board* koordiniert die nationalen Initiativen und sorgt für die Einheitlichkeit und Vergleichbarkeit der Lehr- und Prüfungsinhalte unter den beteiligten Ländern.

In Deutschland hat der ASQF [URL: ASQF] und die Fachgruppe TAV (Test, Analyse und Verifikation von Software, FG 2.1.7) der Gesellschaft für Informatik e. V. [URL: GI] die ISEB-Aktivitäten mit dem Ziel aufgegriffen, ein entsprechendes Qualifizierungs- und Prüfungsschema aufzubauen.

*German Testing Board*

Die fachlichen Inhalte werden vom *German Testing Board* [URL: GTB] überwacht. In diesem Gremium sind Trainingsanbieter, Testexperten aus Industrie und Beratungsunternehmen sowie Vertreter der GI-Fachgruppe TAV unter dem Dach des ASQF organisiert, um gemeinsam die Lehrpläne [URL: GTB CTF 01] und das Prüfungswesen weiterzuentwickeln. Das Board, als unabhängige Instanz, überprüft die angebotenen Kurse nach definierten Kriterien und spricht eine Akkreditierung der Trainingsanbieter aus. Es sorgt für die regelmäßige Durchführung von Prüfungen, bei denen die Kursteilnehmer nach bestandener Prüfung einen anerkannten Qualifikationsnachweis erhalten.

Der Inhalt des Buches entspricht den Anforderungen des *Foundation Certificate*. Das prüfungsrelevante Fachwissen kann im Selbststudium erworben oder nach bzw. parallel zu einer Teilnahme an einem

Kurs vertieft werden. Wir haben die Absicht, die weiterführenden Kursinhalte in ähnlicher Weise zu publizieren.

*Kapitelübersicht*

Die Themen des Buches und somit auch die grobe Struktur der Inhalte der Kurse zum Erwerb des *Foundation Certificate* sind im Folgenden beschrieben.

*Grundlagen*

In Kapitel 2 werden die Grundlagen des Softwaretests erörtert. Neben der Motivation, wann, mit welchen Zielen und wie intensiv getestet werden soll, wird das Konzept eines grundlegenden Testprozesses beschrieben. Es wird auf die psychologischen Schwierigkeiten eingegangen, die entstehen, wenn beim Test der eigenen Software die selbst verursachten Fehler nachgewiesen werden sollen.

*Testen im Softwarelebenszyklus*

Welche Testaktivitäten wann im Softwareentwicklungsprozess durchgeführt werden sollen, wird in Kapitel 3 behandelt. Neben den verschiedenen Teststufen und -phasen wird auf die Unterschiede beim funktionalen und nicht funktionalen Test eingegangen. Die Wirtschaftlichkeit des Testens und das Thema Regressionstest werden ebenfalls angesprochen.

*Statischer Test*

Statische Verfahren, das heißt Verfahren, bei denen das Testobjekt nicht zur Ausführung kommt, werden in Kapitel 4 vorgestellt. Reviews und statische Analysen werden bereits in vielen Unternehmen mit gutem Erfolg angewendet. Die unterschiedlichen Vorgehensweisen werden ausführlich beschrieben.

*Dynamischer Test*

Das fünfte Kapitel behandelt den Test im engeren Sinne. Die Klassifizierung der dynamischen Testverfahren in *Black-box-* und *White-box*-Verfahren wird erörtert. Zu jeder Klasse werden unterschiedliche Testverfahren bzw. -methoden an Beispielen ausführlich erklärt. Auf die sinnvolle Verwendung des intuitiven Tests, nämlich in Ergänzung zu den anderen Verfahren, wird am Ende des Kapitels eingegangen.

*Testmanagement*

Welche Aspekte beim Testmanagement zu berücksichtigen sind, wie eine systematische Fehlerverfolgung aussieht und dass auch ein ausreichendes Konfigurationsmanagement etabliert sein muss, wird im sechsten Kapitel diskutiert.

*Testwerkzeuge*

Testen von Software ist ohne eine entsprechende Werkzeugunterstützung sehr arbeits- und zeitintensiv. Im siebten und letzten Kapitel des Buches werden unterschiedliche Klassen von Werkzeugen zur Testunterstützung vorgestellt und Hinweise zur Werkzeugauswahl und -implementierung gegeben.

*Hinweise zu Lehrstoff und Prüfung im Anhang*

Im Anhang werden wichtige Hinweise zum Lehrstoff und zur Prüfung zum Certified Tester gegeben. Weitere Anhänge des Buches beinhalten Erläuterungen zum Testplan nach [IEEE 829], beispielhafte Übungsaufgaben, ein Glossar und das Literaturverzeichnis. Die Fachbegriffe sind im Buchtext beim ersten Auftreten mit einem entsprechen-

den →Hinweis versehen, der auf die genauere Definition des Begriffs im Glossar verweist. Textpassagen, die über den Stoff des Lehrplans hinausgehen, sind als *Exkurs* gekennzeichnet.

*WWW-Seite zum Buch mit aktuellen und weiteren Informationen*

Unter der [URL: dPunkt] finden sich weitere und aktuelle Informationen, u. a. Übungsaufgaben mit Lösungen, und eine Auswahl an Testwerkzeugen zum Herunterladen steht zur Verfügung.

# 2 Grundlagen des Softwaretestens

*Dieses einleitende Kapitel erklärt die Grundbegriffe des Softwaretests, die in den weiteren Kapiteln vorausgesetzt werden. Wichtige Begriffe werden zusätzlich an einem praxisnahen Fallbeispiel veranschaulicht, das im gesamten Buch immer wieder zur Illustration und Motivation des Lehrstoffs verwendet wird. Dieses Fallbeispiel wird zunächst eingeführt.*

Die in diesem Buch vorgestellten Vorgehensweisen beim Testen von Software werden größtenteils anhand eines durchgängigen Fallbeispiels veranschaulicht. Das diesem Beispiel zugrunde liegende Szenario ist folgendes:

***Fallbeispiel »VirtualShowRoom« – VSR***

Ein Automobilkonzern entwickelt ein neues elektronisches Verkaufssystem, genannt *VirtualShowRoom* (VSR). Dieses Softwaresystem soll in der Endausbaustufe weltweit bei allen Händlern installiert sein. Jeder Kunde, der ein Fahrzeug erwerben möchte, kann dann unterstützt durch einen Verkäufer oder vollkommen selbstständig sein Wunschfahrzeug am Bildschirm konfigurieren (Modellauswahl, Farbe, Ausstattung usw.).

Das System zeigt mögliche Modelle und Ausstattungsvarianten an und ermittelt zu jeder Auswahl des Kunden sofort den jeweiligen Listenpreis. Diese Funktionalität wird vom Teilsystem *DreamCar* realisiert.

Hat sich der Kunde für ein Fahrzeug entschieden, kann er am Bildschirm die für ihn optimale Finanzierung kalkulieren (*EasyFinance*), das Fahrzeug online bestellen (*JustInTime*) und natürlich auch die passende Versicherung (*NoRisk*) abschließen. Das Teilsystem *ContractBase* schließlich verwaltet sämtliche Kundeninformationen und Vertragsdaten.

Die folgende Abbildung 2-1 zeigt eine schematische Darstellung dieses Systems.

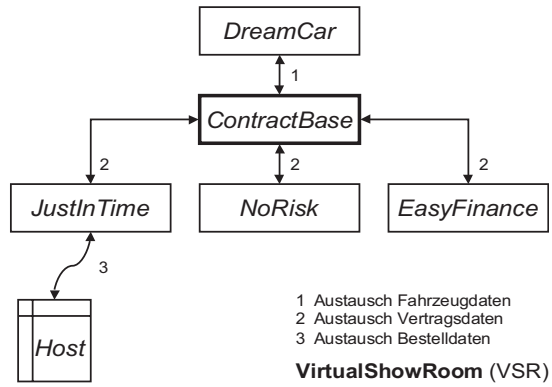



***Abb. 2–1***
*Architektur des VSR-Systems*

Jedes Teilsystem wird von einem eigenen Entwicklungsteam separat entworfen und entwickelt. Insgesamt sind ca. 50 Entwickler und weitere Mitarbeiter aus den jeweils betroffenen konzerninternen Fachabteilungen an dem Projekt beteiligt, sowie externe Softwarefirmen.

Bevor das VSR-System in Betrieb gehen kann, muss es gründlich getestet werden. Die mit diesen Testarbeiten betrauten Projektmitarbeiter setzen hierzu verschiedene Testtechniken und Vorgehensweisen ein. Das Grundlagenwissen, das sie dazu benötigen, wird in diesem Buch vermittelt.

## 2.1 Begriffe und Motivation

*Requirements*

Bei der Herstellung eines Industrieprodukts werden die entstehenden Teilprodukte und das Endprodukt üblicherweise daraufhin kontrolliert, ob sie den gestellten →Anforderungen entsprechen. Es wird geprüft, ob das Produkt die geforderte Aufgabe löst. Je nach Produkt gibt es auch unterschiedliche Anforderungen an die Qualität der Lösung. Erweist sich ein Produkt als fehlerhaft, so müssen ggf. Korrekturen im Produktionsprozess oder in der Konstruktion erfolgen.

*Software ist immateriell*

Was allgemein für die Herstellung eines Industrieprodukts gilt, trifft entsprechend für die Produktion bzw. Entwicklung von Software zu. Die Prüfung der Teilprodukte bzw. des Endprodukts gestaltet sich allerdings schwieriger, da das erstellte Produkt nicht »greifbar«, also immateriell ist und eine →Prüfung nicht »handfest« durchgeführt werden kann. Eine optische Prüfung ist nur sehr eingeschränkt durch intensives Lesen der Entwicklungsdokumente möglich. Beim →Testen wird die zu prüfende Software auf einem Rechner zur Ausführung gebracht, und es muss festgestellt werden, ob die Software sich entsprechend oder nicht entsprechend den Anforderungen verhält. Dem Testen von Software

kommt somit eine sehr wichtige und schwierige Aufgabe im Rahmen der Softwareentwicklung zu.

***Beispiel***

Auch das VSR-System des Fallbeispiels muss vor seinem Einsatz intensiv geprüft werden, um mögliche →Fehler vor der Auslieferung zu erkennen und beheben zu können. Führt das System beispielsweise Bestellvorgänge falsch aus, könnte dies für Kunden und Händler, aber auch für den Autokonzern unter Umständen einen großen finanziellen Schaden oder Imageverlust zur Folge haben.

### 2.1.1 Fehlerbegriff

*Was gilt als Fehler oder Mangel?*

Wann liegt ein nicht anforderungskonformes Verhalten des Systems vor? Eine Situation kann nur dann als fehlerhaft eingestuft werden, wenn vorab festgelegt wurde, wie die erwartete, korrekte, also nicht fehlerhafte Situation aussehen soll. Ein Fehler ist somit die Nichterfüllung einer festgelegten Anforderung, eine Abweichung zwischen dem →Istverhalten (während der Ausführung der Tests oder des Betriebs festgestellt) und dem →Sollverhalten (in der Spezifikation oder den Anforderungen festgelegt). Ein →Mangel liegt vor, wenn eine gestellte Anforderung oder eine berechtigte Erwartung nicht angemessen erfüllt wird. Ein Mangel ist beispielsweise die Beeinträchtigung der Verwendbarkeit bei gleichzeitiger Erfüllung der →Funktionalität oder die Nichterfüllung einer angemessenen Erwartung.

*Ursachenkette für Fehler*

Im Gegensatz zu physikalischen Systemen entstehen Fehler in einem Softwaresystem nicht durch Alterung oder Verschleiß. Jeder Fehler oder Mangel ist seit dem Zeitpunkt der Entwicklung in der Software vorhanden. Er kommt jedoch erst bei der Ausführung der Software zum Tragen.

*Fehlerwirkung*

In der [DIN 66271] wird zur Beschreibung dieses Sachverhalts der Begriff →Fehlerwirkung verwendet. Der englische Fachbegriff hierfür lautet *failure*. Weitere Bezeichnungen sind →äußerer Fehler oder →Ausfall. Beim Test der Software oder auch erst bei deren Betrieb wird eine Fehlerwirkung für den →Tester oder Anwender nach außen sichtbar. Zum Beispiel ist ein Ausgabewert falsch oder das Programm stürzt ab.

*Fehlerzustand*

Zwischen dem Auftreten einer Fehlerwirkung und deren Ursache muss unterschieden werden. Eine Fehlerwirkung hat ihren Ursprung in einem →Fehlerzustand (engl. *fault*) der Software. Dieser Fehlerzustand wird auch als →Defekt oder →innerer Fehler bezeichnet. Dies ist beispielsweise eine falsch programmierte oder vergessene Anweisung im Programm.

*Fehlermaskierung*

Es ist durchaus möglich, dass ein Fehlerzustand durch einen oder mehrere andere Defekte in anderen Teilen des Programms kompensiert wird (→Fehlermaskierung). Eine Fehlerwirkung tritt in diesem Fall erst dann zutage, nachdem der oder die maskierenden Fehler korrigiert worden sind.

*Fehlhandlung*

Ursache für das Vorliegen eines Fehlerzustands oder Defekts ist wiederum die vorausgegangene →Fehlhandlung einer Person, beispielsweise die fehlerhafte Programmierung des Entwicklers. Der englische Begriff hierfür ist *error*. Genauere Erklärungen zu den Begriffen im Bereich Testen werden im folgenden Abschnitt gegeben.

### 2.1.2 Testbegriff

*Testen ist nicht Debugging*

Um einen Defekt korrigieren zu können, muss der Defekt im Softwareprodukt lokalisiert werden. Bekannt ist zunächst nur seine Wirkung, aber nicht die genaue Stelle in der Software, die den Defekt darstellt. Das Lokalisieren und Beheben des Defekts ist Aufgabe des Softwareentwicklers und wird auch als →Debugging[2] bezeichnet. Debugging und Testen werden oft gleichgesetzt. Es sind aber zwei völlig unterschiedliche und getrennte Aufgaben.

Während Debugging das Ziel hat, Defekte bzw. Fehlerzustände zu beheben, ist es Aufgabe des Tests, Fehlerwirkungen (die auf Defekte hinweisen) gezielt und systematisch aufzudecken.

*Ein Test ist eine stichprobenhafte Prüfung*

Unter dem Testen von Software wird jede (im Allgemeinen stichprobenartige) Ausführung eines →Testobjekts verstanden, die der Überprüfung des Testobjekts dient. Die →Randbedingungen für die Ausführung des Tests müssen festgelegt sein. Ein Vergleich zwischen Soll- und Istverhalten des Testobjekts dient zur Bestimmung, ob das Testobjekt die geforderten Eigenschaften erfüllt.

Oft wird der gesamte Prozess, ein Programm auf systematische Weise auszuführen, um die korrekte Umsetzung der Anforderungen nachzuweisen und Fehlerwirkungen aufzudecken, als Test bezeichnet.

Zum →Testprozess gehören neben der Aktivität des Ausführens des Testobjekts mit →Testdaten auch die Planung, Durchführung und das Auswerten der Tests (→Testmanagement). Ein →Testlauf umfasst die Ausführung eines oder mehrerer →Testfälle. Zu einem Testfall gehören die festgelegten Randbedingungen, meist sind dies die Voraussetzungen zur Ausführung, die Eingabewerte und die erwarteten Ausgaben bzw. das erwartete Verhalten des Testobjekts. Ein Testfall sollte so gewählt werden, dass er eine gewisse Wahrscheinlichkeit besitzt, eine bisher nicht bekannte Fehlerwirkung aufzudecken.

---

2. Es gibt leider keine gebräuchliche Übersetzung ins Deutsche.

Oft können mehrere Testfälle zu →Testszenarien aneinander gereiht werden, wobei das Ergebnis eines Testfalls als Ausgangssituation für den folgenden Testfall genutzt wird. So kann beispielsweise ein Testszenario für eine Datenbankapplikation aus einem Testfall bestehen, der ein Datum in die Datenbank schreibt, einem weiteren Testfall, der dieses Datum manipuliert, und einem dritten Testfall, der das geänderte Datum aus der Datenbank ausliest. Alle Testfälle hintereinander werden dann in einem Testlauf zur Ausführung gebracht.

*Kein umfangreiches Softwaresystem ist fehlerfrei*

Ein fehlerfreies Softwaresystem gibt es derzeit nicht und wird es in naher Zukunft wahrscheinlich nicht geben, sobald das System einen gewissen Grad an Komplexität und Umfang an Programmzeilen umfasst. Häufig liegt ein Fehlerzustand darin begründet, dass sowohl während der Entwicklung als auch beim Testen der Software gewisse Ausnahmesituationen nicht bedacht bzw. nicht überprüft wurden. Sei es das Schaltjahr, das nicht richtig berechnet wird, oder die nicht berücksichtigten Randbedingungen, wenn es um Zeitverhalten oder Ressourcenbedarf geht. Auf der anderen Seite arbeiten aber sehr viele Softwaresysteme in ganz unterschiedlichen Bereichen zuverlässig tagein, tagaus.

*Fehlerfreiheit ist durch Testen nicht erreichbar*

Selbst wenn alle ausgeführten Testfälle keinen einzigen Fehler mehr aufdecken, kann (außer bei sehr sehr kleinen Programmen) nicht mit völliger Sicherheit ausgeschlossen werden, dass es nicht zusätzliche Testfälle gibt, die weitere Fehlerwirkungen erzeugen und somit weitere Fehlerzustände in der Software aufzeigen.

***Exkurs: Benennung von Tests***

Es gibt verwirrend viele Bezeichnungen für verschiedene Arten von Softwaretests. Einige werden im Rahmen der Darstellung der verschiedenen Teststufen (s. Kap. 3) genauer erklärt. Dieser Exkurs soll etwas Systematik in die Begriffsvielfalt bringen. Dazu ist es hilfreich, folgende vier Kategorien, nach denen Tests bezeichnet werden, zu unterscheiden:

1. →**Testziel**: Die Benennung einer Testart aufgrund des Zwecks des Tests (z. B. →Lasttest).
2. **Testmethode**: Der Test wird nach der Methode benannt, die zur Spezifikation oder Durchführung der Tests eingesetzt wird (z. B. →geschäftsprozessbasierter Test).
3. **Testobjekt**: Der Test wird nach der Art des Testobjekts, das getestet wird, benannt (z. B. →Klassentest).
4. **Teststufe**: Der Test wird nach der Phase/Stufe des zugrunde liegenden Vorgehensmodells benannt (z. B. →Systemtest).

Nicht hinter jedem Begriff verbirgt sich also eine neue oder andere Art von Test. Vielmehr wird, je nachdem unter welchem Blickwinkel die Testarbeiten betrachtet werden, einer der Aspekte Testziel, Testmethode, Testobjekt oder Teststufe in den Vordergrund gerückt.

### 2.1.3 Softwarequalität

Das Testen von Software dient durch die Identifizierung von Defekten und deren anschließender Beseitigung durch das Debugging zur Steigerung der →Softwarequalität. Softwarequalität umfasst aber mehr als nur die Beseitigung der beim Test aufgedeckten Fehlerwirkungen. Nach der DIN-ISO-Norm 9126 [ISO 9126] gehören zur Softwarequalität folgende Faktoren: Funktionalität, →Zuverlässigkeit, Benutzbarkeit, →Effizienz, Änderbarkeit und Übertragbarkeit.

Alle diese Faktoren bzw. →Qualitätsmerkmale sind beim Testen zu berücksichtigen, um die Gesamtqualität eines Softwareprodukts beurteilen zu können. Welches Qualitätsniveau das Testobjekt dabei in jeder einzelnen Merkmalsdimension aufweisen soll, muss vorab als Qualitätsanforderung festgelegt werden. Die Erfüllung dieser Anforderungen ist dann durch geeignete Tests zu überprüfen.

***Beispiel VirtualShowRoom***

Beim VSR-System muss vom Auftraggeber festgelegt sein, welche Qualitätsmerkmale eine hohe Bedeutung für ihn haben. Diese sind im System entsprechend umzusetzen und intensiv zu prüfen. Die Merkmale Funktionalität, Zuverlässigkeit und Benutzbarkeit haben einen sehr hohen Stellenwert für den Automobilkonzern. Das System muss zuverlässig die geforderte Funktionalität anbieten. Darüber hinaus muss es einfach zu bedienen sein, damit es bei den einzelnen Händlern ohne große Probleme im Alltag eingesetzt werden kann. Die Prüfung der Realisierung dieser geforderten Qualitätsmerkmale ist besonders sorgfältig vorzunehmen.

*Funktionalität*

Die einzelnen Qualitätsmerkmale der DIN-ISO-Norm 9126 [ISO 9126] werden im Folgenden kurz erörtert. Das Qualitätsmerkmal Funktionalität umfasst alle Charakteristiken, welche die geforderten Fähigkeiten des Systems beschreiben. Die Fähigkeiten werden meist durch ein spezifiziertes Ein-/Ausgabe-Verhalten und/oder eine entsprechende Reaktion oder Wirkung des Systems auf entsprechende Eingaben beschrieben. Aufgabe des Tests ist es nachzuweisen, dass jede einzelne geforderte Fähigkeit im System so realisiert wurde, dass das spezifizierte Ein-/Ausgabe-Verhalten oder die spezifizierte Reaktion erfüllt wird. Nach DIN-ISO-Norm gliedert sich das Merkmal Funktionalität in die Teilmerkmale: Angemessenheit, Richtigkeit, Interoperabilität, Ordnungsmäßigkeit und Sicherheit.

Eine angemessene Umsetzung ist gegeben, wenn jede geforderte Fähigkeit im System vorhanden ist und geeignet realisiert wurde. Dabei ist selbstverständlich darauf zu achten und somit beim Test zu überprüfen, dass die richtigen bzw. spezifizierten Reaktionen oder Wirkungen von dem System geliefert werden.

Softwaresysteme müssen mit anderen Systemen zusammenwirken, zumindest mit dem Betriebssystem, es sei denn, das Betriebssystem ist selbst das Testobjekt. Mit Interoperabilität wird das Zusammenspiel zwischen dem zu testenden System und den vorgegebenen Systemen bezeichnet. Ein möglicherweise fehlerhaftes Zusammenwirken soll im Test aufgedeckt werden.

Eine ordnungsgemäße Funktionalität liegt vor, wenn die Software anwendungsspezifische Normen, Vereinbarungen oder gesetzliche Bestimmungen und ähnliche Vorschriften erfüllt. Aspekte der Zugriffs- und Datensicherheit haben bei sehr vielen Anwendungen eine hohe Bedeutung. Es ist nachzuweisen, dass ein unberechtigter Zugriff, sowohl versehentlich als auch vorsätzlich, auf Programme und Daten verhindert wird.

*Zuverlässigkeit*

Zuverlässigkeit beschreibt die Fähigkeit eines Systems, ein Leistungsniveau unter festgelegten Bedingungen über einen definierten Zeitraum zu bewahren. Das Qualitätsmerkmal wird in der Norm unterteilt in Reife, →Fehlertoleranz und Wiederherstellbarkeit.

Unter Reife wird verstanden, wie häufig ein →Versagen der Software durch Fehlerzustände vorkommt.

Fehlertoleranz liegt vor, wenn das Softwareprodukt sein spezifiziertes Leistungsniveau bewahrt oder wieder erlangt, nachdem ein Defekt, ein falscher Schnittstellenaufruf oder eine falsche Bedienung durch den Anwender zu einer Fehlerwirkung geführt hat.

Wie einfach und schnell das geforderte Leistungsniveau nach einem Versagen oder Ausfall wieder erreicht werden kann, beschreibt das Qualitätsmerkmal Wiederherstellbarkeit. Dabei sind der benötigte Aufwand und die benötigte Zeit zum Erreichen des Leistungsniveaus und der Wiedergewinnung der direkt betroffenen Daten zu berücksichtigen.

*Benutzbarkeit*

Bei interaktiven Softwaresystemen spielt die Benutzbarkeit eine entscheidende Rolle für die Akzeptanz der Systeme. Wie hoch ist der Aufwand, der zur Benutzung der Software erforderlich ist unter Berücksichtigung von unterschiedlichen Benutzergruppen? Verständlichkeit, Erlernbarkeit und Bedienbarkeit sind Teilaspekte der Benutzbarkeit. Überprüft werden diese Qualitätsmerkmale u. a. beim →nicht funktionalen Test (s. Kap. 3.4.5).

*Effizienz*

Messbare Ergebnisse lassen sich beim Test der Effizienz erzielen. Die benötigte Zeit und der Verbrauch an Betriebsmitteln für die Erfüllung einer Aufgabe werden mit dem Qualitätsmerkmal Effizienz umschrieben.

*Änderbarkeit und Übertragbarkeit*

Softwaresysteme werden oft über einen längeren Zeitraum und auf unterschiedlichen Plattformen (Betriebssystem und Hardware) einge-

setzt, deshalb sind die beiden letzten Qualitätskriterien von großer Bedeutung: Änderbarkeit und Übertragbarkeit.

Änderbarkeit umfasst die einzelnen Aspekte Analysierbarkeit, Modifizierbarkeit, Stabilität und Prüfbarkeit.

Anpassbarkeit, Installierbarkeit, Konformität und Austauschbarkeit sind bei der Übertragbarkeit der Softwaresysteme zu berücksichtigen. Viele der Aspekte zur Änder- und Übertragbarkeit lassen sich nur durch →statische Analysen (s. Kap. 4.2) prüfen.

Ein Softwaresystem kann nicht alle Qualitätsmerkmale gleich gut erfüllen. Teilweise kann die Erfüllung eines Merkmals auch die Nichterfüllung eines anderen bewirken. So wird ein hoch effizientes Softwaresystem nur sehr eingeschränkt übertragbar sein, weil die Entwickler zur Effizienzsteigerung meistens spezielle Eigenschaften der verwendeten Plattform ausnutzen, die wiederum die Übertragbarkeit negativ beeinflussen.

*Qualitätsmerkmale priorisieren*

Es muss deshalb festgelegt werden, welche Qualitätseigenschaften mit welcher Priorität umgesetzt werden sollen. Diese Festlegung dient dann auch für den Test als Richtschnur zur Bestimmung der Intensität der Überprüfung der einzelnen Qualitätsmerkmale. Mit welchem Aufwand dies verbunden ist, wird im nächsten Kapitel beschrieben.

### 2.1.4 Testaufwand

*Vollständiges Testen ist nicht möglich*

Ein →vollständiger Test ist praktisch nicht durchführbar. Ein vollständiger Test umfasst die Kombination aller möglichen Eingaben unter Berücksichtigung aller unterschiedlichen Randbedingungen, die Einfluss auf den Systemablauf haben. Durch die Vielzahl kombinatorischer Möglichkeiten ergibt sich eine nahezu unbegrenzte Anzahl an Tests, die durchzuführen wären. Ein solches »Austesten« aller Kombinationen ist nicht möglich.

***Beispiel***

Der Sachverhalt soll mit einem Beispiel für →kontrollflussbasiertes Testen verdeutlicht werden (aus [Myers 82]):

Getestet wird ein triviales Programm mit einem einfachen →Kontrollfluss. Das Programm besteht aus vier Verzweigungen (IF-Anweisungen), die teilweise ineinander verschachtelt sind. Der entsprechende →Kontrollflussgraph des Programms ist in Abb. 2-2 dargestellt. Zwischen den Punkten A und B ist eine Schleife, mit einem Rücksprung von Punkt B nach Punkt A. Soll das Programm in Bezug auf die unterschiedlichen kontrollflussbasierten Möglichkeiten des Programmablaufs vollständig getestet oder »ausgetestet« werden, so ist jeder mögliche Ablauf, dass heißt jede mögliche Kombination von Programmteilen, zur Ausführung zu bringen. Bei einer Beschränkung der

Schleife auf maximal 20 Durchläufe ergibt sich unter der Annahme, dass alle Verzweigungen unabhängig voneinander sind, folgende Rechnung:

$$5^{20} + 5^{19} + 5^{18} + ... + 5^{1}$$

wobei 5 die Anzahl der möglichen Wege innerhalb der Schleife ist. $5^1$ Testfälle ergeben sich durch die Ausführung jedes einzelnen möglichen Weges in der Schleife, aber jeweils ohne Rücksprung zum Schleifenanfang. Werden Testfälle durchgeführt, die einen einzigen Rücksprung zum Schleifenanfang bewirken, dann sind insgesamt $5*5=5^2$ unterschiedliche Möglichkeiten der Ausführung zu berücksichtigen usw. Entsprechend ergibt das Gesamtergebnis der Rechnung etwa 100 Billiarden unterschiedliche Abläufe des Programms.

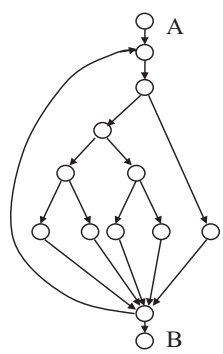



***Abb. 2–2***
*Kontrollflussgraph eines kleinen Programms*

Wird von einem manuell durchgeführten Test ausgegangen und wird wie in [Myers 82] für jeden zu spezifizierenden, auszuführenden und auszuwertenden Testfall ein Zeitaufwand von fünf Minuten veranschlagt, dauert die Abarbeitung aller Testfälle insgesamt eine Milliarde Jahre. Werden statt der fünf Minuten fünf Mikrosekunden veranschlagt, weil der Test weitestgehend automatisiert abläuft, wären es immer noch 19 Jahre.

---

*Testaufwand zwischen 25 % und 50 %*

Ein Programm vollständig zu testen, ist also in der Praxis nicht möglich. Es wird immer nur ein Teil aller denkbaren Testfälle berücksichtigt werden können. Aber auch dann nimmt Testen einen hohen Anteil am Entwicklungsaufwand in Anspruch. Eine Verallgemeinerung der Höhe des →Testaufwands ist jedoch schwierig, da sie stark vom Charakter des Projekts abhängt. Im Folgenden werden einige beispielhafte Zahlen aus Projekten eines großen deutschen Softwarehauses zur Verdeutlichung der Bandbreite unterschiedlicher Aufwände bezogen auf das Gesamtbudget der Entwicklung angeführt:

- Für einige Großprojekte mit mehr als zehn Personenjahren Aufwand umfassten Codierung und Test zusammen etwa 40 %, und für die Integration kamen weitere 8 % hinzu. Bei testintensiven Projekten (z. B. Jahr-2000-Umstellung) kann der Testaufwand auf bis zu 80 % steigen.
- In einem Projekt war der Testaufwand 1,2-mal so hoch wie der Codierungsaufwand. Vom Testaufwand entfielen 2/3 auf den →Komponententest.
- 51,9 % umfasste der Systemtest bei einem weiteren Projekt des Softwarehauses.

*Fehler können hohe Kosten verursachen*

Aber ist dieser hohe Testaufwand bezahlbar und gerechtfertigt? Die Gegenfrage von Tom DeMarco lautet: »Im Vergleich zu was?« Er weist damit auf die zu berücksichtigenden Risiken bei fehlerhaften Softwaresystemen hin. Das Risiko wird aus der Eintrittswahrscheinlichkeit und der zu erwartenden Schadenshöhe ermittelt. Im Test nicht gefundene Fehlerzustände können beim Einsatz der Software hohe Kosten verursachen. In einem Artikel der Frankfurter Allgemeinen Zeitung vom 17. 1. 2002 mit der Überschrift »IT-Systemausfälle kosten viele Millionen« wird eine Stunde Systemausfall im Börsenhandel mit Kosten in Höhe von 7,8 Mio. US-$ veranschlagt. Bei einem Fehlverhalten eines →sicherheitsrelevanten Systems können unter Umständen Leib und Leben von Personen gefährdet werden.

Da ein vollständiger Test nicht möglich ist, muss der Testaufwand in einem vernünftigen Verhältnis zum erzielbaren Ergebnis stehen. »Testen ist ökonomisch sinnvoll, solange die Kosten für das Finden und Beseitigen eines Fehlers im Test niedriger sind als die Kosten, die mit dem Auftreten eines Fehlers bei der Nutzung verbunden sind« [Pol 00]. Der Testaufwand muss daher immer in Abhängigkeit mit dem im Fehlerfall verbundenen Risiko und der Gefährdungsbewertung stehen.

***Beispiel für ein hohes Risiko im Fehlerfall***

Beim VSR-System soll der Kaufinteressent selbstständig sein Wunschfahrzeug am Bildschirm konfigurieren. Wird dabei vom System eine fehlerhafte Preisberechnung durchgeführt, kann der Kunde auf den angezeigten Preis bestehen. Bei einer Ausbaustufe des VSR-Systems plant der Konzern, ein webbasiertes Verkaufsportal anzubieten. Hier kann eine fehlerhafte Preisberechnung dazu führen, dass tausende von Fahrzeugen zu einem zu geringen Preis verkauft werden. Der Gesamtschaden kann, je nachdem wie stark sich das VSR-System je Fahrzeug »verrechnet«, in die Millionen gehen. Aus juristischer Sicht kommt mit der Online-Bestellung ein gültiger Kaufvertrag zum angegebenen Preis zustande.

Systeme mit einem hohen Risiko müssen ausgiebiger getestet werden als Systeme, die im Fehlerfall keine großen Schäden verursachen. Wobei die Risikoeinschätzung für die einzelnen Systemteile oder sogar für einzelne Fehlermöglichkeiten durchzuführen ist. Bei einem hohen Risiko im Fehlverhalten eines Systems oder Teilsystems ist für den Test ein höherer Aufwand zu veranschlagen als bei weniger kritischen System(teil)en. Die Intensität des Tests muss entsprechend hoch sein.

Für die Herstellerfirma eines Computerspiels kann ein fehlerhaftes Speichern eines Spielstandes ein sehr hohes Risiko bedeuten (obwohl es keine direkten Schäden verursacht), da das fehlerhafte Spiel bei der Kundschaft nicht akzeptiert wird und zu hohen Absatzverlusten, möglicherweise bei allen Spielen der Firma, führen kann.

*Testintensität und -umfang in Abhängigkeit vom Risiko festlegen*

Bei jedem Softwareprogramm ist also zu entscheiden, wie intensiv und wie umfangreich es zu testen ist. Diese Entscheidung muss in Abhängigkeit zum erwarteten Risiko bei fehlerhaftem Verhalten des Programms getroffen werden. Da ein vollständiger Test nicht möglich ist, ist ganz entscheidend, wie die beschränkten (Test-)Ressourcen eingesetzt werden. Um zu einem zufrieden stellenden Ergebnis zu kommen, müssen die Tests strukturiert und systematisch durchgeführt werden. Nur so ist es mit angemessenem Aufwand möglich, viele Fehlerwirkungen nachzuweisen und →unnötige Tests, die zu keinen neuen Erkenntnissen führen, zu vermeiden.

*Adäquate Testverfahren auswählen*

Es gibt eine ganze Reihe von unterschiedlichen Verfahren und Methoden zum Testen von Software. Jedes Verfahren berücksichtigt und prüft einen bestimmten Aspekt des Testobjekts besonders intensiv. So steht bei den kontrollflussbasierten Testverfahren der Ablauf des Programms im Mittelpunkt der Untersuchungen. Bei den →datenflussbasierten Testverfahren ist der Gebrauch der Daten und deren Weitergabe der Untersuchungsgegenstand. Es gibt kein Testverfahren, das für jeden Aspekt gleich gut geeignet ist. Eine Kombination von unterschiedlichen Testverfahren ist deshalb immer erforderlich, um Fehlerwirkungen aufzudecken, die unterschiedliche Ursachen haben.

*Test auf Funktionalität, die nicht gefordert ist*

Während der Testphase ist zu prüfen, ob das Testobjekt so funktioniert und arbeitet, wie es in der →Spezifikation festgelegt ist. Auch wichtig – und damit selbstverständlich im Test zu überprüfen – ist, dass das Testobjekt keine über die Anforderungen hinausgehenden Funktionen ausführt, also nur die Funktionen zur Verfügung stellt, die gefordert sind.

*Testfallexplosion*

Der Testaufwand kann sehr umfangreich werden. Testmanager stehen vor dem Dilemma, dass sich die möglichen Testfälle und Testfallvarianten schnell zu einigen hundert oder tausend Tests, die durchzuführen sind, addieren. Das Problem wird auch mit dem Begriff

→Testfallexplosion umschrieben. Neben der notwendigen Beschränkung der Anzahl der Testfälle, hat ein Testmanager in der Regel noch mit einem weiteren Problem zu kämpfen, dem Mangel an Ressourcen.

*Beschränkte Ressourcen*

In jedem Softwareentwicklungsprojekt entbrennt früher oder später ein Kampf um Ressourcen. Die Komplexität der Entwicklungsaufgabe wird unterschätzt, das Projektteam gerät in Zeitverzug, der Kunde drängt, der Projektleiter möchte baldmöglichst »etwas« ausliefern. Der Testmanager hat in diesem »Spiel« die schlechtesten Karten: Oft gibt es nur ein kleines Zeitfenster kurz vor Auslieferung für die Durchführung der Testfälle und eine Hand voll Tester. Mit Sicherheit hat der Testmanager dann zu wenig Zeit und nicht genug Personal, um »astronomisch« viele Testfälle durchführen zu können.

In dieser Situation wird vom Testmanager dennoch erwartet, dass er verlässliche Resultate abliefert und sicherstellt, dass die Software ausreichend getestet wird. Nur wenn sich der Testmanager eine wohldurchdachte, effiziente Strategie zurechtgelegt hat, hat er die Chance, diese Herausforderung erfolgreich zu lösen. Voraussetzung hierzu ist das Vorhandensein eines fundamentalen Testprozesses. Neben der Einhaltung eines fundamentalen Testprozesses sind weitere qualitätssichernde Maßnahmen durchzuführen, wie beispielsweise →Reviews (s. Kap. 4.1). Im nächsten Kapitel wird ein grundlegender Testprozess beschrieben, wie er sinnvollerweise auch bei der Entwicklung und Prüfung des VSR-Systems anzuwenden ist.

## 2.2 Fundamentaler Testprozess

### Vorgehensmodelle der Softwareentwicklung

***Exkurs***

Um eine strukturierte und steuerbare Softwareentwicklung durchzuführen, werden →Softwareentwicklungsmodelle bzw. →Entwicklungsprozesse eingesetzt. Es gibt eine ganze Reihe von unterschiedlichen Modellen. Einige Beispiele sind das Wasserfall-Modell [Boehm 73, 81], das allgemeine →V-Modell[3] [Boehm 79] und das Vorgehensmodell des Bundes und der Länder [Versteegen 99]. Aktuell diskutiert werden der Rational Unified Process [Jacobson 99] und Extreme Programming [Beck 00] zur Entwicklung von objektorientierten Softwaresystemen.

Alle Modelle definieren eine Systematik zur geordneten Projektabwicklung. Meist werden Phasen bzw. Arbeitsabschnitte festgelegt, die mit einem bestimmten Ergebnis in Form eines Dokuments abzuschließen sind. Ein Phasenabschluss, oft auch als →Meilenstein bezeichnet, ist dann erreicht, wenn die geforderten Doku-

3. Das »V-Modell« wird mit dem Zusatz »allgemeines« versehen, um es nicht mit dem »Vorgehensmodell des Bundes und der Länder« zu verwechseln, da dieses Vorgehensmodell in der Literatur oft kurz als »V-Modell« bezeichnet wird.

mente fertig gestellt sind und den festgelegten Qualitätskriterien genügen. Meist werden →Rollen beschrieben, denen bestimmte anfallende Aufgaben in der Softwareentwicklung zugeordnet sind und die von an der Projektentwicklung beteiligten Personen auszufüllen sind. Teilweise werden bei den Modellen auch die einzusetzenden Methoden und Verfahren in den jeweiligen Phasen beschrieben. Mithilfe der Modelle kann eine detaillierte Projektplanung der Ressourcen (Zeit, Personal, Infrastruktur usw.) erfolgen. Die Entwicklungsmodelle definieren die für alle Beteiligten gemeinsame und verbindliche Sicht der auszuführenden Tätigkeiten und deren zeitliche Abfolge in einem Projekt.

Das Testen findet sich in jedem dieser →Vorgehensmodelle wieder, allerdings mit sehr unterschiedlicher Bedeutung und mit unterschiedlichem Umfang. Im Folgenden wird aus Sicht des Testens auf einige Modelle kurz eingegangen.

*Wasserfall-Modell: Testen als »Endprüfung«*

Das erste grundlegende Modell war das Wasserfall-Modell (s. Abb. 2-3, dargestellt mit den ursprünglich vorgesehenen Phasen [Royce 70], Abbildung nach [Balzert 98, S. 99]). Es ist bestechend einfach und hat einen hohen Bekanntheitsgrad erreicht. Erst wenn eine Entwicklungsphase abgeschlossen ist, wird mit der nächstfolgenden begonnen. Es gibt nur zwischen angrenzenden Stufen Rückkopplungsschleifen, die eine ggf. notwendige Überarbeitung in der vorherigen Stufe ermöglichen. Der entscheidende Nachteil ist, dass das Testen als »einmalige« Aktion am Projektende vor der Inbetriebnahme aufgefasst wird. Der Test wird als »Endprüfung« in Analogie einer Warenausgangsprüfung vor Übergabe des Produkts an den Kunden gesehen.

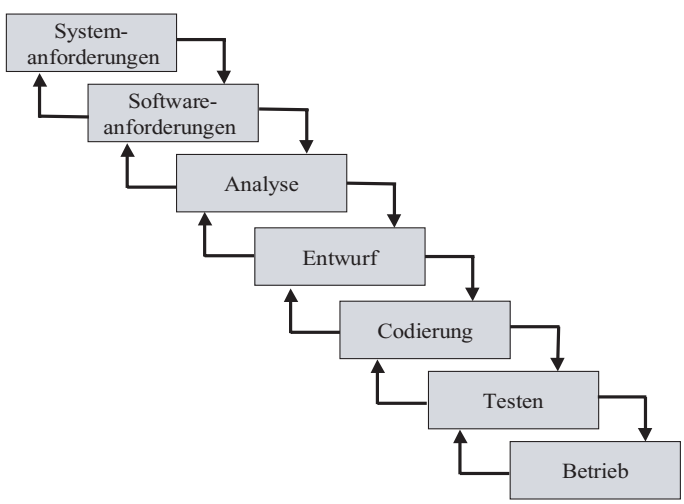



***Abb. 2–3***
*Wasserfall-Modell*

*Allgemeines V-Modell*

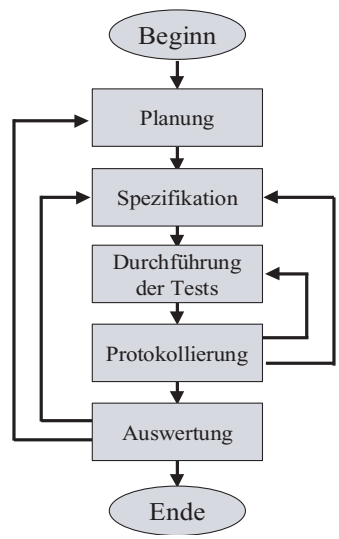
Eine Erweiterung des Wasserfall-Modells ist das allgemeine V-Modell ([Boehm 79]), bei dem die konstruktiven Aktivitäten von den prüfenden Aktivitäten getrennt sind (s. Kap. 3, Abb. 3-1). Das Modell hat die Form eines »V«s. Die konstruktiven Aktivitäten bis zur Implementierung sind auf dem absteigenden Ast des »V«s zu finden. Die testdurchführenden Aktivitäten auf dem aufsteigenden Ast sind in →Teststufen angeordnet, passend zu dem entsprechenden Abstraktionsniveau

der gegenüberliegenden konstruktiven Aktivität. Das allgemeine V-Modell hat eine sehr hohe Verbreitung gefunden und wird in der Praxis häufig eingesetzt.

*Vorgehensmodell des Bundes und der Länder*

Das Vorgehensmodell des Bundes und der Länder besteht aus den folgenden vier Teilmodellen: Systemerstellung, Qualitätssicherung, Konfigurationsmanagement und Projektmanagement. Das Modell definiert die Aktivitäten und Produkte sowohl des Entwicklungs- als auch des →Pflege- und →Wartungsprozesses. Auf unterschiedlichen Abstraktionsebenen werden die logischen Abhängigkeiten zwischen den Aktivitäten und den Produkten und die verschiedenen Zustände der Produkte sehr ausführlich beschrieben. Detaillierte Beschreibungen des Vorgehensmodells sind unter [URL: VM-IESE] und [URL: VM-scope] abrufbar.

***Abb. 2–4***
*Fundamentaler Testprozess in Anlehnung an [BS 7925-2]*




Als Anleitung, um in einem Softwareprojekt Tests strukturiert durchzuführen, reicht eine Darstellung der Aufgaben wie in den oben erläuterten Vorgehensmodellen nicht aus. Zusätzlich zur Einordnung des Testens in den gesamten Entwicklungsablauf wird ein verfeinerter Ablaufplan für die Testarbeiten selbst benötigt (s. Abb. 2-4). Das heißt, der »Inhalt« der Entwicklungsaufgabe »Testen« muss in kleinere Arbeitsabschnitte gegliedert werden, nämlich in: →Testplanung, →Testspezifikation, →Testdurchführung, →Testprotokollierung und Testauswertung. Diese Teilaufgaben bilden einen fundamentalen Testprozess und werden im Folgenden näher beschrieben.

### 2.2.1 Testplanung

Eine strukturierte Abwicklung einer so umfangreichen Aufgabe wie dem Testen kann nicht planlos erfolgen. Mit der Planung des Testprozesses wird zum Beginn des Softwareentwicklungsprojekts begonnen. Wie bei jeder Planung sind während des Projektfortschritts regelmäßig eine Kontrolle der bisherigen Planung, eine Aktualisierung und Anpassung vorzunehmen.

*Planung der Ressourcen*

Für den Testprozess sind die benötigten Ressourcen einzuplanen. Wann werden welche Mitarbeiter zur Durchführung der Aufgaben benötigt, wie viel Zeit ist dafür zu veranschlagen und welche Hilfsmittel und Geräte sind dafür bereitzustellen? Diese Fragen sind während der Planung zu beantworten, und die entsprechenden Festlegungen sind im →Testkonzept (engl. *test plan*, s. Kap. 3.9) vorzunehmen. Mögliche notwendige Schulungen der Mitarbeiter sind vorzusehen. Eine Organisationsstruktur mit dem entsprechenden Testmanagement ist ggf. aufzubauen bzw. anzupassen. Zu den Aufgaben des Testmanagements gehört die Verwaltung des Testprozesses, der →Testinfrastruktur und der →Testware. Die Überwachung des Fortschritts im Testprozess kann auf Grundlage der entsprechenden Berichterstattung der Mitarbeiter erfolgen. Vereinbarungen zu diesen Punkten müssen frühzeitig erfolgen.

*Festlegung der Teststrategie*

Kernaufgabe der Planung ist die Bestimmung der →Teststrategie. Da ein vollständiger Test nicht möglich ist, müssen Prioritäten anhand einer Risikoeinschätzung gesetzt werden. Je nach zu erwartendem Risiko und der Schwere der Auswirkungen beim Eintreten einer Fehlerwirkung sind die Testaktivitäten auf die einzelnen Systemteile zu verteilen. Kritische Systemteile müssen eine erhöhte Aufmerksamkeit im Test erfahren, also intensiver getestet werden. Bei den weniger kritischen Teilen reicht ein weniger umfangreicher Test aus. Es kann sogar auf den Test von Teilen verzichtet werden, wenn keine negativen Auswirkungen beim Eintreten einer Fehlerwirkung zu erwarten sind. Allerdings muss diese Entscheidung sehr fundiert getroffen werden. Ziel der Teststrategie ist die optimale Verteilung der Tests auf die »richtigen« Stellen des Softwaresystems.

---

***Beispiel einer Teststrategie***

Das VSR-System besteht aus den folgenden Teilsystemen:

- *DreamCar* ermöglicht die individuelle Zusammenstellung eines Fahrzeugs und seiner Ausstattung
- *ContractBase* verwaltet sämtliche Kundeninformationen und Vertragsdaten
- *JustInTime* bietet die Möglichkeit, eine Online-Bestellung vorzunehmen (in der ersten Ausbaustufe durch den Händler)

– *EasyFinance* berechnet eine für den Kunden optimale Finanzierung
– *NoRisk* bietet die Möglichkeit zum Abschluss einer passenden Versicherung

Sinnvollerweise sind nicht alle fünf Teilsysteme gleich intensiv zu testen. In Absprache mit dem Auftraggeber wird festgelegt, dass bei einem fehlerhaften Verhalten der Teilsysteme *DreamCar* und *ContractBase* die größten schädlichen Auswirkungen zu erwarten sind. Demzufolge wird in der Teststrategie festgelegt, dass diese beiden Teilsysteme intensiver zu testen sind.

Die Möglichkeit der Online-Bestellung durch das Teilsystem *JustInTime* wird als weniger kritisch angesehen, da der Bestellvorgang notfalls noch auf anderen Wegen (z. B. per Fax) weitergereicht werden kann. Wichtig ist allerdings, dass die Daten des Bestellvorgangs im Teilsystem *JustInTime* weder verfälscht werden noch verloren gehen dürfen. Dieser Aspekt ist deshalb wieder ausführlicher zu testen.

Für die beiden anderen Teilsysteme *NoRisk* und *EasyFinance* wird in der Teststrategie festgelegt, dass alle ihre Hauptfunktionen (Tarif kalkulieren, Vertrag erfassen und abschließen, Vertrag speichern und drucken usw.) zu testen sind. Dabei ist es allerdings aus Zeitgründen nicht möglich, alle denkbaren Vertragsvarianten für die Finanzierung und Versicherung eines Fahrzeugs zu berücksichtigen. Daher wird beschlossen, den Test auf die am häufigsten vorkommenden Tarifkonstellationen zu konzentrieren. Seltenere Konstellationen erhalten eine niedrige Priorität (s. Kap. 2.4).

Die ersten Überlegungen zur Teststrategie des VSR-Systems machen deutlich, dass eine Festlegung der Intensität des Testens sowohl für ganze Teilsysteme als auch für einzelne Aspekte eines Systems sinnvoll ist.

---

*Testintensität für Teilsysteme und einzelne Aspekte festlegen*

Die Intensität des Tests hängt stark ab von den eingesetzten →Testmethoden und dem bei der Ausführung zu erreichenden →Überdeckungsgrad. Der Überdeckungsgrad dient als Kriterium, um ein Ende des Tests festzulegen. Neben den Überdeckungskriterien, die sich auf den Programmtext beziehen (zum Beispiel auf Anweisungen oder Verzweigungen, s. Kap. 5.2), kann die Erfüllung der Anforderungen des Kunden als Endekriterium festgelegt werden. Es kann gefordert sein, dass alle Funktionen mindestens einmal getestet werden oder beispielsweise 70 % der möglichen Transaktionen in einem System einmal zur Ausführung kommen. Selbstverständlich sind auch die Festlegungen der Endekriterien und damit die Intensität der Tests unter Berücksichtigung des im Fehlerfall zu erwartenden Risikos zu bestimmen.

*Priorisierung der Tests*

Da Softwareprojekte oft unter hohem Zeitdruck abgewickelt werden, ist es sinnvoll, den Zeitaspekt in der Planung entsprechend zu berücksichtigen. Eine Priorisierung der durchzuführenden Tests garantiert, dass die kritischen Softwareteile zuerst getestet werden, falls aus Zeitgründen nicht alle geplanten Tests durchgeführt werden können (s. Kap. 2.4).

*Werkzeugunterstützung*

Soweit die notwendige Werkzeugunterstützung (s. Kap. 7) nicht vorhanden ist, ist eine Auswahl und Beschaffung der Werkzeuge frühzeitig einzuleiten. Bei vorhandenen Werkzeugen ist deren Aktualität zu prüfen. Sind Teile der Testinfrastruktur selbst zu realisieren, können die vorbereitenden Aufgaben durchgeführt werden. →Testrahmen (engl. *test bed*), in denen Systemteile zur Ausführung gebracht werden können, sind meist selbst zu programmieren. Deren Erstellung ist rechtzeitig durchzuführen, damit sie nach Abschluss der Programmierung der jeweiligen Testobjekte zur Verfügung stehen.

### 2.2.2 Testspezifikation

Die im Testplan festgelegte Teststrategie bestimmt, welche Testmethoden zum Einsatz kommen sollen. In der Testspezifikation werden dann unter Anwendung dieser Testmethoden die entsprechenden Testfälle spezifiziert.

*Logische und konkrete Testfälle*

Die Spezifikation der Testfälle erfolgt in zwei Schritten. Zuerst sind →logische Testfälle zu definieren. Danach können die logischen Testfälle konkretisiert werden, d. h. die tatsächlichen Eingabewerte festgelegt werden (→konkrete Testfälle). Auch der umgekehrte Weg ist möglich: vom konkreten zum allgemeinen, logischen Testfall. Dieses Vorgehen muss dann angewendet werden, wenn ein Testobjekt unzureichend spezifiziert ist und eher ein experimentelles Vorgehen einzuschlagen ist.

Die Auswahl der logischen Testfälle zu jeder Testmethode erfolgt auf Grundlage der jeweiligen →Testbasis. Die Testfälle können aus der Spezifikation der Testobjekte ermittelt (→Black-box-Verfahren) oder auf Grundlage des Programmtexts erstellt werden (→White-box-Verfahren). Es wird deutlich, dass die Spezifikation der Testfälle zu ganz unterschiedlichen Zeitpunkten im Softwareentwicklungsprozess erfolgen kann. Es hängt von den in der Teststrategie ausgewählten Testverfahren ab. Die zu Beginn des Kapitels 2.2 dargestellten Vorgehensmodelle zeigen in diesem Sinne nur Testdurchführungsphasen. Testplanungs- und Spezifikationsarbeiten können und sollten parallel zu früheren Entwicklungsaktivitäten erfolgen.

*Testfälle umfassen mehr als nur die Testdaten*

Für jeden Testfall muss die Ausgangssituation (Vorbedingung) beschrieben werden. Es muss klar sein, welche Randbedingungen für den Test gelten und einzuhalten sind. Ferner ist vor der Testdurchführung festzulegen, welche Ergebnisse bzw. welches Verhalten erwartet wird. Testfälle lassen sich nach zwei Kriterien unterscheiden:

- Testfälle zur Prüfung der spezifizierten und vom Testobjekt zu liefernden Ergebnisse und Reaktionen. Dabei sind auch die Testfälle

zu berücksichtigen, die die spezifizierte Behandlung von Ausnahme- und Fehlersituationen überprüfen (→Negativtest). Allerdings ist es oft schwierig, die für die Ausführung der Testfälle notwendigen Randbedingungen herzustellen (z. B. die Überlastung einer Netzverbindung).

- Testfälle zur Prüfung der Reaktion des Testobjekts auf ungültige und unerwartete Eingaben bzw. Randbedingungen, für die keine →Ausnahmebehandlungen (engl. *exception handling*) spezifiziert wurden.

***Beispiel für Testfälle***

Das folgende Beispiel soll die Unterschiede zwischen logischen und konkreten Testfällen verdeutlichen.

Eine Firma hat ein Programm in Auftrag gegeben, das die Weihnachtsgratifikation der Mitarbeiter in Abhängigkeit von der Firmenzugehörigkeit berechnen soll. In der Beschreibung der Anforderungen findet sich folgende Textpassage: »Mitarbeiter erhalten ab einer Zugehörigkeit zur Firma von mehr als drei Jahren 50 % des Monatsgehalts als Weihnachtsgratifikation. Mitarbeiter, die länger als fünf Jahre in der Firma tätig sind, erhalten 75 %. Bei einer Firmenzugehörigkeit von mehr als acht Jahren wird eine Gratifikation von 100 % gewährt.«

Aus dem Text lassen sich folgende Zusammenhänge für die Gewährung der Gratifikation in Abhängigkeit der Firmenzugehörigkeit aufstellen:

| | |
|---|---|
| Firmenzugehörigkeit <= 3 | ergibt Gratifikation = 0 % |
| 3 < Firmenzugehörigkeit <= 5 | ergibt Gratifikation = 50 % |
| 5 < Firmenzugehörigkeit <= 8 | ergibt Gratifikation = 75 % |
| Firmenzugehörigkeit > 8 | ergibt Gratifikation = 100 % |

Aus dieser formaleren Angabe lassen sich folgende logische Testfälle erstellen (s. Tab. 2-1).

***Tab. 2–1***
*Tabelle mit logischen Testfällen*

| logischer Testfall | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| Eingabewert x (Firmenzugehörigkeit) | x <= 3 | 3<x<=5 | 5<x<=8 | x>8 |
| Erwartetes Ergebnis (Gratifikation in %) | 0 | 50 | 75 | 100 |

Zum Ausführen der Testfälle müssen die logischen Testfälle in konkrete Testfälle umgewandelt werden, d. h., es müssen ganz konkrete Eingabewerte festgelegt werden (s. Tab. 2-2). Besondere Ausgangssituationen oder Randbedingungen sind für diese Testfälle nicht gegeben.

| konkreter Testfall | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| Eingabewert x (Firmenzugehörigkeit) | 2 | 4 | 7 | 12 |
| Erwartetes Ergebnis (Gratifikation in %) | 0 | 50 | 75 | 100 |

***Tab. 2–2*** *Tabelle mit konkreten Testfällen*

Die hier gewählten Werte sollen lediglich als Beispiel für die Unterschiede zwischen logischen und konkreten Testfällen dienen. Es ist keine explizite Testmethode zur Erstellung der Testfälle verwendet worden, und es wird auch nicht der Anspruch erhoben, dass die vier Testfälle das Programmstück, das die Berechnung der Gratifikation vornimmt, ausreichend genug prüfen. So sind beispielsweise keine Testfälle für die Behandlung von falschen Eingaben (z. B. eine negative Firmenzugehörigkeit) berücksichtigt worden. Genauere Angaben zur systematischen Erstellung der Testfälle mit den entsprechenden Testmethoden sind in Kapitel 5 zu finden.

*Konkrete Testfälle*

Wenn der Testprozess vorangeschritten ist und mehr über die technische Durchführung bekannt ist, werden die logischen Testfälle konkretisiert. Die konkreten Testfälle können dann ohne weitere Änderungen oder Ergänzungen während der Testdurchführung verwendet werden.

Parallel zu den beschriebenen Aktivitäten der Spezifikation der Testfälle kann und sollte beurteilt und festgelegt werden, welche Testinfrastruktur benötigt wird, um das Programm unter Benutzung der spezifizierten Testfälle (genauer: deren Vorbedingungen und Eingabedaten) ablaufen zu lassen (s. u.). Damit es bei der Durchführung der Tests nicht zu zeitlichen Verzögerungen kommt, sollte die Testinfrastruktur bereits zu diesem Zeitpunkt soweit wie möglich realisiert und aufgebaut werden.

### 2.2.3 Testdurchführung

Da alle vorbereitenden Aufgaben für den Test abgeschlossen sind, kann direkt nach der Programmierung und der Übergabe der zu testenden Systemteile an die Tester mit der Testdurchführung begonnen werden.

*Prüfung auf Vollständigkeit*

Zunächst erfolgt eine Prüfung auf Vollständigkeit der zu testenden Teile. Hierzu wird das Testobjekt in der bereitstehenden Testumgebung installiert und auf prinzipielle Start- und Ablauffähigkeit überprüft.

*Prüfung der Hauptfunktionen*

Es empfiehlt sich, die Testdurchführung mit der Überprüfung der Hauptfunktionen des Testobjekts zu beginnen. Sollten sich hier bereits Fehlerwirkungen oder Abweichungen von den Sollergebnissen zeigen, ist ein tiefes Einsteigen in den Test wenig sinnvoll, da vorab die fehlerhaften Hauptfunktionen korrigiert werden sollten. Ein solches Vorgehen muss in der Teststrategie festgelegt sein.

### 2.2.4 Testprotokollierung

*Tests ohne Protokollierung sind wertlos*

Die Durchführung der Tests ist exakt und vollständig zu protokollieren. Zum einen muss anhand dieser →Testprotokolle die Testdurchführung auch für nicht direkt beteiligte Personen, zum Beispiel für den Kunden, nachvollziehbar sein. Zum anderen muss nachweisbar sein, ob die geplante Teststrategie tatsächlich umgesetzt wurde. Aus dem Testprotokoll muss hervorgehen, welche Teile wann, von wem, wie intensiv und mit welchem Ergebnis getestet wurden.

Zu jeder Testdurchführung gehören neben dem Testobjekt eine ganze Reihe von Informationen und Dokumenten: Testrahmen, Eingabedateien, Testprotokoll usw. Die zu einem Testfall bzw. Testlauf gehörenden Informationen und Daten sind so zu verwalten, dass zu einem späteren Zeitpunkt eine Wiederholung der Tests mit den gleichen Daten und Randbedingungen problemlos möglich ist. Ein →Konfigurationsmanagement der Testware ist vorzunehmen.

### 2.2.5 Auswertung und Bewertung des Testendes

*Fehler gefunden?*

Tritt bei der Testausführung ein Unterschied zwischen dem tatsächlichen und dem erwarteten Ergebnis auf, ist bei der Auswertung der Testprotokolle zu entscheiden, ob tatsächlich eine Fehlerwirkung vorliegt. Die Ursache der Nichtübereinstimmung mit dem Sollergebnis kann nämlich auch in der fehlerhaften oder ungenauen Testspezifikation, in einer fehlerhaften Testinfrastruktur oder in einem fehlerhaften Testfall oder Testlauf liegen. Welche der Möglichkeiten zutrifft, muss sorgsam geprüft werden. Nichts ist für die Glaubwürdigkeit eines Testers schädlicher als eine gemeldete vermeintliche Fehlerwirkung, deren Ursache aber in einem fehlerhaften Testfall liegt. Befürchtungen in dieser Richtung sollen aber nicht dazu führen, dass vorsichtshalber potenzielle Fehlerwirkungen nicht gemeldet werden. Das könnte genauso fatal sein.

Anhand der →Fehlerklasse (s. Kap. 6.4.3) ist zu entscheiden, mit welcher Priorität der Fehler zu beheben ist. Nach der Korrektur des Fehlerzustands ist zu prüfen, ob die Fehlerwirkung beseitigt ist und bei der Beseitigung keine weiteren Fehlerzustände hinzugekommen sind (s. Kap. 3.6.3). Gegebenenfalls sind neue Testfälle zu spezifizieren, die den geänderten bzw. neuen Programmtext überprüfen. Es wäre günstig, Fehlerzustände einzeln zu korrigieren und erneut zu testen, um unbeabsichtigte gegenseitige Beeinflussungen der Änderungen zu vermeiden. In der Praxis wird sich dies allerdings nicht durchführen lassen. Wird der Test nicht vom Entwickler selbst ausgeführt, sondern von unabhängigen Testern, ist eine separate Korrektur des Fehlerzustände

nicht praktikabel. Der Aufwand, jede Fehlerwirkung einzeln dem Entwickler zu melden und erst dann weiterzutesten, nachdem der gemeldete Fehler korrigiert wurde, ist nicht gerechtfertigt. Hier werden mehrere Fehlerzustände korrigiert und dann mit einem neuen Softwareversionsstand zum erneuten Test wieder vorgelegt.

*Testende erreicht?*

Es ist zu entscheiden, ob das im Testkonzept festgelegte Kriterium zur Beendigung der Tests erfüllt ist. Für jede verwendete Testmethode ist unter Berücksichtigung des Risikos ein angemessenes Endekriterium zu bestimmen. Es kann beispielsweise spezifiziert werden, dass ein Test dann als ausreichend angesehen wird, wenn 80 % aller Anweisungen des Testobjekts bei der Testdurchführung zur Ausführung gekommen sind. Dies wäre allerdings keine sehr hohe Anforderung an den Test. Die Ermittlung solcher →Metriken (oder Maße) zur Entscheidung der Beendigung des Tests sollten von entsprechenden Testwerkzeugen vorgenommen werden (s. Kap. 7.1.4).

Ist mindestens ein Testendekriterum nach Durchführung aller Tests nicht erfüllt, müssen weitere Tests zur Ausführung kommen. Dabei ist sehr darauf zu achten, dass die zusätzlichen Tests zu einer weiteren Annäherung an das Endekriterium führen. Andere Testfälle bringen nur zusätzlichen Aufwand, aber keinen Fortschritt im Hinblick auf das Testende.

*Weiterer Aufwand gerechtfertigt?*

Eine nähere Analyse des Problems kann auch ergeben, dass der notwendige Aufwand zur Erfüllung des Endekriteriums nicht im angemessenen Verhältnis zum möglichen Ergebnis steht und dass auf weitere Tests verzichtet wird. Eine solche Entscheidung muss selbstverständlich das damit verbundene Risiko berücksichtigen.

Ein Beispiel für einen solchen Fall kann die Behandlung einer Ausnahmesituation sein. Mit der zur Verfügung stehenden Testumgebung kann evtl. die Ausnahmesituation nicht ausgelöst bzw. simuliert und somit der entsprechende Programmtext zur Behandlung der Ausnahmesituation auch nicht ausgeführt und getestet werden. In einem solchen Fall sind dann andere Prüfverfahren einzusetzen, zum Beispiel statische Analysen (s. Kap. 4).

*Unerreichbarer Programmcode*

Ein weiterer Fall der Nichterfüllung von Testendekriterien kann eintreten, wenn das spezifizierte Kriterium im konkreten Fall überhaupt nicht erfüllbar ist. Wenn beispielsweise →tote Programmanweisungen (engl. *dead code*) im Testobjekt enthalten sind, können diese nicht zur Ausführung gebracht werden. Eine 100%ige Überdeckung aller Anweisungen (und dazu würden auch die toten, nicht erreichbaren Anweisungen gehören) ist somit nie erfüllbar. Auch diese Möglichkeit muss berücksichtigt werden, um nicht weitere in Bezug auf das Kriterium nutzlose Tests durchzuführen. Ein nicht erfüllbares Krite-

rium ist meist ein Hinweis auf mögliche Ungereimtheiten oder Ungenauigkeiten in den Anforderungen oder der Spezifikation. In dem gegebenen Beispiel macht es sicherlich Sinn zu ermitteln, warum Anweisungen im Programm enthalten sind, die nicht zur Ausführung gebracht werden können. Weitere Fehlerzustände können so gefunden bzw. deren Wirkungen vorab vermieden werden.

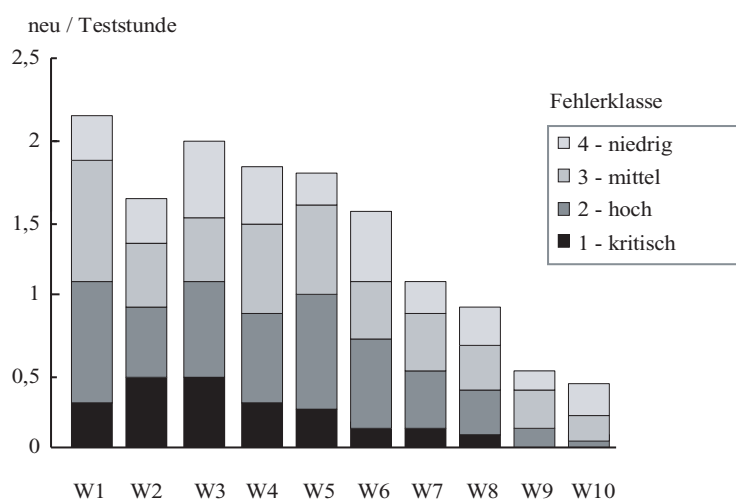
Sind zusätzliche Tests durchzuführen, ist der Testprozess wieder aufzunehmen und zu entscheiden, an welcher Stelle der erneute Einstieg erfolgen kann. Es kann sogar notwendig sein, die Planung zu überarbeiten, da zusätzliche Ressourcen erforderlich sind. Genauso kann es sein, dass die Testspezifikation zu ergänzen ist, um das geforderte Endekriterium zu erfüllen.

*Weitere Kriterien für die Bestimmung des Testendes*

Neben den Testüberdeckungskriterien können weitere Kriterien für die Bestimmung des Testendes verwendet werden. Ein mögliches Kriterium kann die Bestimmung der →Fehlerfindungsrate sein. In Abb. 2-5 ist die mittlere Anzahl von neuen, also bisher nicht aufgedeckten Fehlerwirkungen pro Teststunde über zehn Wochen verteilt dargestellt. In der ersten Woche wurden im Mittel pro Teststunde mehr als zwei neue Fehlerwirkungen nachgewiesen. In der zehnten Woche ist es knapp ein halber Fehler. Sinkt die Fehlerfindungsrate unter ein vorgegebenes Maß (z. B. Nachweis von weniger als einer Fehlerwirkung pro Teststunde), dann wird davon ausgegangen, dass das Verhältnis von Testaufwand und Nachweis von Fehlerwirkungen nicht mehr ökonomisch ist und der Test beendet werden kann.

Hierbei ist allerdings zu beachten, dass Fehler sehr unterschiedliche Auswirkungen haben können. Eine Einteilung und Unterscheidung der

***Abb. 2–5***
*Fehlerfindungsrate*



Fehlerwirkungen nach der Schwere ihrer Auswirkungen ist deshalb sinnvoll und generell zu berücksichtigen (s. Kap. 6.4.3).

*Mehrere Testzyklen berücksichtigen*

Die beim Test nachgewiesenen Fehlerwirkungen müssen je nach Fehlerklasse beseitigt werden, und ein erneuter Test wird notwendig. Oft entstehen auch Zyklen, wenn beim Test von Änderungen weitere Fehlerwirkungen nachgewiesen werden, die es zu beseitigen und dann erneut zu testen gilt. Keine solchen Korrekturzyklen einzuplanen ist unrealistisch, da dann davon ausgegangen wird, dass keine Fehlerwirkungen im Test nachgewiesen werden, die zu beheben und in einem weiteren →Testzyklus zu testen sind. Projektverzögerungen sind dann die Regel. Der benötigte Aufwand der Fehlerfindung und der folgenden Zyklen lässt sich nur schwer im Voraus kalkulieren. In der Projektplanung müssen entsprechende Zeitpuffer und Personalressourcen vorgesehen werden.

*Endekriterien der Praxis: Zeit und Kosten*

In der Praxis wird das Ende des Tests häufig durch Faktoren bestimmt, die nicht im direkten Zusammenhang mit dem Test stehen: Zeit und Kosten. Führen diese zum Abbruch der Testaktivitäten, so sind bei der Projektplanung nicht ausreichend Ressourcen zur Verfügung gestellt worden oder der Aufwand für einen angemessenen Test wurde unterschätzt.

*Erfolgreiches Testen spart Kosten ein!*

Auch wenn beim Testen mehr Mittel als geplant verbraucht werden, so kommt es doch im Endeffekt zu Einsparungen durch die Beseitigung der Fehlerzustände in der Software. Fehlerzustände, die nicht entdeckt werden, verursachen im Betrieb nämlich meist erheblich höhere Kosten (s. Kap. 3.8).

*Abschließende Arbeiten*

Sind die Testendekriterien erfüllt bzw. eine Klärung des Sachverhalts bei deren Nichterfüllung gegeben, sind noch die folgenden abschließenden bzw. vorbereitenden Aufgaben durchzuführen.

Zunächst sollte sozusagen eine Konservierung der Testware für die Zukunft vorgenommen werden. Softwaresysteme werden über einen längeren Zeitraum eingesetzt. In dieser Zeit werden Fehlerwirkungen auftreten, die im Test nicht gefunden wurden, oder es gibt Änderungswünsche des Kunden. Beides führt zu einer Überarbeitung des Programms, und der geänderte Programmtext muss in jedem Fall erneut getestet werden. Ein erheblicher Teil dieses Testaufwands in der Wartungsphase kann vermieden werden, wenn die Testware (Testfälle, Testprotokolle, Testinfrastruktur, eingesetzte Werkzeuge …) weiterhin zur Verfügung steht. In der Wartung muss dann nur eine Anpassung und keine neue Erstellung der Testware vorgenommen werden. Die Testware lässt sich auch für Projekte mit ähnlicher Aufgabenstellung nach einer entsprechenden Anpassung Gewinn bringend verwenden.

*Evaluation des Testprozesses*

Eine weitere aus Zeitgründen leider oft vernachlässigte Aktivität ist die Evaluation des Testprozesses. Ein kritischer Rückblick auf die durchgeführten Tätigkeiten im Testprozess unter Berücksichtigung des Aufwands und der erzielten Ergebnisse bringt sicherlich Verbesserungspotenzial zutage. Werden diese Erkenntnisse für die nachfolgenden Projekte genutzt, ist eine kontinuierliche Prozessverbesserung gegeben. Ausführliche Hinweise zur Bewertung des Testprozesses sind in [Pol 00] nachzulesen.

## 2.3 Sollwerte und Testorakel

Nach jedem durchgeführten Testfall muss entschieden werden, ob eine Fehlerwirkung vorliegt oder nicht. Um das entscheiden zu können, muss das beobachtete Ergebnis (Istergebnis/Istverhalten) mit dem erwarteten Ergebnis (Sollergebnis/Sollverhalten) verglichen werden. Das Sollverhalten des Testobjekts für jeden Testfall muss dazu vorab bestimmt werden. Diese Information muss sich der Tester bei der Spezifikation eines Testfalls aus geeigneten Quellen beschaffen. In diesem Zusammenhang wird auch von einem Orakel oder →Testorakel gesprochen, das befragt werden muss und die jeweiligen Sollergebnisse vorhersagt.

*Ermittlung der Sollwerte*

Nach [Riedemann 97] sind die Sollwerte aus der Spezifikation abzuleiten. Er unterscheidet dabei die drei folgenden Möglichkeiten:

- Der Tester leitet das Solldatum aus dem Eingabedatum auf Grundlage der Spezifikation des Testobjekts ab. Dies ist das in der Praxis übliche Vorgehen.
- Liegt eine formale Spezifikation vor, kann daraus ein ausführbarer Prototyp werkzeuggestützt erzeugt werden. Die Ergebnisse des Prototypen können dann als Testorakel für den Test des eigentlichen Programms dienen.
- Das Softwareprogramm wird mehrfach von unabhängigen Entwicklungsgruppen parallel erstellt. Jede Programmversion wird dann mit den gleichen Testdaten gegen eine andere getestet. Bei unterschiedlichen Ergebnissen der Programmversionen ist ein Fehlerzustand in einer Version vorhanden. Nur diejenigen Fehlerzustände, die in allen Versionen mit den gleichen Fehlerwirkungen vorhanden sind, bleiben unentdeckt. Dieses Vorgehen wird als →*Back-to-back*-Test bezeichnet.

Die beiden letzten Möglichkeiten der Ermittlung des Testorakels werden wegen der relativ geringen Verbreitung von formalen Methoden in

der Praxis bzw. wegen des großen Aufwands selten oder nur bei sehr hohen Qualitätsanforderungen verwendet. Ein Back-to-back-Test wird allerdings häufig zur Prüfung neuer Versionen von Programmen eingesetzt, also nicht bei reinen Neuentwicklungen. Im Back-to-back-Test ist dann nachzuweisen, dass die alte Version und die neue Version in den nicht geänderten Teilen übereinstimmen.

*Weitere Orakel*

Nach [Sneed 02] kommen als Orakel folgende weitere Möglichkeiten in Betracht: Benutzungshandbuch, Diagramme und Modelle sowie Nutzungsprofile.

Aus einer vollständigen und gut strukturierten Benutzungsdokumentation können Testfälle und die erwarteten Ergebnisse abgeleitet werden. Alle Eingabemöglichkeiten und die entsprechenden Funktionen und Reaktionen des Programms, z. B. auch auf fehlerhafte Eingaben durch den Benutzer, sind in der Dokumentation beschrieben. Aus den Informationen lassen sich dann die Testfälle mit den Eingabewerten und die erwarteten Ausgabewerte bzw. Systemreaktionen ableiten.

Werden Diagramme oder Modelle (z. B. Anwendungsfalldiagramm oder Aktivitätsdiagramm der Unified Modeling Language) während der Entwicklung eingesetzt, so können auch diese als Grundlage für die Testüberlegungen dienen. Um direkt Testfälle aus den Diagrammen und Modellen abzuleiten, sind diese meist nicht detailliert genug. Der Tester muss bei der manuellen Erstellung der Testfälle noch Ergänzungen vornehmen. Die konkreten Ein- und Ausgabewerte sind festzulegen.

Sind weder eine umfassende Benutzungsdokumentation noch eine detaillierte Spezifikation oder Diagramme verfügbar, so muss das System selbst als Testorakel dienen. Ein Nutzungsprofil wird erstellt, das heißt, das System wird in einer oder mehreren Nutzungsumgebungen nach Möglichkeit von unterschiedlichen Personen benutzt. Es werden dabei Daten gesammelt und aufgezeichnet, die als Testdaten Verwendung finden. Diese Daten können durch weitere Testfälle ergänzt werden. Diese Möglichkeit des Testorakels ist sicherlich die schlechteste, da das Testobjekt selbst Grundlage ist und eine Kernforderung, die Festlegung der Sollwerte vor der Testausführung, nicht erfüllt ist.

*Werkzeuggestützter Vergleich*

Der Vergleich der Sollwerte mit den im Test tatsächlich ermittelten Istwerten sollte auf jeden Fall von einem Werkzeug vorgenommen werden. Dabei ist allerdings darauf zu achten, dass nicht die Werte des ersten Testlaufs als Vergleichswerte herangezogen werden. Falsche Werte im ersten Testlauf dienen sonst als vermeintlich richtige Kontrollwerte für die nachfolgenden Tests.

Nicht immer lassen sich die konkreten Sollwerte exakt vorhersagen oder ermitteln. In diesen Fällen muss das vom Programm gelieferte Er-

gebnis innerhalb eines Toleranzbereichs liegen oder zumindest plausibel sein. Die Kontrolle der Werte kann dann nicht ausschließlich einem Werkzeug überlassen werden.

*Erfahrung des Testers ist wichtig*

Die Erfahrung des Testers spielt eine nicht zu unterschätzende Rolle bei der Erstellung der Testfälle und der Festlegung der Sollergebnisse. Meistens ist auch entsprechendes Fachwissen aus dem Anwendungsbereich notwendig, um zu erfolgreichen Testfällen zu kommen und möglichst viele Fehlerwirkungen aufzudecken. Hier ist die kollegiale Zusammenarbeit von Testern und Kunden gefordert.

## 2.4 Priorisierung des Tests

In vielen Projekten ist trotz ausreichender Planung die Zeit zur Durchführung aller spezifizierten Testfälle nicht vorhanden. Es ist dann eine sinnvolle Auswahl der Testfälle zu treffen, um sicherzustellen, dass trotz der eingeschränkten Anzahl an durchzuführenden Testfällen noch möglichst viele kritische Fehlerwirkungen gefunden werden. Hierzu ist eine Priorisierung der Testfälle erforderlich. Die Priorisierung soll so erfolgen, dass bei einer (vorzeitigen) Beendigung der Tests zu einem beliebigen Zeitpunkt das bis dahin bestmögliche Ergebnis erreicht wird.

*Die wichtigen Testfälle zuerst*

Eine Priorisierung hat darüber hinaus den Vorteil, dass die wichtigen Testfälle zuerst zur Ausführung kommen und schwer wiegende Fehlerwirkungen frühzeitig erkannt und behoben werden. Eine Gleichverteilung der beschränkten Testressourcen auf alle Testobjekte des Projekts nach dem Gießkannenprinzip ist wenig sinnvoll, da bei einem solchen Vorgehen kritische und unkritische Programmteile gleich intensiv getestet werden. Bei den kritischen Teilen ist dann nicht ausreichend genug getestet worden, bei den unkritischen Teilen sind die Ressourcen unnütz verschwendet worden.

Im Folgenden werden Kriterien für die Priorisierung und damit für die Festlegung der Reihenfolge der durchzuführenden Testfälle aufgeführt. Welches Kriterium herangezogen werden soll, hängt vom Projekt, dem Anwendungsbereich und den Kundenanforderungen ab. Eine allgemein gültige Festlegung oder Reihenfolge kann nicht gegeben werden.

*Kriterien der Priorisierung*

Kriterien für die Priorisierung der durchzuführenden Testfälle:

- Die Eintrittswahrscheinlichkeit einer Fehlerwirkung beim Betrieb der Software ist neben der Fehlerschwere ebenfalls zu berücksichtigen. Werden bestimmte Funktionen der Software oft genutzt, so ist, falls dort ein Fehlerzustand enthalten ist, eine hohe Wahr-

scheinlichkeit gegeben, dass dieser Fehlerzustand im Betrieb ausgelöst wird und zu einer Fehlerwirkung oder einem Ausfall führt. Testfälle zur Prüfung dieser Funktionen sollten daher mit einer höheren Priorität versehen werden als Tests von Funktionen mit einer sehr geringen Nutzungshäufigkeit.

- Die kombinierte Betrachtung von Fehlerschwere und Eintrittswahrscheinlichkeit führt zum Begriff des Fehlerrisikos. Das Risiko ergibt sich aus dem erwarteten Schaden im Fehlerfall multipliziert mit der Wahrscheinlichkeit des Eintretens der Fehlersituation. Solche Risiken können z. B. darin bestehen, dass der Geschäftsablauf des Kunden, der die Software einsetzt, durch einen Ausfall behindert wird und dies zu finanziellen Einbußen beim Kunden führt. Tests, die risikoreiche Fehlerwirkungen aufdecken könnten, erhalten höhere Priorität als Tests, die risikoarme Ausfälle aufdecken sollen.
- Die Wahrnehmung einer Fehlerwirkung durch den Endanwender ist ein weiteres Kriterium zur Priorisierung der Testfälle. Dies ist besonders bei interaktiven Systemen wichtig. Beispielsweise wird der Benutzer eines Stadtinformationssystems durch offensichtliche Fehlerwirkungen des Systems in der Bedienoberfläche verunsichert und vertraut dann auch der Gültigkeit der restlichen Informationen nicht mehr.
- Testfälle können in Abhängigkeit der Priorität der Anforderungen ausgewählt werden. Die einzelnen Funktionen, die ein System liefert, haben für den Kunden unterschiedliche Bedeutung. Er kann auf bestimmte Teile der Funktionalität möglicherweise verzichten, wenn sie fehlerhaft sind, bei anderen Teilen ist dies nicht möglich.
- Neben den Anforderungen sind auch die Qualitätsmerkmale zu berücksichtigen, die für den Kunden unterschiedliche Gewichtung haben. Die korrekte Umsetzung der Qualitätsmerkmale mit hoher Bedeutung für den Kunden muss geprüft werden. Testfälle zum Nachweis der Einhaltung der geforderten Qualitätsmerkmale erhalten eine hohe Priorität.
- Die Priorisierung der Testfälle kann auch aus Sicht der Entwicklung vorgenommen werden. Besonders intensiv sollen →Komponenten getestet werden, bei deren Versagen mit schwer wiegenden Auswirkungen, z. B. dem Ausfall des Systems, zu rechnen ist.
- Die Komplexität der einzelnen Komponenten und Systemteile kann zur Priorisierung der Testfälle herangezogen werden. Komplizierte Programmstücke sollten intensiver getestet werden, da Entwickler hier vermutlich häufiger fehlerhaft programmiert haben. Allerdings kann es auch vorkommen, das vermeintlich einfache Programmteile eine ganze Reihe von Fehlerzuständen enthalten, da die Ent-

wicklung dieser Programmteile nicht mit der erforderlichen Sorgfalt durchgeführt wurde. Liegen Erfahrungswerte aus früheren Projekten vor, kann eher entschieden werden, welche der beiden Möglichkeiten im konkreten Projektumfeld zutrifft.

- Fehlerwirkungen mit einem hohen Projektrisiko müssen frühzeitig erkannt werden. Gemeint sind Fehlerwirkungen, die eine aufwändige Überarbeitung erfordern, Ressourcen binden und eine erhebliche zeitliche Verzögerung des Projekts verursachen.

*Wo viele Fehler sind, finden sich meist noch mehr*

In Projekten tritt auch folgendes Phänomen häufig auf: Wo schon viele Fehlerzustände gefunden wurden, sind meist noch weitere vorhanden. Um auf solche Gegebenheiten reagieren zu können, muss es möglich sein, Prioritäten der Testfälle ändern zu können.

Ohne eine Priorisierung der Testfälle ist eine angemessene und erfolgreiche Verteilung der beschränkten Ressourcen nicht möglich. Eine Konzentration auf die wichtigen Testfälle ist unabdingbar.

## 2.5 Psychologie des Testens

*Errare humanum est*

Menschen machen Fehler, aber sie geben es nur sehr ungern zu! Ein Ziel des Testens von Software ist die Aufdeckung von Abweichungen gegenüber der Spezifikation bzw. den Kundenwünschen. Die gefundenen Fehlerwirkungen müssen den Entwicklern mitgeteilt werden. Wie mit den dabei auftretenden psychologischen Problemen umgegangen werden kann, wird in diesem Kapitel beschrieben.

Die Aufgaben der Entwicklung von Software werden oft als konstruktive und die der Prüfung der Dokumente und der Software als destruktive Tätigkeiten angesehen. Allein schon durch diese Sichtweise ergeben sich bei den beteiligten Personen meist Unterschiede in der Einstellung zu ihrer Tätigkeit. Diese Unterschiede sind allerdings nicht gerechtfertigt, denn »Testen ist eine extrem kreative und intellektuell herausfordernde Aufgabe« ([Myers 82, S.15]).

*Entwicklertest*

«Kann der Entwickler sein eigenes Programm testen?«, ist eine oft gestellte und entscheidende Frage. Eine allgemein gültige Antwort gibt es nicht. Ist der Tester auch der Autor des Programms, muss er seine eigene Arbeit kritisch prüfen. Nur wenige Menschen sind in der Lage, den dafür notwendigen Abstand zum selbst erschaffenen Produkt herzustellen. Wer weist schon gerne seine eigenen Fehler nach? Es besteht doch eher ein Interesse daran, möglichst keine Fehlerzustände im eigenen Programmtext zu finden.

Die große Schwäche von →Entwicklertests ist die, dass jeder Entwickler, der das von ihm selbst programmierte Programmstück testen

muss, seinem eigenen »Werk« immer zu optimistisch begegnet. Die Gefahr ist sehr groß, dass er sinnvolle Testfälle vergisst oder – weil er sich mehr für das Programmieren statt für das Testen interessiert – nur oberflächlich testet.

*Blindheit gegenüber eigenen Fehlern*

Falls er einen grundsätzlichen Designfehler eingebaut hat, z. B. weil er die Aufgabenstellung falsch verstanden hat, kann er das auch durch seine Tests nicht herausfinden. Der passende Testfall wird ihm überhaupt nicht in den Sinn kommen. Eine Möglichkeit, dieses Problem der »Blindheit gegenüber eigenen Fehlhandlungen« zu verringern ist, paarweise zusammenzuarbeiten und die eigenen Programme jeweils vom Entwicklerkollegen testen zu lassen.

Andererseits ist die gute Kenntnis des eigenen Testobjekts auch vorteilhaft. Eine Einarbeitung ist nicht notwendig und spart somit Zeit. Es ist vom Management abzuwägen, wann der Vorteil der Zeitersparnis verbunden mit dem Nachteil der Blindheit den eigenen Fehlhandlungen gegenüber zum Tragen kommen kann. Die Entscheidung muss in Abhängigkeit der Kritikalität des Testobjekts und dem damit verbundenen Risiko im Fehlerfall getroffen werden.

*Unabhängiges Testteam*

Förderlicher für die Qualität und Schärfe der Tests ist es, wenn ein unabhängiges Testteam die Tests durchführt. Der Tester kann dem Testobjekt unvoreingenommen gegenübertreten: Es ist nicht »sein« Produkt, und mögliche Annahmen und Missverständnisse des Entwicklers sind nicht zwangsläufig auch die Annahmen und Missverständnisse des Testers. Um Testfälle zu erstellen, muss der Tester sich aber das benötigte Wissen über das Testobjekt aneignen, was Zeit kostet. Allerdings bringt er vertieftes Test-Know-how mit, das ein Entwickler nicht hat bzw. sich erst aneignen muss (oder besser müsste, da dazu die notwendige Zeit oft nicht vorhanden ist).

*Mitteilung von Fehlern*

Es ist Aufgabe des Testers, die gefundenen Fehlerwirkungen und Abweichungen dem Autor und/oder dem Management mitzuteilen. Die Art und Weise, wie dies erfolgt, kann für die Zusammenarbeit zwischen Entwickler und Tester förderlich sein oder die wichtige Kommunikation zwischen den beiden Gruppen negativ beeinflussen. Anderen Fehlhandlungen nachzuweisen, ist keine leichte Aufgabe und erfordert ein gewisses Fingerspitzengefühl.

Oft besteht das Problem, dass im Test nachgewiesene Fehlerwirkungen von den Entwicklern in der Entwicklungsumgebung nicht reproduzierbar sind. Neben der genauen Beschreibung der Fehlerwirkung muss somit auch die Testumgebung ausführlich dokumentiert werden, damit Unterschiede in den Umgebungen, die Ursache für das unterschiedliche Verhalten sein können, nachvollziehbar sind.

Wann eine Fehlerwirkung oder Abweichung vorliegt und wann nicht, ist vorab festzulegen. Wenn es nicht eindeutig aus den Anforderungen oder der Spezifikation hervorgeht, ist der Kunde oder das Management gefordert, eine Entscheidung zu treffen. Eine Diskussion zwischen den beteiligten Personen, Entwickler und Tester, zum Thema »Fehler ja oder nein«, bringt keinen Fortschritt. Auch die oft gehörte Reaktion der Entwickler auf die aus ihrer Sicht vermeintliche Fehlerwirkung: »It's not a bug, it's a feature!«, ist wenig hilfreich.

*Gegenseitiges Verständnis*

Förderlich für die Zusammenarbeit zwischen Tester und Entwickler ist die gegenseitige Kenntnis der Aufgaben. Entwickler sollten die Grundlagen des Testens kennen, und Tester sollten Grundkenntnisse der Softwareentwicklung vorweisen können. Ein Nachvollziehen der gegenseitigen Aufgaben und Probleme wird dadurch erleichtert.

Die geschilderte Problematik zwischen Entwickler und Tester besteht analog auf der Managementebene. Der Testmanager muss dem Projektmanager die Testergebnisse mitteilen und ist damit oft Überbringer von schlechten Nachrichten. Der Projektmanager hat dann zu entscheiden, ob er den Auslieferungstermin noch halten kann und eventuell Software mit bekannten Fehlerwirkungen ausliefert, oder ob die Auslieferung sich verzögert und die zusätzliche Zeit zur Fehlerkorrektur genutzt wird. Die Entscheidung muss in Abhängigkeit der Schwere der Fehlerwirkung getroffen werden und ob die Fehlerzustände in der Software gegebenenfalls umgangen werden können. Ein solches Umgehen von Fehlern wird auch als *work around* bezeichnet.

## 2.6 Zusammenfassung

- Tests sind wichtige Maßnahmen zur Qualitätssicherung in der Softwareentwicklung. Die internationale Norm DIN ISO 9126 [ISO 9126] definiert entsprechende Qualitätsmerkmale.
- Fachbegriffe im Gebiet des Softwaretests werden häufig sehr unterschiedlich definiert und verwendet, was zu Missverständnissen führen kann. Die Kenntnis einschlägiger Normen (u. a. [BS 7925-1], [IEEE 610.12], [ISO 9126], [DIN 66271]) und der dort enthaltenen Begriffsdefinitionen ist deshalb wichtiger Bestandteil der Ausbildung zum »Certified Tester«. Das Glossar im Anhang dieses Buches stellt die relevanten Begriffe zusammen.
- Für den vollständigen Test von in der Praxis vorkommenden Softwareprogrammen wäre eine astronomische Anzahl von Testfällen erforderlich. Ein vollständiges »Austesten« von Programmen ist daher nicht möglich. Jeder Test ist deshalb immer nur eine Stichprobe,

die Fehlerwirkungen mit einer gewissen Wahrscheinlichkeit aufdeckt. Fehlerfreiheit kann durch das Testen nicht nachgewiesen werden.

- Bei jedem Softwareprogramm ist individuell zu entscheiden, wie intensiv und wie umfangreich es zu testen ist. Diese Entscheidung muss in Abhängigkeit zum erwarteten Risiko bei fehlerhaftem Verhalten des Programms getroffen werden. Die Priorisierung der Tests soll so erfolgen, dass bei einer (vorzeitigen) Beendigung des Testens zu einem beliebigen Zeitpunkt das bis dahin beste Ergebnis (Aufdeckung möglichst aller schwer wiegenden Fehler) erreicht wird.
- Der fundamentale Testprozess besteht aus den Phasen Planung, Spezifikation, Testdurchführung, Protokollierung und anschließender Auswertung. Das Testen kann beendet werden, wenn vorab definierte Testendekriterien erfüllt sind.
- Ein Testfall besteht aus Eingabewert, Sollergebnis und der Angabe definierter Vorbedingungen, unter denen der Testfall ablaufen muss, sowie der spezifizierten Nachbedingungen. Wird der Testfall ausgeführt, zeigt das Testobjekt ein Istverhalten. Sind Soll- und Istverhalten verschieden, liegt eine Fehlerwirkung vor. Die Sollwerte müssen für jeden Testfall vor Testdurchführung im Rahmen der Testspezifikation bestimmt werden (Testorakel).
- Menschen machen Fehler, aber sie geben es nur ungern zu! Aus diesem Grund spielen beim Testen auch psychologische Aspekte eine nicht zu vernachlässigende Rolle.

# 3 Testen im Softwarelebenszyklus

*Dieses Kapitel erklärt anhand des allgemeinen V-Modells, welche Rolle das Testen im gesamten Lebenszyklus eines Softwaresystems spielt und welche Teststufen und Testmethoden im Entwicklungsverlauf zum Einsatz kommen.*

Jedes Softwareentwicklungsvorhaben sollte nach einem im Voraus festgelegten Vorgehensmodell geplant und durchgeführt werden. Wichtige Vertreter solcher Modelle wurden in Kapitel 2.2 erläutert. Jedes dieser Modelle enthält auch gewisse Modellvorstellungen über das Testen von Software. Aus Sicht des Testens spielt hier das allgemeine V-Modell nach [Boehm 79] eine besondere Rolle.

*Testen in Vorgehensmodellen*

Boehms Modell zeigt die Testaktivitäten als gleichwertig zur Entwicklung und Programmierung. Das Verständnis vom Softwaretest wurde damit nachhaltig beeinflusst. Jeder Tester, aber auch jeder Entwickler sollte das V-Modell und die dort enthaltenen Vorstellungen über das Testen kennen. Auch wenn im Projekt nach einem anderen Vorgehensmodell gearbeitet wird, lassen sich die im Folgenden dargestellten Prinzipien übertragen.

## 3.1 Das allgemeine V-Modell

Die Grundidee des allgemeinen V-Modells ist, dass Entwicklungsarbeiten und Testarbeiten zueinander korrespondierende, gleichberechtigte Tätigkeiten sind. Bildlich dargestellt wird dies durch die zwei Äste eines »V«s.

Der linke Ast steht für die immer detaillierter werdenden Entwicklungsschritte, in deren Verlauf das vom Kunden gewünschte System schrittweise entworfen und schließlich programmiert wird. Der rechte Ast steht für Integrations- und Testarbeiten, in deren Verlauf elementare Programmbausteine sukzessive zu größeren Teilsystemen zusammengesetzt (integriert) und jeweils auf richtige Funktion geprüft werden. →Integration und Test enden mit der Abnahmeprüfung des auf diesem Weg entstandenen Gesamtsystems. Abbildung 3-1 zeigt ein solches V-Modell[4].

4. Das V-Modell wird in zahlreichen unterschiedlichen Versionen dargestellt. Benennung und Anzahl der Phasen variieren je nach Literaturquelle oder Interpretation der anwendenden Firma.

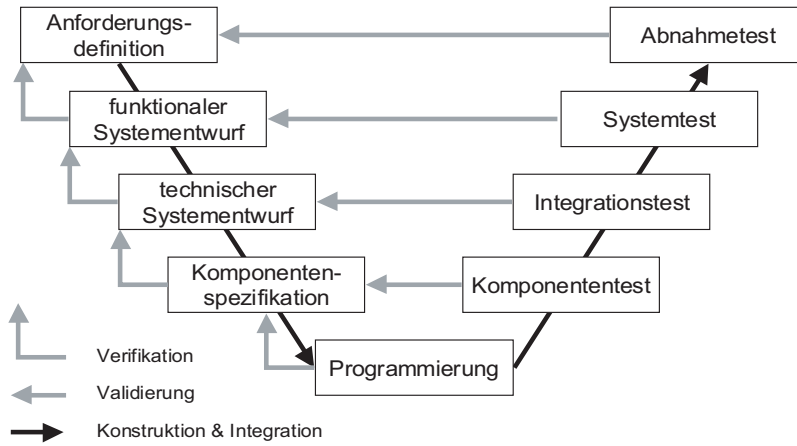



***Abb. 3–1***
*Allgemeines V-Modell*

Die konstruktiven Aktivitäten im linken Ast sind aus dem Wasserfall-Modell bekannt:

- **Anforderungsdefinition**: Die Wünsche und Anforderung des Auftraggebers oder späteren Systemanwenders werden gesammelt, spezifiziert und verabschiedet. Zweck und gewünschte Leistungsmerkmale des zu erstellenden Softwaresystems liegen damit fest.
- **Funktionaler Systementwurf**: Die Anforderungen werden auf Funktionen und Dialogabläufe des neuen Systems abgebildet.
- **Technischer Systementwurf**: Die technische Realisierung des Systems wird entworfen. Hierzu gehören u. a.: Definition der Schnittstellen zur Systemumwelt und die Zerlegung des Systems in überschaubarere Teilsysteme (Systemarchitektur), die möglichst unabhängig voneinander entwickelt werden können.
- **Komponentenspezifikation**: Für jedes Teilsystem werden Aufgabe, Verhalten, innerer Aufbau und Schnittstelle zu anderen Teilsystemen definiert.
- **Programmierung**: Programmierung (Implementierung) jedes spezifizierten Bausteins (Modul, Unit, Klasse o. Ä.) in einer Programmiersprache.

Entlang dieser Konstruktionsphasen wird das Softwaresystem zunehmend detaillierter beschrieben. Da Fehler am einfachsten auf derselben Abstraktionsstufe gefunden werden, auf der sie entstanden sind, ordnet der rechte Ast nun jedem Spezifikations- bzw. Konstruktionsschritt eine korrespondierende Teststufe zu:

- Der **Komponententest** (s. Kap. 3.2) prüft, ob jeder einzelne Softwarebaustein (Komponente) für sich die Vorgaben seiner Spezifikation erfüllt.
- Der →**Integrationstest** (s. Kap. 3.3) prüft, ob Gruppen von Komponenten wie im technischen Systementwurf vorgesehen zusammenspielen.
- Der **Systemtest** (s. Kap. 3.4) prüft, ob das System als Ganzes die spezifizierten Anforderungen erfüllt.
- Der →**Abnahmetest** (s. Kap. 3.5) prüft, ob das System aus Kundensicht die vertraglich vereinbarten Leistungsmerkmale aufweist.

In jeder Teststufe ist also zu überprüfen, ob die Entwicklungsergebnisse diejenigen Anforderungen erfüllen, die auf der jeweiligen Abstraktionsstufe relevant bzw. spezifiziert sind. Dieses Prüfen der Entwicklungsergebnisse gegen die ursprünglichen Anforderungen wird →Validierung genannt.

*Löst ein Produkt die gestellte Aufgabe?*

Beim Validieren[5] bewertet der Tester, ob ein (Teil-)Produkt eine festgelegte (spezifizierte) Aufgabe tatsächlich löst und deshalb für seinen Einsatzzweck tauglich bzw. nützlich ist.

*Ist es das richtige System?*

Untersucht wird, ob das Produkt im Kontext der beabsichtigten Produktnutzung sinnvoll ist

*Erfüllt ein Produkt seine Spezifikation?*

Neben validierenden Prüfungen fordert das V-Modell auch so genannte verifizierende[6] Prüfungen. →Verifikation ist im Gegensatz zur Validierung auf eine einzelne Entwicklungsphase bezogen und soll die Korrektheit und Vollständigkeit eines Phasenergebnisses relativ zu seiner direkten Spezifikation (Phaseneingangsdokumente) nachweisen.

*Ist das System richtig?*

Untersucht wird, ob die Spezifikationen korrekt umgesetzt wurden, unabhängig von einem beabsichtigten Zweck oder Nutzen des Produkts.

In der Praxis beinhaltet jeder Test beide Aspekte, wobei der Validierungsanteil mit steigender Teststufe zunimmt. Zusammenfassend hier noch einmal die wichtigsten Kennzeichen und Modellvorstellungen des V-Modells:

*Kennzeichen des allgemeinen V-Modells*

- Konstruktions- und Testaktivitäten sind getrennt, aber gleichwertig (linke Seite/rechte Seite).
- Das »V« veranschaulicht die Prüfaspekte Verifikation und Validierung.
- Es werden arbeitsteilige Teststufen unterschieden, wobei jede Stufe »gegen« ihre korrespondierende Entwicklungsstufe testet.

5. validieren: bekräftigen, für gültig erklären
6. verifizieren: beweisen, nachprüfen

Das V-Modell erweckt den Eindruck, dass Testen erst relativ spät, nämlich nach der Implementierung beginnen würde. Dies ist falsch. Die Teststufen im rechten Ast des Modells sind als Phasen der Testdurchführung zu verstehen. Die zugehörige Testvorbereitung (Testplanung, Testspezifikation) startet früher und wird parallel zu den Entwicklungsschritten im linken Ast durchgeführt[7].

Die Unterscheidung verschiedener Teststufen im V-Modell ist mehr als nur eine zeitliche Unterteilung von Testaktivitäten. Vielmehr werden hier technisch sehr unterschiedliche Teststufen definiert, in denen unterschiedliche Ziele verfolgt werden und deshalb unterschiedliche Testmethoden, unterschiedliche Testwerkzeuge und spezialisiertes Testpersonal zum Einsatz kommen. Die genauen Inhalte und Vorgehensweisen in jeder Teststufe werden in den folgenden Kapitel erklärt.

## 3.2 Komponententest

### 3.2.1 Begriffsklärung

In der ersten Teststufe, dem Komponententest, werden die in der nach dem V-Modell unmittelbar vorangehenden Programmierphase erstellten Softwarebausteine erstmalig einem systematischen Test unterzogen.

Abhängig davon, welche Programmiersprache die Entwickler einsetzen, werden diese kleinsten Softwareeinheiten unterschiedlich bezeichnet, zum Beispiel als Module, Units oder Klassen (im Fall objektorientierter Programmierung). Die entsprechenden Tests werden →Modul-, →Unit- bzw. →Klassentest genannt.

*Komponente und Komponententest*

Von der verwendeten Programmiersprache abstrahiert, wird von Komponente oder Softwarebaustein gesprochen. Der Test eines solchen einzelnen Softwarebausteins wird als Komponententest bezeichnet.

### 3.2.2 Testobjekte

Kennzeichnend für den Komponententest ist, dass jeweils ein einzelner Softwarebaustein überprüft wird, und zwar isoliert von anderen Softwarebausteinen des Systems. Die Isolierung hat dabei den Zweck, komponentenexterne Einflüsse beim Test auszuschließen. Deckt der Test eine Fehlerwirkung auf, lässt sich deren Ursache dann klar der getesteten Komponente zuordnen.

---

7. Ein Vorgehensmodell, das diese Parallelisierung von Entwicklung und Test explizit darstellt, ist das so genannte W-Modell (s. [Spillner 00]).

Die zu testende Komponente kann auch eine aus mehreren Bausteinen zusammengesetzte Einheit sein. Entscheidend ist, dass komponenteninterne Aspekte geprüft werden, jedoch nicht die Wechselwirkung mit Nachbarkomponenten. Das ist Gegenstand des Integrationstests.

*Der Komponententest prüft komponenteninterne Aspekte*

### 3.2.3 Testumgebung

Die Testobjekte, mit denen sich der Komponententest als niedrigste Teststufe befasst, stammen gewissermaßen »frisch« von der Festplatte der Entwickler. Damit ist klar, dass in dieser Teststufe sehr entwicklungsnah gearbeitet wird.

---

***Beispiel: Test einer Klassenmethode***

Bei der Berechnung des Fahrzeugpreises im VSR-Teilsystem *DreamCar* ergibt sich der Fahrzeugpreis laut Spezifikation wie folgt:

- Ausgegangen wird vom Grundpreis des Fahrzeugs (`baseprice`), abzüglich Händlerrabatt (`discount`).
- Sondermodellaufschlag (`specialprice`) und Preis der weiteren Zusatzausstattung (`extraprice`) sind zu addieren.
- Werden drei oder mehr Zusatzausstattungen (nicht im gewählten Sondermodell enthalten) ausgewählt (`extras`), erfolgt nur auf diese Ausstattungsmerkmale eine Rabattierung von 10 %. Bei fünf oder mehr Zusatzausstattungen erhöht sich diese Rabattierung auf 15 %.
- Eine Kumulierung von Händlerrabatt und Zubehörrabatt ist nicht möglich.

Die Berechnung des Gesamtpreises erfolgt durch folgende C++-Methode:

```
double calculate_price
       (double baseprice, double specialprice,
       double extraprice, int extras, double discount)
{
       double addon_discount;
       double result;

       if (extras >= 3) addon_discount = 10;
       else if (extras >= 5) addon_discount = 15;
       else addon_discount = 0;

       if (discount > addon_discount)
              addon_discount = discount;

       result = baseprice/100.0*(100-discount)
              + specialprice
       + extraprice/100.0*(100-addon_discount);

       return result;
}
```

Um die Preisberechnung zu testen, benutzt der Tester die entsprechende Klassenschnittstelle, ruft also die Methode `calculate_price()` auf und versorgt sie mit geeigneten Testdaten. Anschließend erfasst er die Reaktion der Komponente auf diesen Aufruf. Das heißt, der Rückgabewert des vorangegangenen Methodenaufrufs wird gelesen und protokolliert. Hierzu ist ein so genannter →Testtreiber notwendig. Ein Testtreiber ist ein Programm, das die jeweiligen Schnittstellenaufrufe absetzt und anschließend die Reaktion des Testobjekts entgegennimmt.

Für das Testobjekt `calculate_price()` kann ein sehr einfacher Testtreiber beispielsweise wie folgt aussehen:

```
bool test_calculate_price() {

double price;
bool   test_ok = TRUE;

// testcase 01
price = calculate_price(10000.00,2000.00,1000.00,3,0);
test_ok = test_ok && (price == 12900.00);

// testcase 02
price = calculate_price(25500.00,3450.00,6000.00,6,0);
test_ok = test_ok && (price == 34050.00);

// testcase ...

// test result
return test_ok;
}
```

Der Testtreiber im Beispiel ist sehr einfach programmiert. Sinnvolle Erweiterungen wären z. B. die Protokollierung der Testdaten und Resultate mit Datum/Uhrzeit, das Einlesen der Testfälle aus einer Datentabelle usw.

Zur Erstellung des Testtreibers ist Entwickler-Know-how notwendig. Der Programmcode des Testobjekts (im Beispiel eine Klassenmethode) – zumindest der Code der Schnittstelle – muss verfügbar und verstanden sein, damit im Testtreiber der Aufruf des Testobjekts korrekt programmiert werden kann. Das heißt, die Programmiersprache muss beherrscht werden und entsprechende Programmierwerkzeuge müssen vorhanden sein, um einen passenden Testtreiber zu schreiben.

Aus diesen Gründen werden Komponententests sehr oft von den Entwicklern selbst durchgeführt. Es wird dann vom Entwicklertest ge-

sprochen, obwohl ein Komponententest gemeint ist. Die Nachteile, wenn ein Entwickler sein eigenes Programm testet, wurden in Kapitel 2.5 schon aufgezählt.

***Tipp***

- Der Einsatz vorgefertigter Testtreiber kann den Komponententest erleichtern, da die aufwändige Programmierung einzelner Testtreiber für jede einzelne Komponente entfällt. Vorgefertigte Testtreiber sind am Markt erhältliche generische Testtreiber oder individuell erstellte, die für alle bzw. mehrere Komponententests im Projekt genutzt werden. Werden einheitliche Testtreiber eingesetzt, können die Testarbeiten auch leichter von Teamkollegen[8] erledigt werden, die mit der einzelnen Komponente und der Entwicklungsumgebung nicht im Detail vertraut sind. Solche Testtreiber können z. B. über eine Kommandoschnittstelle bedient werden und bieten komfortable Mechanismen zur Handhabung der Testdaten sowie zur Protokollierung und Auswertung der Tests. Da alle Testdaten und Testprotokolle vollkommen gleichartig strukturiert sind, ist auch eine komponentenübergreifende Testauswertung möglich.

### 3.2.4 Testziele

Die Teststufe Komponententest wird nicht nur durch die Art der Testobjekte und der Testumgebung charakterisiert, im Komponententest verfolgt der Tester auch für diese Stufe spezifische Testziele.

*Test der Funktionalität*

Wichtigste Aufgabe des Komponententests ist die Sicherstellung, dass das jeweilige Testobjekt die laut seiner Spezifikation geforderte Funktionalität korrekt und vollständig realisiert. Diese Tests werden daher auch →Funktionstest oder →funktionaler Test genannt. Funktionalität ist dabei gleichbedeutend mit dem Ein-/Ausgabe-Verhalten des Testobjekts. Um Korrektheit und Vollständigkeit der Implementierung zu prüfen, wird die Komponente einer Reihe von Testfällen unterzogen, wobei jeder Testfall eine bestimmte Ein-/Ausgabe-Kombination (Teilfunktionalität) abdeckt.

***Beispiel: Test der VSR-Preisberechnung***

Bei den Testfällen zur Preisberechnung des *CarConfigurators* im vorangehenden Beispiel ist diese Prüfung der Ein-/Ausgabe-Kombination sehr schön zu sehen. Jeder Testfall versorgt das Testobjekt mit einer bestimmten Preiskombination bei einer bestimmten Zubehöranzahl. Dann wird geprüft, ob das Testobjekt aus diesen Daten den korrekten Gesamtpreis berechnet. Testfall 2 prüft beispielsweise die Teilfunktionalität »Rabattierung bei fünf oder mehr Zusatzausstattungen«. Wird Testfall 2 ausgeführt, ist festzustellen, dass das

8. Manchmal arbeiten zwei Entwickler paarweise zusammen, um gegenseitig die vom Kollegen erstellte(n) Komponente(n) zu testen.

Testobjekt einen falschen Gesamtpreis liefert. Testfall 2 erzeugt eine Fehlerwirkung (s. a. Kap. 5.2.4). Das Testobjekt realisiert die spezifizierte Funktionalität nicht vollständig.

---

Typische Softwaredefekte, die beim funktionalen Komponententest aufgedeckt werden, sind Berechnungsfehler oder fehlende und falsch gewählte Programmpfade (z. B. vergessene Sonderfälle).

Jede Softwarekomponente muss später beim Betrieb des Gesamtsystems mit einer Vielzahl von Nachbarkomponenten zusammenarbeiten und Daten austauschen. Dabei ist nicht auszuschließen, dass die Komponente unter Umständen auch falsch, d. h. entgegen ihrer Spezifikation angesprochen oder verwendet wird. In solchen Fällen sollte die falsch angesprochene Komponente nicht gleich den Dienst einstellen und das Gesamtsystem zum Absturz bringen. Vielmehr sollte sie die Fehlersituation abfangen und »vernünftig« bzw. robust reagieren.

*Test auf Robustheit*

Der Test auf →Robustheit ist deshalb ein weiterer sehr wichtiger Aspekt des Komponententests. Das Vorgehen ist dasselbe wie beim funktionalen Test. Allerdings werden als Testeingaben Methodenaufrufe, Daten und Sonderfälle verwendet, die laut Spezifikation eigentlich unzulässig oder nicht vorgesehen sind. Solche Testfälle werden auch Negativtest genannt. Als Reaktion bzw. Ausgabewert der Komponente wird eine angemessene Ausnahmebehandlung (*exception handling*) erwartet. Fehlen solche Ausnahmebehandlungen, treten möglicherweise Wertebereichsfehler auf (z. B. Division durch Null, Zugriff über Null-Pointer u. Ä.), die unter Umständen zum Programmabsturz führen.

---

***Beispiel: Negativtest***

Im Preisberechnungsbeispiel sind solche Negativtests zum Beispiel Aufrufe mit negativen Zahlenwerten oder falschen Datentypen (`char statt int` o. Ä.)[9]:

```
// testcase 20
price = calculate_price(-1000.00,0.00,0.00,0,0);
test_ok = test_ok && (ERR_CODE == INVALID_PRICE);
...
// testcase 30
price = calculate_price("abc",0.00,0.00,0,0);
test_ok = test_ok && (ERR_CODE == INVALID_ARGUMENT);
```

---

9. Abhängig vom eingesetzten Compiler können Datentypfehler schon beim Übersetzungslauf aufgedeckt werden.

*Exkurs*

Einige interessante Aspekte werden hier deutlich:

- Sinnvolle Negativtests finden sich mindestens so viele wie »Positivtests«.
- Der Testtreiber muss erweitert werden, um die vom Testobjekt vorgenommene Ausnahmebehandlung auswerten zu können.
- Die Ausnahmebehandlung im Testobjekt (z. B. Auswertung von `ERR_CODE` im Beispiel) erfordert dort zusätzliche Funktionalität. In der Praxis dienen nicht selten mehr als 50 % des Programmcodes der Behandlung solcher Ausnahmesituationen. Robustheit hat ihren Preis.

Neben Funktionalität und Robustheit sollten im Komponententest des Weiteren alle diejenigen Komponenteneigenschaften überprüft werden, die die Qualität der Komponente maßgebend beeinflussen und die in höheren Teststufen nicht mehr oder nur mit wesentlich höherem Aufwand geprüft werden können. Dies gilt beispielsweise für die nicht funktionalen Eigenschaften Effizienz und Wartbarkeit.

*Test der Effizienz*

Die Effizienz gibt an, wie wirtschaftlich die Komponente mit den verfügbaren Rechnerressourcen umgeht. Teilaspekte sind hier Verbrauch von Speicherplatz, benötigte Rechenzeit bzw. Ausführungszeit komponenteninterner Funktionen/Algorithmen usw. Im Unterschied zu den meisten anderen Testzielen kann die Effizienz eines Testobjekts anhand geeigneter Teilkriterien (z. B. Speicherverbrauch in Kilobyte, Antwortzeit in Millisekunden) im Test exakt gemessen werden. Entsprechende Effizienzuntersuchungen werden selten für alle Komponenten eines Systems vorgesehen. Nur dort, wo entsprechende Forderungen in den Anforderungskatalogen oder Spezifikationen definiert sind, müssen diese dann im Komponententest auch verifiziert werden. Beispielsweise beim Test eingebetteter Systeme, wenn hardwareseitig nur limitierte Ressourcen zugeteilt werden. Oder im Falle von Echtzeitsystemen, wenn vorgegebene Zeitschranken garantiert werden müssen.

*Test auf Wartbarkeit*

Unter Wartbarkeit werden all diejenigen Eigenschaften eines Programms subsumiert, die Einfluss darauf haben, wie leicht oder schwer es fällt, das Programm zu ändern oder weiterzuentwickeln. Entscheidend dabei ist, wie viel Aufwand es den Entwickler kostet, das vorhandene Programm und dessen Kontext zu verstehen. Das gilt für den Entwickler des ursprünglichen Programms, der nach Monaten oder Jahren mit einer Weiterentwicklung beauftragt wird, genauso wie für einen Entwickler, der Code eines Kollegen übernimmt. Bei der Prüfung der Wartbarkeit stehen deshalb folgende Prüfaspekte im Vordergrund: Codestruktur, Modularität, Kommentierung des Codes, Verständlichkeit und Aktualität der Dokumentation usw.

***Beispiel: schwer wartbarer Code***

Der Code des Beispiels `calculate_price()` weist hier einige Defizite auf. So fehlen Kommentare völlig, und numerische Konstanten sind nicht als solche deklariert, sondern fest in den Code hineincodiert. Muss ein solcher Wert nachträglich geändert werden, ist unklar, ob und an welchen anderen Stellen im Gesamtsystem diese Konstante außerdem vorkommt und ebenfalls zu ändern ist.

Solche Eigenschaften lassen sich natürlich nicht durch →dynamische Tests (s. Kap. 5) überprüfen. Notwendig sind Analysen der Programmtexte und Spezifikationen. Mittel hierzu ist der →statische Test und insbesondere das Review (s. Kap. 4.1). Da Eigenschaften der einzelnen Komponente untersucht werden, werden solche Analysen aber zweckmäßigerweise im Rahmen des Komponententests durchgeführt.

### 3.2.5 Teststrategie

Wie oben erklärt, wird im Komponententest sehr entwicklungsnah gearbeitet. Der Tester hat in der Regel Zugang zum Sourcecode, was den Komponententest zur Domäne des White-box-Tests (s. Kap. 5.2) macht.

*White-box-Test*

Der Tester kann Testfälle unter Ausnutzung seines Wissens über komponenteninterne Programmstrukturen, Methoden und Variablen entwerfen. Auch bei der Testdurchführung ist das Vorliegen des Programmcodes nützlich. Mit entsprechenden Werkzeugen (→Debugger, s. Kap. 7.1.3) können während des Testablaufs Programmvariablen beobachtet werden, um auf ein korrektes oder fehlerhaftes Verhalten der Komponente zu schließen. Der interne Zustand der Komponente kann aber nicht nur beobachtet, sondern per Debugger auch manipuliert werden. Dies ist besonders für Robustheitstests hilfreich, da so bestimmte Ausnahmesituationen vom Tester gezielt ausgelöst werden können.

***Beispiel: Code als Testgrundlage***

Bei der Analyse des Codes von `calculate_price()` kann folgende Anweisung als testrelevante Programmstelle erkannt werden:

```
if (discount > addon_discount)
      addon_discount = discount;
```

Zusätzliche Testfälle, die dazu führen, dass die Bedingung `(discount > addon_discount)` erfüllt wird, können anhand des Codes leicht abgeleitet werden. Die Spezifikation der Preisberechnung enthält hierzu keinerlei Informationen, eine entsprechende Funktionalität ist dort überhaupt nicht vorgesehen.

In der Praxis wird aber in vielen Fällen auch der Komponententest »nur« als Black-box-Test durchgeführt, dass heißt, die innere Struktur wird zur Auswahl der Testfälle nicht herangezogen. Zum einen bestehen reale Softwaresysteme oft aus hunderten oder tausenden elementarer Komponenten. Ein Einstieg in den Code ist hier sicher nur bei ausgewählten Komponenten praktikabel. Zum anderen werden im Verlauf der Integration die elementaren Programmbausteine zu größeren Einheiten zusammengesetzt. Oft sind für die Tester auch in der ersten Teststufe nur solche zusammengesetzten Programmbausteine als testbare Einheiten sichtbar. Diese Testobjekte sind dann aber schon zu groß, um mit vertretbarem Aufwand Beobachtungen oder Eingriffe auf Codeebene verstehen und vornehmen zu können. Ob beim Komponententest elementare oder bereits zusammengesetzte Programmbausteine getestet werden, ist Sache der Integrations- und Testplanung und dort festzulegen.

## 3.3 Integrationstest

### 3.3.1 Begriffsklärung

Als zweite Teststufe nach dem Komponententest schließt sich im V-Modell der Integrationstest an. Der Integrationstest setzt voraus, dass die ihm übergebenen Testobjekte (d. h. einzelne Komponenten) bereits getestet sind und Defekte möglichst korrigiert wurden.

*Integration*

Gruppen dieser Komponenten werden dann von Entwicklern, Testern oder speziellen Integrationsteams zu größeren Baugruppen bzw. Teilsystemen verbunden. Dies wird als Integration bezeichnet.

*Integrationstest*

Anschließend muss getestet werden, ob das Zusammenspiel aller Einzelteile miteinander richtig funktioniert. Der Integrationstest hat also das Ziel, Fehlerzustände in Schnittstellen und im Zusammenspiel zwischen integrierten Komponenten zu finden.

Warum ist Integrationstesten notwendig, wenn doch jeder Einzelbaustein bereits getestet wurde? Unser Fallbeispiel veranschaulicht die Problematik:

---

***Beispiel: Integrationstest VSR-DreamCar***

Das VSR-Teilsystem *DreamCar* (vgl. Abb. 2-1) ist eine aus elementaren Softwarebausteinen zusammengesetzte Komponente.

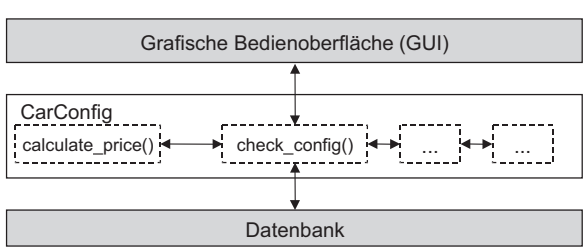
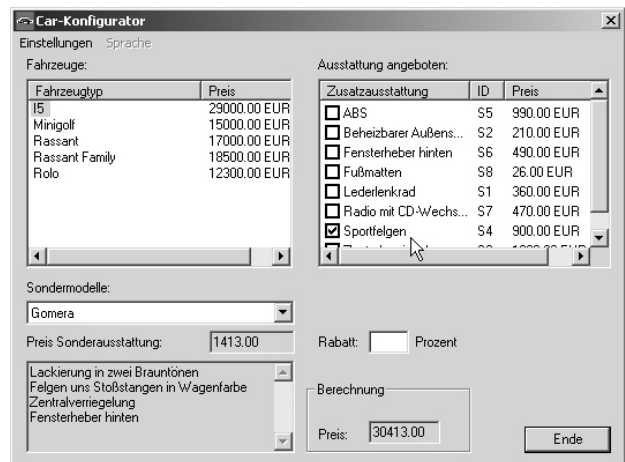



***Abb. 3–2***
*Struktur der Komponente VSR-DreamCar*

Ein Element ist die Klasse `CarConfig` mit den Methoden `calculate_price()`, `check_config()` und weiteren Methoden. `check_config()` bezieht sämtliche Fahrzeugdaten aus einer Datenbank und bietet diese dem Endanwender über eine grafische Bedienoberfläche (GUI) zur Auswahl an. Aus Sicht des Endanwenders sieht dies wie in Abbildung 3-3 gezeigt aus.

***Abb. 3–3***
*Bedienoberfläche VSR-DreamCar*




Stellt der Anwender ein Fahrzeug zusammen, überprüft `check_config()` jede Konfiguration (Fahrzeugbasismodell, Sondermodellausstattung, Liste des weiteren Zubehörs) auf Plausibilität und sorgt parallel für die Berechnung des zugehörigen Preises. Im Beispiel (s. Abbildung 3-3) müsste sich aufgrund des ausgewählten Fahrzeugs, Sondermodells und der Ausstattung ein Gesamtpreis von 29000 EUR + 1413 EUR + 900 EUR = 31313 EUR ergeben. Angezeigt wird aber nur 30413 EUR. Offensichtlich kann in der zum Test vorliegenden Programmversion Zubehör (z. B. Sportfelgen) ausgewählt werden, ohne dass dies in den Gesamtpreis eingeht. Irgendwo auf dem Weg zwischen GUI und `calculate_price()` geht verloren, dass Sportfelgen ausgewählt sind.

Zeigen die Testprotokolle des jeweils vorangegangenen Komponententests, dass der Fehlerzustand nicht in den Methoden `calculate_price()` oder `check_config()` liegt, kommt als Ursache eine fehlerhafte Datenübergabe zwischen GUI und `check_config()` oder zwischen `check_config()` und `calculate_price()` in Frage.

---

Auch ein vorausgehender vollständiger Komponententest kann solche Schnittstellenfehler nicht ausschließen. Der Integrationstest als weitere Teststufe ist deshalb notwendig. Er muss solche Probleme erkennen und deren Ursachen eingrenzen.

***Beispiel: VSR-Integrationstest***

Mit der Integration der Einzelbausteine zum Teilsystem *DreamCar* hat der Integrationstest im Projekt *VSR* allerdings erst begonnen. Die anderen VSR-Teilsysteme (s. Kap. 2, Abb. 2-1) müssen ebenfalls integriert werden. Anschließend sind diese dann untereinander zu verbinden. So muss *DreamCar* an das Teilsystem *ContractBase* angeschlossen werden, mit welchem wiederum die Teilsysteme *JustInTime* (Bestellwesen), *NoRisk* (Fahrzeugversicherung) und *EasyFinance* (Fahrzeugfinanzierung) verbunden sind. Als einer der letzten Integrationsschritte wird *VSR* mit dem externen Zentralrechner (*Host*) im Konzernrechenzentrum verbunden.

*»Integrationstest im Großen«*

Wie das Beispiel zeigt, sind auch die Schnittstellen zur Systemumgebung Gegenstand von Integration und Integrationstest. Werden Schnittstellen zu externen Softwaresystemen überprüft, wird manchmal auch von einem »Integrationstest im Großen« oder Systemintegrationstest gesprochen. Ein spezielles Risiko liegt hier darin, dass das Entwicklungsteam nur »eine Hälfte« einer solchen Schnittstelle nach extern kontrolliert. Die »andere Hälfte« wird vom externen System bestimmt und kann sich unerwartet ändern. Ein bestandener Integrationstest ist also keine Garantie für eine jederzeit einwandfreie Funktion an dieser Stelle.

### 3.3.2 Testobjekte

*Zusammengesetzte Komponenten*

Im Zuge der Integration werden die Einzelbausteine schrittweise zu größeren Einheiten zusammengesetzt (s. a. Kap. 3.3.5). An jeden solchen Schritt kann und sollte sich im Idealfall ein Integrationstest anschließen. Jedes entstandene Teilsystem kann anschließend Basis für die weitere Integration noch größerer Einheiten sein. Testobjekte des Integrationstests können also auch mehrfach zusammengesetzte Einheiten sein.

*Fremdsysteme oder zugekaufte Komponenten*

In der Praxis wird ein Softwaresystem selten auf der grünen Wiese entwickelt, sondern ein vorhandenes System wird verändert, ausgebaut oder mit anderen Systemen gekoppelt. Auch sind viele Systemkomponenten Standardprodukte, die am Markt zugekauft werden (beispielsweise die Datenbank in *DreamCar*). Im Komponententest werden solche Alt- oder Standardkomponenten vermutlich nicht beachtet. Im Integrationstest müssen diese Systemteile jedoch berücksichtigt und deren Zusammenspiel mit anderen Teilen überprüft werden.

### 3.3.3 Testumgebung

Beim Integrationstest werden ebenfalls Testtreiber benötigt, die die Testobjekte mit Testdaten versorgen und Ergebnisse entgegennehmen sowie protokollieren. Da die Testobjekte zusammengesetzte Komponenten sind, die keine anderen Schnittstellen »nach außen« aufweisen als ihre Einzelkomponenten, ist es nahe liegend und sinnvoll, die vorhandenen Testtreiber des Komponententests wiederzuverwenden.

*Wiederverwendung der Testumgebung*

War der Komponententest gut organisiert, gibt es entweder einen gemeinsamen, generischen Testtreiber für alle Komponenten oder zumindest Testtreiber, die nach einer einheitlichen Architektur entworfen wurden und zueinander kompatibel sind. In diesem Fall können die Tester diese Treiber mit geringem Aufwand übernehmen und wiederverwenden.

Bei einem schlecht organisierten Komponententest gibt es vielleicht nur für einige wenige Testobjekte brauchbare Treiber, die dann oftmals vollkommen unterschiedlich zu bedienen sind. Der Preis dafür ist, dass die Tester jetzt zu einem sehr späten Projektzeitpunkt viel Aufwand in die Erstellung oder Nachbesserung der Testumgebung investieren müssen und somit wertvolle Zeit für die Testdurchführung verloren geht.

Da Schnittstellenaufrufe und der Datenverkehr über die Testtreiberschnittstellen getestet werden müssen, werden im Integrationstest als zusätzliches Diagnoseinstrument oft so genannte →Monitore benötigt. Monitore sind Programme, die Datenbewegungen zwischen Komponenten mitlesen und protokollieren. Am Markt erhältlich sind Monitore für Standardprotokolle (z. B. Netzwerkprotokolle). Zur Beobachtung projektspezifischer Komponentenschnittstellen sind individuelle Monitore zu entwickeln.

### 3.3.4 Testziele

*Falsche Schnittstellenformate*

Die Testziele der Teststufe Integrationstest sind klar: Schnittstellenfehler aufdecken. Probleme können schon beim Versuch der Integration zweier Bausteine auftreten, wenn diese sich nicht zusammenbinden lassen, weil ihre Schnittstellenformate nicht passen, weil einige Dateien fehlen oder die Entwickler das System in ganz andere Komponenten aufgeteilt haben, als spezifiziert war (vgl. statischer Test, Kap. 4.2).

*Typische Fehler beim Datenaustausch*

Die schwerer zu findenden Probleme betreffen allerdings die Ausführung der miteinander in Wechselwirkung stehenden Programmteile, die nur durch einen dynamischen Test aufgedeckt werden können. Dies sind Fehlerzustände im Datenaustausch bzw. in der Kommunikation zwischen den Komponenten. Folgende Typen des Fehlerzustands können grob unterschieden werden:

- Eine Komponente übermittelt keine oder syntaktisch falsche Daten, so dass die empfangende Komponente nicht arbeiten kann oder abstürzt (funktionaler Fehler einer Komponente, inkompatible Schnittstellenformate, Protokollfehler).
- Die Kommunikation funktioniert, aber die beteiligten Komponenten interpretieren übergebene Daten unterschiedlich (funktionaler Fehler einer Komponente, widersprüchliche oder fehlinterpretierte Spezifikationen).
- Die Daten werden richtig übergeben, aber zum falschen oder verspäteten Zeitpunkt (Timing-Problem) oder in zu kurzen Zeitintervallen (Durchsatz- oder Lastproblem).

***Beispiel: Integrationsfehler in VSR***

Im VSR-Integrationstest könnten folgende Fehlerwirkungen auftreten, die den beschriebenen Typen von Fehlerzuständen zugeordnet werden können:

- In der *DreamCar*-GUI selektiertes Zubehör wird nicht an `check_config()` übergeben. Als Folge wären Preis und Bestelldaten fehlerhaft.
- In *DreamCar* wird die Wagenfarbe durch eine bestimmte Codenummer dargestellt (z. B. 442 für blaumetallic). Im Bestellsystem auf Hostseite werden einige Codenummern aber anders interpretiert (442 steht dort beispielsweise für rot-perleffekt). Eine aus Sicht des *VSR* vollkommen korrekte Bestellung hätte zu einer falschen Lieferung geführt.
- Der Hostrechner bestätigt eine übertragene Bestellung nach Prüfung auf Lieferbarkeit mit einer Bestellbestätigung. In einigen Fällen benötigt diese Prüfung auf Lieferbarkeit so lange, dass *VSR* einen Verbindungsfehler annimmt und den Bestellvorgang abbricht. Ein Kunde hätte seinen sorgfältig zusammengestellten Wunschwagen nicht bestellen können.

Keiner dieser Typen von Fehlerzuständen kann im Komponententest gefunden werden, denn die Fehlerwirkung äußert sich erst in der Wechselwirkung zwischen zwei Softwarebausteinen.

*Ist der Komponententest verzichtbar?*

Kann auf den Komponententest verzichtet werden, so dass alle Testfälle erst nach erfolgter Integration durchgeführt werden? Das ist natürlich möglich und leider auch eine in der Praxis oft anzutreffende Vorgehensweise. Damit verbunden sind aber gravierende Nachteile:

- Die meisten Fehlerwirkungen, die in einem derart angelegten Test auftreten werden, sind durch funktionale Fehlerzustände einzelner Komponenten verursacht. Es wird also ein impliziter Komponententest in einer dazu nicht geeigneten Testumgebung durchgeführt, die den Zugang zur Einzelkomponente erschwert.

- Weil kein geeigneter Zugang zur Einzelkomponente möglich ist, können manche Fehlerwirkungen nicht provoziert und viele Fehlerzustände deshalb nicht gefunden werden (vgl. Kap. 3.2.4).
- Wenn eine Fehlerwirkung oder ein Ausfall im Test auftritt, kann es schwierig oder sogar unmöglich sein, seinen Entstehungsort und damit seine Ursache einzugrenzen.

Der Aufwand, der durch Verzicht auf den Komponententest vermeintlich eingespart wird, wird mit einer schlechteren Fehlerfindungsrate und mit einem erhöhten Diagnoseaufwand bezahlt. Eine Kombination von Komponententest mit nachfolgendem Integrationstest ist effizienter.

### 3.3.5 Integrationsstrategien

Der eben angesprochene Aspekt der Testeffizienz, also des Verhältnisses zwischen Testkosten (Personalaufwand, Werkzeugeinsatz usw.) und Testnutzen (Anzahl und Schwere der aufgedeckten Fehlerwirkungen) in einer Teststufe führt zur Frage, in welcher Reihenfolge die Einzelkomponenten integriert werden sollten, damit die notwendigen Testarbeiten möglichst einfach und schnell durchführbar sind. Es ist Aufgabe des Testmanagers, eine für das jeweilige Projekt optimale Integrationsstrategie auszuwählen und einzuschlagen.

*Komponenten werden zu unterschiedlichen Zeitpunkten fertig*

In der Praxis besteht die Schwierigkeit, dass die verschiedenen im Projekt entstehenden Softwarekomponenten zu unterschiedlichen Zeitpunkten fertig gestellt werden, die eventuell Wochen oder Monate auseinander liegen können. Kein Projektmanager und auch kein Testmanager kann es tolerieren, dass seine Tester so lange untätig warten, bis alle Komponenten fertig sind und gemeinsam integriert werden können.

Eine nahe liegende Ad-hoc-Strategie besteht darin, einfach in der (zufälligen) Lieferreihenfolge zu integrieren. Das heißt, sobald eine Komponente ihren Komponententest absolviert hat, wird geprüft, ob sie zu einer anderen schon vorhandenen und bereits getesteten Komponente oder zu einem teilintegrierten Subsystem passt. Wenn ja, werden beide Teile integriert und der Integrationstest zwischen beiden wird durchgeführt.

---

***Beispiel: Integrationsstrategie im Projekt VSR***

Im Projekt VSR zeigt sich, dass das zentrale Teilsystem *ContractBase* komplexer ist als vermutet. Seine Fertigstellung verzögert sich um mehrere Wochen, weil die Arbeiten wesentlich aufwändiger sind, als ursprünglich geschätzt. Um nicht noch mehr Zeit zu verlieren, entscheidet der Projektmanager, die Tests mit den vorhandenen Bausteinen *DreamCar* und *NoRisk*

zu beginnen. Diese besitzen zwar keine direkte gemeinsame Schnittstelle, aber tauschen Daten über *ContractBase* aus. Zur Kalkulation einer Versicherung benötigt *NoRisk* den Wagentyp, denn dieser bestimmt die Kaskoklasse und andere Parameter. Als vorläufiger Ersatz für *ContractBase* wird ein →Platzhalter (engl. *stub*) programmiert. Dieser nimmt einfache Fahrzeugkonfigurationen von *DreamCar* entgegen, ermittelt daraus den Code des Wagentyps und reicht ihn an *NoRisk* weiter. Des Weiteren ermöglicht der Platzhalter, verschiedene prämienrelevante Daten des Versicherungsnehmers einzugeben. Die von *NoRisk* daraus berechneten Prämien werden in einem Fenster zur Kontrolle angezeigt und als Testprotokoll gespeichert. Der Platzhalter dient also als provisorischer Ersatz für das nicht fertige Teilsystem *ContractBase*.

---

Das Beispiel macht deutlich: Je früher, um Zeit zu gewinnen, mit dem Integrationstest begonnen wird, umso mehr Aufwand muss in die Erstellung von Platzhaltern investiert werden. Der Testmanager muss seine Integrationsstrategie so wählen, dass beide Faktoren (Zeitgewinn vs. Aufwand für Testumgebung) optimiert werden.

*Randbedingungen für die Integration*

Welche Strategie optimal ist (am Zeit sparendsten, am kostengünstigsten), hängt von Randbedingungen ab, die in jedem Projekt individuell analysiert werden müssen:

- Die **Systemarchitektur** bestimmt, aus welchen und wie vielen Komponenten das Gesamtsystem besteht und wie diese voneinander abhängen.
- Der **Projektplan** legt fest, zu welchen Zeitpunkten im Projekt einzelne Systemteile entwickelt werden und wann diese testbereit sein sollen.
- Das **Testkonzept** legt fest, welche Systemaspekte wie intensiv getestet werden müssen und auf welcher Teststufe das jeweils geschehen soll.

*Integrationsstrategie absprechen*

Der Testmanager muss aus diesen Randbedingungen die für sein Projekt passende Integrationsstrategie aufstellen. Da der Lieferzeitpunkt der Komponenten mit entscheidend ist, tut er gut daran, den Projektmanager bei der Erstellung des Projektplans zu beraten und darauf hinzuwirken, dass frühzeitig eine aus Testsicht günstige Lieferreihenfolge vorgesehen wird.

*Basisstrategien*

An folgenden Grundstrategien kann sich der Testmanager bei seiner Planung orientieren:

- **Top-down-Integration**: Der Test beginnt mit der Komponente des Systems, die weitere Komponenten aufruft, aber selbst (außer vom

Betriebssystem) nicht aufgerufen wird. Die untergeordneten Komponenten sind dabei durch Platzhalter ersetzt. Sukzessive werden die Komponenten niedrigerer Systemschichten hinzuintegriert. Die getestete höhere Schicht dient dabei jeweils als Testtreiber.

- Vorteil: Es werden keine bzw. nur einfache Testtreiber benötigt, da übergeordnete, bereits getestete Komponenten den wesentlichen Teil der Ablaufumgebung bilden.
- Nachteil: Untergeordnete, noch nicht integrierte Komponenten müssen durch Platzhalter ersetzt werden, was sehr aufwändig sein kann.
- **Bottom-up-Integration**: Der Test beginnt mit den elementaren Komponenten des Systems, die keine weitere Komponenten aufrufen (außer Funktionen des Betriebssystems). Größere Teilsysteme werden sukzessive aus getesteten Komponenten zusammengesetzt, mit anschließendem Test dieser Integration.
- Vorteil: Es werden keine Platzhalter benötigt.
- Nachteil: Übergeordnete Komponenten müssen durch Testtreiber simuliert werden.
- **Ad-hoc-Integration**: Die Bausteine werden z. B. in der (zufälligen) Reihenfolge ihrer Fertigstellung integriert (s. o.).
- Vorteil: Zeitgewinn, da jeder Baustein frühestmöglich in seine passende Umgebung integriert wird.
- Nachteil: Es werden sowohl Platzhalter als auch Testtreiber benötigt.

Top-down- oder Bottom-up-Integration lassen sich in Reinform nur bei streng hierarchisch gegliederten Programmsystemen einsetzen, was bei in der Praxis anzutreffenden Softwaresystemen selten gegeben ist. Daher wird in der Realität immer eine mehr oder minder individuelle Mischung obiger Integrationsstrategien gewählt.

*big bang vermeiden!*

Eine nicht inkrementelle Integration – auch *big bang* genannt – sollte vermieden werden. Eher aus Mangel einer Strategie als geplant, wird hierbei mit der Integration gewartet, bis alle Softwarebauteile entwickelt sind, und dann wird alles auf einmal zusammengeworfen. Im schlimmsten Fall wird auch auf vorgelagerte Komponententests verzichtet. Die Nachteile sind offensichtlich:

- Die Wartezeit bis zum *big bang* ist leichtfertig verlorene Testdurchführungszeit. Da Testen ohnehin immer unter Zeitmangel leidet, sollte kein einziger Testtag verschenkt werden.
- Alle Fehlerwirkungen treten geballt auf; es wird schwierig oder unmöglich sein, das System überhaupt zum Laufen zu bringen. Die

Lokalisierung und Behebung von Defekten gestaltet sich schwierig und Zeit raubend.

## 3.4 Systemtest

### 3.4.1 Begriffsklärung

Nach abgeschlossenem Integrationstest folgt vor Auslieferung des Systems als dritte Teststufe der Systemtest. Gegenstand ist der Test des integrierten Systems, um zu überprüfen, dass die spezifizierten Anforderungen vom Produkt erfüllt werden. Auch hier stellt sich die Frage, warum dies nach vorangegangenem Komponenten- und Integrationstest noch notwendig ist. Gründe dafür sind:

*Gründe für den Systemtest*

- In den niedrigeren Teststufen wurde gegen technische Spezifikationen geprüft, aus der Perspektive des Softwareherstellers. Der Systemtest betrachtet das System hingegen aus der Perspektive des Kunden und des späteren Anwenders[10]. Die Tester validieren, ob die Anforderungen vollständig und angemessen umgesetzt wurden.
- Viele Funktionen und Systemeigenschaften resultieren aus dem Ineinandergreifen aller Systemkomponenten und sind somit erst auf Ebene des Gesamtsystems beobachtbar und testbar.

***Beispiel: VSR-Systemtest***

Der Hauptzweck des VSR-Systems ist, die Bestellung eines Fahrzeugs so einfach wie möglich zu machen. Im Zuge einer Bestellung nutzt der Anwender alle Komponenten des VSR-Systems: Das Fahrzeug wird konfiguriert (*DreamCar*); Finanzierung und Versicherung werden kalkuliert (*EasyFinance, NoRisk*); die Bestellung wird übermittelt (*JustInTime*) und die Verträge werden archiviert (*ContractBase*). Nur wenn all diese Systemfunktionen quer durch alle Komponenten korrekt zusammenspielen, wird das System seinem Einsatzzweck gerecht. Ob dies der Fall ist, zeigt erst der Systemtest.

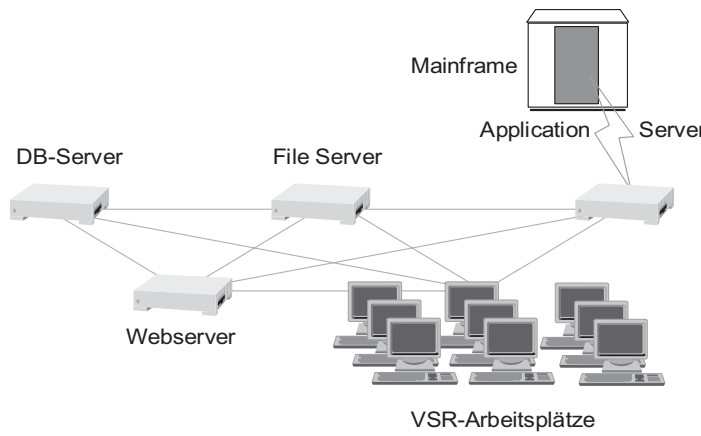
### 3.4.2 Testobjekt und Testumgebung

Mit abgeschlossenem Integrationstest liegt das komplett zusammengebaute Softwaresystem vor. Im Systemtest wird dieses System als Ganzes betrachtet, und zwar in einer Testumgebung, die der späteren →Produktivumgebung möglichst nahe kommt. Statt Testtreibern und

10. Kunde (hat das System bestellt und bezahlt) und Anwender (nutzt das System) können verschiedene Personengruppen oder Organisationen sein, mit jeweils eigenen Interessen und Systemanforderungen.

Platzhaltern sollen also auf allen Ebenen möglichst die später tatsächlich zum Einsatz kommenden Hard- oder Softwareprodukte in der Testumgebung installiert sein (Hardwareausstattung, Systemsoftware, Treibersoftware, Netzwerk, Fremdsysteme usw.). Abbildung 3-4 veranschaulicht das am Beispiel der VSR-Systemtestumgebung.

***Abb. 3–4***
*Beispielhafte Systemtestumgebung*




*Der Systemtest erfordert eine separate Testumgebung*

Um Kosten und Aufwand zu sparen, wird oft der Fehler begangen, den Systemtest statt in einer separaten Testumgebung in der Produktivumgebung des Kunden durchzuführen. Dies ist aus folgenden Gründen schädlich:

- Im Systemtest werden Fehlerwirkungen auftreten. Dabei besteht immer die Gefahr, dass die Produktivumgebung des Kunden beeinträchtigt wird. Teure Systemausfälle und Datenverluste im produktiven Kundensystem können die Folge sein.
- Die Tester haben keine oder nur geringe Kontrolle über Parameter und Konfiguration der Produktivumgebung. Durch den gleichzeitig zum Test weiterlaufenden Betrieb der anderen Kundensysteme werden die Testbedingungen unter Umständen schleichend verändert. Die durchgeführten Systemtests sind schwer oder nicht mehr reproduzierbar (vgl. Kap. 3.6.3 →Regressionstest).

*Der Systemtestaufwand wird oft unterschätzt*

Der Aufwand für einen hinreichenden Systemtest ist u. a. wegen der komplexen Testumgebung nicht zu unterschätzen. [Bourne 97] nennt als Erfahrungswert, dass bei Beginn des Systemtests erst die Hälfte der Test- und Qualitätssicherungsarbeiten absolviert sind (insbesondere, wenn wie im Fallbeispiel ein Client/Server-System entwickelt wird).

### 3.4.3 Testziele

Wie oben beschrieben, ist es Ziel des Systemtests zu validieren, ob und wie gut das fertige System die an das System gestellten Anforderungen erfüllt. Aus Sicht des Tests müssen auf jeder Teststufe zwei Klassen von Anforderungen unterschieden werden:

- →**Funktionale Anforderungen** spezifizieren das Verhalten, welches das System oder Systemteile erbringen müssen. Sie beschreiben »was« das (Teil)System leisten soll. Ihre Umsetzung ist Voraussetzung dafür, dass das System überhaupt einsetzbar ist. Merkmale der Funktionalität nach [ISO 9126] sind: Angemessenheit, Richtigkeit, Interoperabilität, Ordnungsmäßigkeit, Sicherheit.
- →**Nicht funktionale Anforderungen** beschreiben Attribute des funktionalen Verhaltens, also »wie gut« bzw. mit welcher Qualität das (Teil)System seine Funktion erbringen soll. Ihre Umsetzung beeinflusst stark, wie zufrieden der Kunde bzw. Anwender mit dem Produkt ist und wie gerne er es einsetzt. Merkmale nach [ISO 9126] sind: Zuverlässigkeit, Benutzbarkeit, Effizienz. Indirekt haben auch die Änderbarkeit und Übertragbarkeit Einfluss auf die Kundenzufriedenheit. Je einfacher und schneller sich ein System den geänderten Anforderungen anpassen lässt, desto zufriedener wird der Kunde und Anwender sein.

Zum Test beider Arten von Anforderungen werden unterschiedliche Teststrategien angewendet. Die beiden folgenden Abschnitte erklären diese anhand des Systemtests genauer. Die entsprechenden Vorgehensweisen können aber auf allen Teststufen relevant sein. Der Test nicht funktionaler Anforderungen ist aber meist Gegenstand des Systemtests.

### 3.4.4 Test funktionaler Anforderungen

*Anforderungsbasiertes Testen*

Die Anforderungen des Auftraggebers an ein neues Softwaresystem, werden im Rahmen der Projektphase »Anforderungsdefinition« gesammelt und in einem Anforderungsdokument (oft auch als Lasten- oder Pflichtenheft bezeichnet; eine Gliederung liefert z. B. [VDE 3694]) niedergeschrieben. Wird das Projekt konsequent nach dem V-Modell abgewickelt, dann ist das Anforderungsdokument z. B. durch ein Review verifiziert und anschließend freigegeben worden. Üblicherweise wird es damit Bestandteil der Verträge zwischen Kunde und Softwarehersteller.

***Beispiel: Anforderungen an das VSR-System***

Der folgende Text zeigt einen Ausschnitt aus dem Anforderungsdokument für das System VSR zum Thema Preisberechnung (s. a. Kap 3.2.3):

A 100: Der Anwender kann ein Fahrzeugmodell aus dem jeweils aktuellen Modellprogramm zur Konfiguration auswählen.

A 101: Für ein ausgewähltes Modell wird die lieferbare Sonderausstattung angezeigt. Aus dieser Sonderausstattung kann die individuelle Wunschausstattung zusammengestellt werden.

A 102: Zu der jeweils ausgewählten Konfiguration wird laufend der Gesamtpreis gemäß aktueller Preisliste berechnet und angezeigt.

*Anforderungsdokument als Testbasis*

Beim →anforderungsbasierten Testen wird das freigegebene Anforderungsdokument als Testbasis herangezogen. Zu jeder Anforderung wird mindestens ein Systemtestfall abgeleitet und in der Systemtestspezifikation dokumentiert. Auch die Systemtestspezifikation wird durch ein Review verifiziert. Tests der Anforderung 102 aus obigem Beispiel könnten folgendermaßen aussehen:

***Beispiel: anforderungsbasiertes Testen***

T 102.1: Ein Modell wird ausgewählt; dessen Grundpreis gemäß Verkaufshandbuch wird angezeigt.

T 102.2: Eine Sonderausstattung wird ausgewählt, der Preis erhöht sich um den Preis dieses Zubehörteils.

T 102.3: Eine Sonderausstattung wird abgewählt, der Preis sinkt entsprechend.

T 102.4: Es werden drei Sonderausstattungen ausgewählt; die Rabattierung erfolgt gemäß Spezifikation ...

In der Regel wird mehr als nur ein Testfall benötigt, um eine Anforderung zu testen. Im Beispiel sind in Anforderung 102 eine Reihe von Varianten der Preisberechnug enthalten, die durch eine Abfolge von Testfällen (102.1–102.4) abgedeckt werden müssen. Natürlich können diese Beispieltestfälle noch beliebig verfeinert und ergänzt werden. Entscheidend ist: Sind die einmal definierten Testfälle (bzw. eine in der Testspezifikation definierte Mindestanzahl davon) im Systemtest fehlerfrei gelaufen, wird das System als validiert betrachtet.

***Beispiel: Geschäftsprozess-basiertes Testen***

Aus Sicht des Händlers unterstützt *VSR* ihn beim Verkaufsprozess. Dieser kann beispielsweise so aussehen:

- Der Kunde wählt aus den verfügbaren Modellen einen Typ aus, für den er sich interessiert.
- Er informiert sich zu diesem Typ über Ausstattungen und Preise und entscheidet sich für seinen »Wunschwagen«.
- Der Verkäufer schlägt Finanzierungsalternativen vor.
- Der Kunde entscheidet sich und schließt den Kaufvertrag ab.

Wenn ein Softwaresystem wie in obigem Beispiel den Zweck hat, einen bestimmten Geschäftsprozess des Kunden zu automatisieren oder zu unterstützen, ist →geschäftsprozessbasiertes Testen eine gut geeignete Systemteststrategie. Eine Geschäftsprozessanalyse (meist als Teil der Anforderungsanalyse erstellt) zeigt, welche Geschäftsprozesse relevant sind, wie häufig und in welchem Kontext sie auftreten, welche Personen, Firmen, Fremdsysteme daran beteiligt sind usw. Anschließend werden auf Grundlage dieser Analyse als Testbasis Testszenarien aufgestellt, die typische Geschäftsvorfälle nachbilden. Die Priorität der Testszenarien richtet sich nach Häufigkeit und Relevanz der entsprechenden Geschäftsprozesse.

Während beim anforderungsbasierten Testen einzelne Systemfunktionen im Blickpunkt stehen (z. B. Bestellung übertragen), sind dies beim geschäftsprozessbasierten Test Abläufe (z. B. Verkaufsgespräch bestehend aus Konfigurieren, Abschluss des Kaufvertrags und Übertragung der Bestellung), also hintereinander geschaltete Tests.

*Anwendungsfallbasiertes Testen*

Eine weitere Möglichkeit, Systemtestfälle abzuleiten, besteht in der Überlegung, wie der Anwender mit dem System umgeht und welche Aktionen er dann typischerweise ausführt (→anwendungsfallbasiertes Testen). Verschiedene Anwendergruppen (im Beispiel: Käufer Kleinwagen, Käufer Mittelklasse, Nicht-Käufer usw.) besitzen jeweils ihre eigenen Benutzerprofile. Das heißt, für sie lassen sich typische Aktionsmuster oder →Anwendungsfälle (so genannte *use cases*) in typischer Häufigkeit identifizieren.

Aus diesen Aktionsmustern lassen sich wieder Testszenarien ableiten. Anhand der Häufigkeit, mit der die entsprechenden Aktionen im späteren Betrieb der Software ausgelöst werden, ermittelt der Tester, wie wichtig das zugehörige Testszenario ist und mit welcher Priorität es deshalb im Testplan aufgenommen werden sollte.

Für den Anwender des Beispielsystems *VirtualShowRoom* ist natürlich nicht nur interessant, ob er Autos auswählen und kaufen kann. Für die Akzeptanz ist oft entscheidender, wie gut er mit dem System zu-

rechtkommt. Dies hängt zum Beispiel davon ab, ob die Software leicht zu bedienen ist, ob sie schnell genug »antwortet« und übersichtliche Ausdrucke liefert. Neben funktionalen Kriterien müssen im Systemtest deshalb auch die nicht funktionalen Eigenschaften geprüft und validiert werden.

### 3.4.5 Test nicht funktionaler Anforderungen

Genauso wichtig und berechtigt wie funktionale Anforderungen sind nicht funktionale Anforderungen, denn sie legen wichtige qualitative Aspekte der Systemleistung fest. Nach [Myers 82] und [Wallmüller 90] sollten folgende nicht funktionale Systemeigenschaften in entsprechenden Tests (in der Regel im Systemtest) berücksichtigt werden:

- →**Lasttest**: Messung des Systemverhaltens in Abhängigkeit steigender Systemlast (z. B. Anzahl parallel arbeitender Anwender, Anzahl Transaktionen).
- →**Performanztest**: Messung der Verarbeitungsgeschwindigkeit bzw. Antwortzeit für bestimmte Anwendungsfälle, in der Regel in Abhängigkeit steigender Last.
- →**Volumen-/Massentest**: Beobachtung des Systemverhaltens in Abhängigkeit zur Datenmenge (z. B. Verarbeitung sehr großer Dateien).
- →**Stresstest**: Beobachtung des Systemverhaltens bei Überlastung.
- **Test der (Daten)Sicherheit** gegen unberechtigten Systemzugang oder Datenzugriff.
- **Test der Stabilität/Zuverlässigkeit** im Dauerbetrieb (z. B. Ausfälle pro Betriebsstunde bei gegebenem Benutzungsprofil).
- **Test auf Robustheit** gegenüber Fehlbedienung, Fehlprogrammierung, Hardwareausfall usw. sowie Prüfung der Fehlerbehandlung und des Wiederanlaufverhaltens (*recovery*).
- **Test auf Kompatibilität/Datenkonversion**: Prüfung der Verträglichkeit mit vorhandenen Systemen. Import/Export von Datenbeständen usw.
- **Test unterschiedlicher Konfigurationen** des Systems, z. B. unterschiedliche Betriebssystemversion, Landessprache, Hardwareplattform usw.
- **Test auf Benutzungsfreundlichkeit**: Prüfung der Angemessenheit der Bedienung; Verständlichkeit der Systemausgaben usw., jeweils bezogen auf die Bedürfnisse einer bestimmten Anwendergruppe.
- **Prüfung der Dokumentation** auf Übereinstimmung mit dem Systemverhalten (z.B. Bedienungsanleitung und GUI).

- **Prüfung auf Änderbarkeit/Wartbarkeit:** Verständlichkeit und Aktualität der Entwicklungsdokumente; modulare Systemstruktur usw.

Beim Test nicht funktionaler Anforderungen stellt sich meist das Problem, dass diese lückenhaft und »schwammig« formuliert sind. Formulierungen wie »Das System soll leicht bedienbar sein« oder »schnell reagieren« sind in dieser Form nicht testbar.

***Tipp***

- Vertreter der (späteren) Systemtests sollten am Review des Anforderungsdokuments teilnehmen und darauf achten, dass jede nicht funktionale Anforderung messbar und damit testbar formuliert wird.

Des Weiteren gelten viele nicht funktionale Anforderungen als derart selbstverständlich, dass niemand mehr auf die Idee kommt, sie im Anforderungsdokument erwähnen zu müssen (vorausgesetzte Anforderungen). Auch solche nicht spezifizierten, aber dennoch relevanten Eigenschaften gilt es zu validieren.

***Beispiel: vorausgesetzte Anforderungen***

Das *VSR*-System ist für den Ablauf unter dem Betriebssystem eines marktführenden Herstellers konzipiert. Es ist selbstverständlich, dass die *VSR*-Bedienoberfläche in Aussehen und Bedienmechanismus (*look & feel*) dem Stil und den Konventionen folgt, die für Anwendungen in dieser Betriebssystemumgebung allgemein üblich oder empfohlen sind. Die *DreamCar*-Bedienoberfläche (vgl. Abb. 3-3) verletzt diese Richtlinien an einigen Punkten. Auch wenn keine entsprechende Anforderung formuliert wurde, können und sollen solche Abweichungen von selbstverständlichen Konventionen als Mangel bzw. Defekt gewertet werden.

***Exkurs: Test nicht funktionaler Eigenschaften***

Zum Test nicht funktionaler Eigenschaften wird zweckmäßigerweise auf vorhandene funktionale Tests zurückgegriffen. Ein eleganter allgemeiner Testansatz ist folgender:

Aus den funktionalen Tests werden solche Szenarien ausgewählt, die einen Querschnitt durch die Funktionalität des Gesamtsystems repräsentieren. Die zu testende nicht funktionale Größe muss im betreffenden Testszenario beobachtbar sein. Beim Ablauf des Testszenarios wird dann die nicht funktionale Größe gemessen. Liegt der gemessene Wert unter einem vorgegebenen Grenzwert, gilt der Test als bestanden. Das funktionale Testszenario dient praktisch als Messvorschrift zur Ermittlung der zu untersuchenden nicht funktionalen Systemeigenschaft.

### 3.4.6 Probleme in der Systemtestpraxis

***Exkurs*** In (zu) vielen Projekten geschieht die schriftliche Dokumentation der Anforderungen nicht oder nur sehr lückenhaft. Die Tester haben dann das Problem, dass mehr oder weniger unklar ist, was als korrektes Sollverhalten des Systems anzusehen ist. Demzufolge fällt es schwer, Fehlerzustände dingfest zu machen.

*Unklare Kundenanforderungen*

Wo keine Anforderungen existieren ist zunächst jedes Systemverhalten zulässig bzw. nicht bewertbar. Natürlich wird der Anwender oder Kunde eine gewisse Vorstellung davon haben, was er von »seinem« Softwaresystem erwartet. Es existieren also sehr wohl Anforderungen. Nur sind diese eben nirgends nachlesbar, sondern sie sind nur »in den Köpfen« einiger am Projekt beteiligten Personen vorhanden. Den Testern fällt dann die undankbare Rolle zu, alle diese Informationen über das gewünschte Sollverhalten nachträglich zusammenzutragen.

*Versäumte Entscheidungen*

Wenn die Tester die ursprünglichen Anforderungen identifizieren, werden sie feststellen, dass in den Köpfen der verschiedenen Personen zu ein und derselben Sache ganz unterschiedliche Ansichten und Vorstellungen existieren. Da im Projekt versäumt wurde, die Anforderungen schriftlich zu dokumentieren, abzustimmen und freizugeben, ist dies nicht weiter verwunderlich. Der Systemtest muss also nicht nur Anforderungen zusammensammeln, sondern auch noch Klärungs- und Entscheidungsprozesse, die viele Monate unterblieben sind, zu einem eigentlich viel zu späten Zeitpunkt erzwingen. Dieses Zusammentragen der Informationen ist sehr zeit- und kostenintensiv. Test- und Fertigstellung des Systems werden mit Sicherheit verzögert.

*Projekte scheitern*

Wenn Anforderungen nicht dokumentiert sind, fehlen natürlich auch den Entwicklern klare Ziele. Die Wahrscheinlichkeit, dass das konstruierte System die impliziten Kundenanforderungen erfüllt, ist deshalb außerordentlich gering. Niemand kann ernsthaft hoffen, dass unter solchen Projektbedingungen ein auch nur halbwegs brauchbares System entsteht. In derart gelagerten Projekten kann der Systemtest oftmals nur das Scheitern des Projekts »offiziell« attestieren.

## 3.5 Abnahmetest

Bei den bisher beschriebenen Teststufen handelt es sich um Testarbeiten, die in Verantwortung des Herstellers oder der entwickelnden Projektgruppe durchgeführt werden, bevor die Software an den jeweiligen Kunden oder Nutzer übergeben wird.

Vor Inbetriebnahme der Software (insbesondere wenn kundenspezifische →Individualsoftware erstellt wurde) erfolgt nun als abschließender Test noch ein so genannter Abnahmetest. Hierbei stehen die

Sicht und das Urteil des Kunden bzw. Anwenders im Vordergrund. Der Abnahmetest ist unter Umständen der einzige Test, den der Kunde nachvollziehen kann oder an dem er direkt beteiligt ist.

### 3.5.1 Test auf vertragliche Akzeptanz

Auf Basis der Ergebnisse des Abnahmetests entscheidet der Kunde, ob er das bestellte Softwaresystem als mangelfrei betrachtet und den Entwicklungsvertrag bzw. die vertraglich geschuldete Leistung als erfüllt ansieht. Dies kann auch ein mehr oder weniger formal gestalteter Vertrag oder Projektauftrag sein, der zwischen beauftragender Fachabteilung und realisierender IT-Abteilung einer Firma oder eines Konzerns besteht.

*Die Abnahme ist eine spezielle Form des Systemtests*

Der Abnahmetest ist also ein Test auf vertragliche Akzeptanz. Als Testkriterien gelten demnach die im Entwicklungsvertrag festgeschriebenen Abnahmekriterien, die deshalb klar und eindeutig formuliert werden müssen.

In der Praxis wird natürlich der Softwarehersteller schon in seinem eigenen Systemtest diese Abnahmekriterien auf Erfüllung prüfen und entsprechende Testfälle vorsehen. Für den Abnahmetest reicht es dann aus, die gemäß Vertrag abnahmerelevanten Testfälle zu wiederholen und so dem Kunden zu demonstrieren, dass die Akzeptanzkriterien des Vertrags erfüllt sind.

*Abnahmetest beim Kunden*

Im Gegensatz zum Systemtest, der in der Systemtestumgebung des Herstellers stattfindet, werden die Abnahmetests in der Abnahmeumgebung des Kunden durchgeführt. Wegen der unterschiedlichen Testumgebungen kann ein Testfall in der Abnahme durchaus fehlschlagen, der im Systemtest nie Probleme bereitet hat. Die Abnahmeumgebung soll so weit wie möglich der späteren Produktivumgebung entsprechen. Eine Testdurchführung in der Produktivumgebung selbst ist allerdings zu vermeiden, um den produktiven Betrieb laufender Softwaresysteme nicht zu gefährden.

Zur Ermittlung geeigneter Abnahmetests oder Abnahmekriterien können dieselben Methoden herangezogen werden wie für die Testfallermittlung im Systemtest. Nach [Wallmüller 90] sind insbesondere auch Geschäftsvorfälle einer typischen Zeit- oder Abrechnungsperiode (z. B. Monatsabschluss) zu berücksichtigen.

### 3.5.2 Test auf Benutzerakzeptanz

Ein weiterer Aspekt im Rahmen der Abnahme – als letzte Stufe der Validierung – ist der Test auf Benutzerakzeptanz. Ein solcher Test ist im-

mer dann zu empfehlen, wenn Kunde und Anwender des Systems verschiedene Personen(gruppen) sind.

***Beispiel: Unterschiedliche Benutzergruppen***

Im Beispiel *VSR* ist der für die Softwareentwicklung verantwortliche Auftraggeber der Automobilkonzern. Eingesetzt wird das System aber bei den Händlern dieses Konzerns. Als Endanwender benutzen das System dort Mitarbeiter der Händler und Endverbraucher, die bei ihrem Händler ein Fahrzeug kaufen möchten. Auch in der Konzernzentrale werden Sachbearbeiter mit dem System arbeiten (z. B. um neue Preislisten in das System einzuspielen).

*Akzeptanz jeder Anwendergruppe sicherstellen*

Die unterschiedlichen Anwendergruppen haben in der Regel ganz verschiedene Erwartungen an das neue System. Wenn eine Anwendergruppe das System ablehnt, z. B. weil es als »umständlich« empfunden wird, kann dies das Scheitern der gesamten Systemeinführung zur Folge haben, obwohl das System funktional vollkommen in Ordnung ist. Deshalb ist es wichtig, für jede Anwendergruppe Tests auf Benutzerakzeptanz vorzusehen. Diese Tests werden meist vom Kunden selbst organisiert, der auch die Testfälle auswählt (basierend auf seinen Geschäftsprozessen und typischen Anwendungsszenarien).

*Prototypen frühzeitig den Anwendern vorstellen*

Wenn im Abnahmetest gravierende Akzeptanzprobleme sichtbar werden, ist es für über Kosmetik hinausgehende Gegenmaßnahmen allerdings oft zu spät. Um solchen Desastern vorzubeugen, ist es vernünftig, schon in frühen Projektphasen Prototypen durch repräsentativ ausgewählte Vertreter der späteren Anwender begutachten zu lassen.

### 3.5.3 Feldtest

Soll die zu liefernde Software in sehr vielen verschiedenen Produktivumgebungen betrieben werden, ist es für den Softwarehersteller sehr kostenintensiv oder gar unmöglich, im Systemtest jede dieser Produktivumgebungen mit einer entsprechende Testumgebung nachzubilden.

In solchen Fällen wird der Softwarehersteller dem Systemtest einen so genannten →Feldtest nachschalten. Ziel des Feldtests ist, Einflüsse aus nicht vollständig bekannten oder nicht spezifizierten Produktivumgebungen zu erkennen und ggf. zu beheben.

*Test durch repräsentative Kunden*

Der Hersteller liefert hierzu stabile Vorabversionen der Software an einen ausgewählten Kundenkreis, der den Markt für die Software gut repräsentiert oder dessen Produktivumgebungen die verschiedenen möglichen Umgebungen gut abdecken.

Diese ausgewählten Kunden führen dann entweder vom Hersteller vorgegebene Testszenarien durch oder sie setzen das vorläufige Pro-

dukt probehalber unter realistischen Bedingungen ein. Anschließend geben sie ihre Fehlermeldungen (aber auch allgemeine Kommentare und Eindrücke über das neue Produkt) an den Hersteller zurück. Dieser kann dann entsprechende Anpassungen vornehmen.

*Alpha- und Beta-Tests*

Derartige Tests von Vorabversionen durch repräsentative Kunden werden oft auch als →Alpha-Test oder →Beta-Test bezeichnet. Alpha-Tests finden dabei beim Hersteller statt, Beta-Tests beim Kunden.

Ein Feldtest darf einen hausinternen Systemtest des Herstellers natürlich nicht ersetzen (auch wenn einige Hersteller dies vielleicht so sehen). Erst wenn der Systemtest nachgewiesen hat, dass die Software hinreichend stabil ist, sollte das Produkt dem Endkunden für einen Feldtest zugemutet werden.

## 3.6 Test neuer Produktversionen

Bisher wurde stillschweigend davon ausgegangen, dass ein Softwareentwicklungsvorhaben mit bestandenem Abnahmetest und der Auslieferung des neuen Produkts beendet ist. Die Realität sieht anders aus. Mit der erstmaligen Auslieferung steht ein Softwareprodukt erst am Anfang seines Lebenszyklus. Einmal installiert, ist es oft Jahre oder Jahrzehnte lang im Einsatz und wird während dieser Zeitspanne vielfach korrigiert, geändert und erweitert. Jedesmal entsteht eine neue →Version des ursprünglichen Produkts. Dieses Kapitel erläutert, was beim Test solcher neuen Produktversionen zu beachten ist.

### 3.6.1 Softwarewartung

Im Gegensatz zu »klassischen« Industrieprodukten geht es bei der Softwarewartung nicht darum, durch regelmäßige Pflege die Einsatzfähigkeit zu erhalten oder Schäden, die z. B. durch Abnutzung entstehen, zu reparieren. Denn Software altert nicht. Von Softwarewartung wird gesprochen, wenn ein Produkt an geänderte Einsatzbedingungen angepasst wird (Softwarepflege) oder wenn Defekte beseitigt werden, die schon immer im Produkt enthalten waren (Softwarewartung im engeren Sinne).

---

***Beispiel: Fehlerauswertung der VSR-Hotline***

Das *VSR*-System wurde nach intensiven Tests ausgeliefert und im Händlernetz installiert. Um herauszufinden, wo das System dennoch Schwächen hat, erstellt die zentrale Hotline eine Auswertung aller aus dem Feld eingegangenen Hotline-Anfragen. Hier einige Beispiele:

1. Einige wenige Händler betreiben das System auf einer nicht freigegebenen Plattform mit veraltetem Betriebssystemstand. Hier kommt es zu sporadischen Systemabstürzen beim Hostzugriff.
2. Die Auswahl der Sonderausstattung empfinden viele Kunden als umständlich, insbesondere wenn sie Preisvergleiche zwischen verschiedenen Ausstattungspaketen vornehmen wollen. Viele Anwender wünschen sich deshalb, dass sie Ausstattungskombinationen zwischenspeichern und nach Änderung wieder zurückholen können.
3. Einige seltene Versicherungstarife können nicht gerechnet werden, da vergessen wurde, die entsprechende Kalkulationsvorschrift in die Versicherungskomponente einzubauen.
4. Nach einer Fahrzeugbestellung ist mitunter auch nach 15 Minuten kein Bestelleingang vom Werkrechner bestätigt. Das System trennt die Verbindung nach spätestens 15 Minuten, um offene unbenutzte Verbindungen zu vermeiden. Die Kunden reagieren verärgert, weil sie unnütz auf eine Bestellbestätigung warten mussten. Der Händler muss den Vorgang wiederholen und dem Kunden die Bestätigung nachsenden.

Problem 1 hat eigentlich der Händler zu vertreten, der das System auf einer nicht vorgesehenen Plattform betreibt. Dennoch wird der Softwarehersteller hier unter Umständen nachbessern, vielleicht um betroffenen Händlern eine teure Hardwareaufrüstung zu ersparen.

Problem 2 wird immer vorkommen, egal wie gut und umfassend die Anforderungen ursprünglich erhoben wurden. Einfach deshalb, weil erst das laufende System viele neue Erfahrungen liefert und damit zwangsläufig neue Wünsche generiert.

*Testplan ergänzen*

Problem 3 hätte im Systemtest gefunden werden können. Aber Testen kann Fehlerfreiheit nicht garantieren, sondern ist immer nur eine Stichprobe, in der Fehlerwirkungen mit einer gewissen Wahrscheinlichkeit aufgedeckt werden. Ein guter Testmanager wird hier prüfen, durch welche Tests diese Fehlerwirkung gefunden worden wäre und seinen Testplan ergänzen oder anpassen.

Problem 4 wurde im Integrationstest erkannt und behoben. Das *VSR*-System wartet ohne abzubrechen auch über 15 Minuten auf eine Bestellbestätigung des Werkrechners. Die lange Wartezeit entsteht in seltenen Fällen, wenn im Hostrechner bestimmte Batchprozesse ablaufen. Dass der Kunde nicht so lange im Geschäft warten will, ist eine andere Sache.

---

Die vier Beispiele oben stehen stellvertretend für typische Probleme, die auch beim ausgereiftesten Softwaresystem nach einer gewissen Einsatzzeit auftreten:

*Typische Wartungsanlässe*

1. Das System wird unter neuen, unvorhergesehenen, nicht geplanten Einsatzbedingungen betrieben.
2. Neue Kundenwünsche werden geäußert.
3. Funktionen für seltene und deshalb vergessene Sonderfälle werden benötigt.
4. Es werden Ausfälle beobachtet, die sporadisch oder erst nach sehr langer Betriebszeit auftreten. Diese sind oft durch externe Einflüsse verursacht.

Jedes Softwaresystem bedarf also nach seiner Auslieferung gewisser Korrekturen und Ergänzungen. In diesem Zusammenhang wird von Softwarewartung und Softwarepflege gesprochen. Die Tatsache, dass Wartung auf jeden Fall notwendig ist, darf aber nicht als Argument missbraucht werden, um bei Komponenten-, Integrations- oder Systemtests zu sparen. Nach dem Motto: »Wir müssen ja sowieso immer wieder neue Versionen rausbringen; also ist es nicht so schlimm, wenn wir es mit dem Testen nicht so genau nehmen und Defekte übersehen.«

### 3.6.2 Geplante Weiterentwicklung

Außer den durch Mängel und Fehler ausgelösten Wartungsarbeiten gibt es auch Änderungs- und Erweiterungsarbeiten, die das Projektmanagement von Anfang an vorgesehen hat.

***Beispiel: VSR-Entwicklungsplanung***

Im *VSR*-Entwicklungsplan für Release 2 sind u. a. folgende Arbeiten vorgesehen:

1. Da der Host im Konzernrechenzentrum neue Kommunikationssoftware erhält, muss in Folge das Kommunikationsmodul des *VSR* angepasst werden.
2. Verschiedene Systemerweiterungen, die zum ersten Liefertermin nicht fertig gestellt werden konnten, werden nun in Release 2 mitgeliefert.
3. Die Installationsbasis soll auf das europäische Händlernetz ausgedehnt werden. Dazu müssen Land für Land landesspezifische Anpassungen eingebaut und sämtliche Bedientexte und Handbücher übersetzt werden.

Diese drei Arbeitspunkte sind weder durch Defekte noch durch unvorhergesehene Anwenderwünsche verursacht. Sie zählen demnach nicht zur Wartung, sondern gehören zur normalen Produktweiterentwicklung.

Punkt 1 resultiert aus einer geplanten Änderung eines Nachbarsystems. Punkt 2 ist eine von Anfang an vorgesehene Funktionalität, die aus Zeitgründen nicht zur Systemeinführung geliefert werden konnte.

Punkt 3 steht für Erweiterungen, die im Zuge einer geplanten Marktausdehnung notwendig werden.

Ein Softwareprojekt ist also keinesfalls mit der Lieferung der ersten Produktversion abgeschlossen. Vielmehr findet eine kontinuierliche Weiterentwicklung statt, die z. B. jährlich verbesserte Produktversionen auf den Markt bringt. Zweckmäßigerweise werden diese Lieferungen mit den Wartungsarbeiten synchronisiert, so dass z. B. halbjährlich eine neue Version erscheint: ein Wartungsupdate und ein echtes funktionales Update.

Nach jeder Auslieferung beginnt das Projekt quasi mit einem neuen Durchlauf durch alle Projektphasen. Dieses Vorgehen wird deshalb auch iterative Softwareentwicklung genannt. Iterative Softwareentwicklung ist heute nicht die Ausnahme, sondern der Regelfall[11].

Wie muss das Testen darauf reagieren? Müssen alle Teststufen in vollem Umfang ihre Tests bei jedem →Release des Produkts komplett wiederholen? Diese Frage nach Regressionstests wird im nächsten Abschnitt behandelt.

### 3.6.3 Regressionstest

Sowohl durch Wartungsarbeiten als auch bei Weiterentwicklung werden Teile vorhandener Software geändert oder neue Softwarebausteine ergänzt. In beiden Fällen muss die geänderte Software erneut getestet werden. Diese Tests heißen Regressionstests.

*Regressionstest*

Der Regressionstest ist ein erneuter Test eines bereits getesteten Programms nach dessen Modifikation mit dem Ziel nachzuweisen, dass durch die vorgenommenen Änderungen keine neuen Defekte eingebaut oder bisher maskierte Fehlerzustände freigelegt wurden.

Die Frage, die sich stellt, ist, was aufgrund einer Änderung getestet werden muss, d. h., wie umfangreich ein Regressionstest sein sollte. Folgende Möglichkeiten lassen sich abgrenzen:

*Umfang des Regressionstests*

1. Wiederholung aller Tests, die Fehlerwirkungen erzeugt haben, deren Ursache der korrigierte Defekt war (→Fehlernachtest)?
2. Test aller Programmstellen, an denen korrigiert oder geändert wurde (Test geänderter Funktionalität)?
3. Test aller Programmteile oder Bausteine, die neu eingefügt wurden (Test neuer Funktionalität)?
4. Das komplette System (vollständiger Regressionstest)?

---

11. Im allgemeinen V-Modell ist dieser Aspekt nicht dargestellt. Erst modernere Vorgehensmodelle modellieren Iterationen explizit (hierzu s. a. [Sneed 02]).

Sowohl ein Fehlernachtest (1) als auch Tests nur am »Ort« der Modifikation (2 und 3) sind zu wenig. Denn in Softwaresystemen können scheinbar simple lokale Änderungen unerwartete Auswirkungen und Seiteneffekte auf beliebige andere (auch weit entfernte) Systemteile haben.

*Änderungen können unerwartete Seiteneffekte haben*

Wenn also nur die Codeteile getestet werden, die geändert oder ergänzt wurden, wird versäumt zu testen, welche Konsequenzen diese Änderungen auf unveränderte Teile haben. Wo solche unerwünschten Konsequenzen auftreten könnten, lässt sich aber leider mit vertretbarem Aufwand nur grob abschätzen. Besonders schwierig ist dies, wenn Systeme geändert werden, die unzureichend dokumentiert sind, oder Spezifikationen völlig fehlen. Bei der Wartung von Altsystemen ist dies leider oft der Fall.

*Vollständiger Regressionstest*

Zusätzlich zum Test korrigierter Fehlerzustände und zusätzlich zum Test geänderter Funktionen müssten eigentlich alle vorhandenen Testfälle wiederholt werden. Erst dann hätte der Test die gleiche Aussagekraft wie der entsprechende Test, der an der Ausgangsversion des Programms durchgeführt wurde. Ein solcher vollständiger Regressionstest müsste ebenfalls durchgeführt werden, wenn die Systemumgebung geändert wurde, da dies potenziell auf jedes Systemteil Rückwirkungen haben kann.

In der Praxis ist ein vollständiger Regressionstest aber fast immer zu zeit- und kostenintensiv. Gesucht sind demnach Kriterien, anhand derer entschieden werden kann, welche alten Testfälle ohne zu großen Informationsverlust weggelassen werden können. Wie immer beim Testen ist dies eine Abwägung zwischen niedrigeren Kosten und höherem Risiko. Folgende Auswahlstrategien werden häufig angewendet:

*Auswahl von Regressionstestfällen*

- Wiederholung nur von derjenigen Tests aus dem Testplan, denen hohe Priorität zugeordnet ist.
- Bei funktionalen Tests Verzicht auf gewisse Varianten (Sonderfälle).
- Einschränkung der Tests auf bestimmte Konfigurationen (z. B. nur Test der englischsprachigen Produktversion, nur Test auf einer bestimmten Betriebssystemversion u. Ä.).
- Einschränkung der Tests auf bestimmte Teilsysteme oder Teststufen.

***Exkurs***

Die aufgelisteten Regeln beziehen sich im Wesentlichen auf den Systemtest. Auf niedrigeren Teststufen können Regressionstestkriterien auch auf Designunterlagen (z. B. Klassenhierarchie) oder White-box-Informationen basieren. Eine weitergehende Darstellung findet sich in [Sneed 02]. Dort behandeln die Autoren nicht nur spezielle Probleme beim Regressionstest von objektorientierten Programmen, sondern beschreiben auch die allgemeinen Grundsätze des Regressionstests sehr ausführlich.

## 3.7 Kosten- und Wirtschaftlichkeitsaspekte

Testen kann sehr aufwändig werden und ein bedeutender Kostenfaktor innerhalb eines Entwicklungsprojekts sein. Die Frage ist, wie viel Testaufwand ist für ein bestimmtes Softwareprojekt angemessen? Ab wann übersteigt der Aufwand den möglichen Nutzen?

Um dies zu beantworten, muss betrachtet werden, welche Fehlerkosten ein Verzicht auf Prüfungen und Tests nach sich zieht. Zwischen Fehlerkosten und Testkosten gilt es dann abzuwägen.

### 3.7.1 Fehlerkosten

Werden Prüfungen und Tests im Umfang reduziert oder ganz eingespart, erhöht sich als Folge die Zahl der unentdeckten Fehler und Mängel. Diese Fehler und Mängel verbleiben im Produkt und führen ggf. zu folgenden Kosten:

*Kosten durch Produktmängel*

- **Direkte Fehlerkosten**: Kosten, die dem Kunden durch Fehlerwirkungen beim Betrieb des Softwareprodukts entstehen (und für die der Hersteller evtl. haften muss). Zum Beispiel Kosten von Berechnungsfehlern (Datenverlust, Fehlbuchung, Schaden an Hardware oder Anlagenteilen, Personenschäden); Kosten wegen Ausfalls softwaregesteuerter Maschinen, Anlagen oder Geschäftsprozesse, Kosten durch Einspielen neuer Versionen ggf. verbunden mit Neueinweisung von Mitarbeitern usw. An diese Kosten denken die wenigsten, sie sind aber riesig, wenn bedacht wird, wie viel Zeit beim Einspielen einer neuen Version bei allen Kunden insgesamt anfällt.
- **Indirekte Fehlerkosten**: Kosten bzw. Umsatzverlust für den Hersteller, weil der Kunde mit dem Produkt unzufrieden ist. Zum Beispiel Vertragsstrafen oder Minderung wegen nicht erfüllten Vertrags, erhöhter Aufwand für Kundenhotline und Support, Imageschaden, Verlust des Kunden usw.
- **Fehlerkorrekturkosten**: Kosten, die dem Hersteller im Zuge der Fehlerkorrektur entstehen. Beispielsweise Zeit für Fehleranalyse und Korrektur, Zeit für Regressionstest, erneute Auslieferung und Installation, Nachschulung des Kunden, Verzug bei Neuprodukten wegen Bindung der Entwicklerkapazität im Wartungsbereich, sinkende Konkurrenzfähigkeit usw.

*Risikoanalyse*

Welche dieser Kostenarten eintreten, mit welcher Wahrscheinlichkeit und in welcher Höhe, wie hoch also das Fehlerkostenrisiko für ein Projekt tatsächlich ist, kann nur schwer ermittelt werden. Abhängig ist

dieses Risiko sicherlich von Art und Größe des Softwareprodukts, von Art und Branche des Kunden (Vertragsgestaltung, gesetzliche Rahmenbedingungen), von Art und Anzahl auftretender Ausfälle, von der Anzahl der betroffenen Produktinstallationen bzw. Endanwender. Große Unterschiede bestehen sicherlich auch zwischen Individualsoftware und →Standardsoftware. Im Zweifelsfall müssen alle diese Einflussfaktoren in einer projektspezifischen Risikoanalyse abgeschätzt werden.

*Fehler möglichst früh finden, senkt Kosten*

Unabhängig davon, wie hoch das Risiko eines Fehlers genau ausfällt, gilt: Wichtig ist, den Fehler möglichst früh nach seiner Entstehung zu finden. Denn Fehlerkosten steigen rapide über die Entwicklungsphasen an.

- Ein Fehler, der sehr früh entsteht (z. B. ein Fehler in der Anforderungsdefinition), kann, solange er unentdeckt bleibt, in den anschließenden Entwicklungsphasen viele Folgefehler produzieren (Multiplikation des Effekts des ursprünglichen Fehlers).
- Je später ein Fehler entdeckt wird, umso mehr Korrekturen sind notwendig. Unter Umständen müssen vorangegangene Phasen (Anforderungsdefinition, Design, Programmierung) zumindest teilweise wiederholt werden.

Als Richtwert kann davon ausgegangen werden, dass sich mit jeder Teststufe die Korrekturkosten, die ein Defekt verursacht, gegenüber der Vorphase in etwa verdoppeln. Entsprechende Untersuchungen finden sich in [Möller 96] oder [URL: NIST Report].

Ist die Software schon beim Kunden installiert, kommt zusätzlich das Risiko direkter und indirekter Fehlerkosten hinzu. Im Falle sicherheitskritischer Software (Steuerung von Anlagen, Verkehrsmitteln, medizintechnischen Geräten u. a.) können die potenziellen Kosten und Fehlerfolgen katastrophal sein. Zahlreiche gravierende Beispiele sind in [Thaller 00] oder [Wiener 94] dokumentiert.

### 3.7.2 Testkosten

Die wichtigste Maßnahme, das Fehlerrisiko zu senken oder wenigstens zu begrenzen, besteht darin, Prüfungen und Tests vorzusehen. Doch auch die Faktoren, die die Testkosten beeinflussen, sind vielfältig und in der Praxis schwer zu quantifizieren. Die folgende Liste zeigt die wichtigsten Einflussfaktoren, die ein Testmanager bei seiner Aufwandschätzung in Rechnung stellen sollte:

*Testkostenbeeinflussende Faktoren*

- **Reifegrad**[12] des Entwicklungsprozesses
  - Fehlerhäufigkeit der Entwickler
  - Rate von Softwareänderungen
  - Gültigkeit, Bestand und Aussagekraft von Plänen
- **→Testbarkeit der Software**
  - Aussagekraft und Aktualität der Dokumentation
  - Art der Software und Systemumgebung
  - Komplexität der Software (z. B. nach McCabe, s. Kap. 4.2.5)
- **Mitarbeiterqualifikation**
  - Erfahrung und Know-how der Tester bzgl. »Testen«
  - Erfahrung und Know-how der Tester bzgl. »Testobjekt«
  - Zusammenarbeit Tester-Entwickler-Management-Kunde
- **Qualitätsziele**
  - Angestrebte →Testabdeckung
  - Angestrebte Restfehlerrate bzw. Zuverlässigkeit nach dem Test
- **Teststrategie**
  - Anzahl und Umfang der Teststufen (Komponenten-, Integrations-, Systemtest ...)
  - Wahl der Testmethoden (Black-box- oder White-box-Verfahren)
  - Zeitliche Planung der Tests (Beginn und Durchführung der Testarbeiten im Projekt bzw. im Softwarelebenszyklus)

Nur wenige dieser Faktoren kann der Testmanager direkt beeinflussen. Aus seiner Sicht sieht die Situation folgendermaßen aus:

*Einfluss des Testmanagers*

- **Reifegrad** des Softwareentwicklungsprozesses: kurzfristig nicht zu beeinflussende, gegebene Größe; muss hingenommen werden, wie es ist; nur langfristig beeinflussbar durch Prozessverbesserungsprogramm.
- **Testbarkeit** der Software: hängt stark vom Reifegrad des Prozesses ab. Ein gut strukturierter Prozess mit entsprechenden Reviews führt zu besser strukturierter Software, die einfacher zu testen ist. Daher nur langfristig beeinflussbar durch Prozessverbesserungsprogramm.
- **Mitarbeiterqualifikation**: kurzfristig bedingt beeinflussbar durch Auswahl des Testpersonals; mittelfristig beeinflussbar durch Aus- und Weiterbildung.

12. Es gibt verschiedene Methoden zur Bewertung von Softwareentwicklungsprozessen. Eine vergleichende Darstellung findet sich z. B. in [Mellis 96].

- **Qualitätsziele**: vom Kunden vorgegeben; bedingt beeinflussbar (Priorisierung).
- **Teststrategie**: frei wählbar; die einzige Stellgröße, die der Testmanager auch kurzfristig beeinflussen und kontrollieren kann.

### 3.7.3 Konsequenzen für die Teststrategie

Der Auswahl einer angemessenen Teststrategie kommt also eine zentrale Bedeutung zu. Es gilt, einen Mix von Vorgehensweisen auszuwählen, der unter den gegebenen Randbedingungen (Reifegrad, Testbarkeit, Mitarbeiterqualifikation) ein optimales Verhältnis zwischen Testkosten, verfügbaren Ressourcen und drohenden Fehlerkosten darstellt.

Die Kosten des Testens sollen dabei natürlich deutlich niedriger bleiben als die Kosten, die durch nicht beseitigte Defekte und Mängel im Endprodukt verursacht würden. Datenmaterial, das es erlaubt, diese Kosten-Nutzen-Relation zu quantifizieren, besitzt jedoch kaum eine softwareentwickelnde Organisation.

*Wann soll mit Testen begonnen werden?*

Bezüglich der Frage, wann mit Prüfen und Testen begonnen werden soll, sprechen die Kostenbetrachtungen jedoch eine eindeutige Sprache:

- So früh im Projekt wie möglich und
- alle Phasen des Projekts kontinuierlich begleitend.

Das allgemeine V-Modell (vgl. Abb. 3-1) sieht deshalb am Ende jeder Entwicklungsphase eine Verifikation des jeweiligen Phasenergebnisses vor. Viele Analyse- und Designfehler können so früh gefunden und behoben werden. Die Teststufen im rechten Ast des Modells sind als Phasen der Testdurchführung zu verstehen. Die zugehörende Testvorbereitung (Testplanung, Testspezifikation und ggf. Testautomatisierung) startet früher und wird parallel zu den Entwicklungsschritten im linken Ast durchgeführt.

***Beispiel: VSR-Testplanung***

Im Projekt *VSR* wurde direkt nach Freigabe des Anforderungsdokuments mit der Testplanung und der Testspezifikation begonnen. Zu jeder Anforderung wurde mindestens ein Testfall entworfen. Die so entstandene noch sehr grobe Testspezifikation wurde einem Review unterzogen, an dem Vertreter des Kunden, der Entwicklung und des späteren Systemtests beteiligt waren. Als Ergebnis dieses Reviews der Testspezifikation wurde eine ganze Reihe von Anforderungen als »unklar« oder »lückenhaft« identifiziert. Ebenso wurden falsche oder unzureichende Testfälle aufgedeckt.

Allein das Aufschreiben sinnvoller Tests und die Diskussion darüber mit den Entwicklern haben somit auf viele Probleme aufmerksam gemacht, lange bevor der erste Test tatsächlich ausgeführt wurde.

## 3.8 Testplanung

Der erste Schritt in der Testvorbereitung besteht darin, eine für das jeweilige Projekt geeignete Teststrategie aufzustellen. Testen sollte aber nicht als einzige Maßnahme zur →Qualitätssicherung (QS) eingesetzt werden, sondern im Verbund mit anderen QS-Maßnahmen stehen. Die übergreifende Planung der qualitätssichernden Maßnahmen in einem Projekt wird im Qualitätssicherungsplan dokumentiert.

### 3.8.1 Qualitätssicherungsplan

Gliederungsschemata für einen Qualitätssicherungsplan liefert z. B. die Norm [IEEE 730] oder das Vorgehensmodell des Bundes und der Länder [URL: VM-scope]. Folgende Themen sind zu behandeln:

**Gliederung eines Qualitätssicherungsplans nach IEEE 730**

1 Beschreibung des Projekts
2 Qualitätsziele und Risiken im Projekt
2.1 Qualitätsziele für Produkte und Prozesse
2.2 Qualitätsrisiken
2.3 Maßnahmen aufgrund der Qualitätsziele und -risiken
3 QS-Maßnahmen gemäß Kritikalität und IT-Sicherheit
3.1 Verwendete Richtlinien und Normen
3.2 Einstufungsbedingte QS-Maßnahmen
4 Entwicklungsbegleitende Qualitätssicherung
4.1 Zu prüfende Produkte
4.2 Zu prüfende Aktivitäten
5 Spezifische Kontrollmaßnahmen
5.1 Eingangskontrolle von Fertigprodukten
5.2 Kontrolle von Unterauftragnehmern
5.3 Ausgangskontrolle der Softwarebausteine
6 Änderungskontrolle
7 Kontrolle von Bearbeitungskompetenzen
8 Kontrolle des Konfigurationsmanagements

Im Rahmen der QS-Planung wird auch festgelegt, welche Rolle Tests als spezielle, analytische QS-Maßnahme spielen. Details hierzu werden in der Testplanung festgelegt.

### 3.8.2 Testkonzept

Eine so umfangreiche Aufgabe wie das Testen erfordert eine sorgfältige Planung. Zu planen ist das Ineinandergreifen der einzelnen Teststufen, aber auch die Abläufe und Inhalte innerhalb der Teststufen. Die Grobplanung hierzu erfolgt möglichst früh im Softwareprojekt und wird im Testkonzept[13] dokumentiert. Dort beschreibt und begründet der Testmanager, wie er die anstehenden Testaufgaben lösen will, welchen Testumfang er plant, welche Werkzeuge eingesetzt werden sollen usw. Die internationale Norm [IEEE 829] stellt eine Referenzgliederung bereit:

**Testkonzept nach IEEE 829**

1. Testkonzeptbezeichnung
2. Einführung
3. Testobjekte
4. Zu testende Leistungsmerkmale
5. Leistungsmerkmale, die nicht getestet werden
6. Teststrategie
7. Abnahmekriterien
8. Kriterien für Testabbruch und Testfortsetzung
9. Testdokumentation
10. Testaufgaben
11. Testumgebung
12. Verantwortlichkeiten/Zuständigkeiten
13. Personal, Einarbeitung, Ausbildung
14. Zeitplan/Arbeitsplan
15. Planungsrisiken und Unvorhergesehenes
16. Genehmigung/Freigabe

Diese Gliederung hat sich in der Praxis bewährt. Die genannten Kapitel werden in dieser oder leicht modifizierter Form in vielen Projekten wiederzufinden sein. Eine detailliertere Charakterisierung der genanten Gliederungspunkte findet sich in Anhang A des Buches.

## 3.9 Zusammenfassung

- Das allgemeine V-Modell definiert grundlegende Teststufen: Komponententest, Integrationstest, Systemtest und Abnahmetest und

13. engl. *test plan*; nicht zu verwechseln mit der auf das Testkonzept aufbauenden zeitlichen Feinplanung (engl. *schedule*).

unterscheidet zwischen verifizierender und validierender Prüfung. Modernere Vorgehensmodelle haben diese Modellvorstellungen übernommen.

- Der Komponententest testet einzelne Softwarebausteine. Der Integrationstest prüft das Zusammenwirken solcher Bausteine. Funktionaler und nicht funktionaler Systemtest betrachten das Gesamtsystem aus Sicht des späteren Anwenders. Im Abnahmetest prüft der Auftraggeber das erstellte Produkt auf vertragliche Akzeptanz und Benutzerakzeptanz. Wenn das System in sehr vielen Produktivumgebungen betrieben werden soll, bieten Feldtests eine zusätzliche Möglichkeit, Einsatzerfahrungen mit Vorabversionen des Produkts zu sammeln.
- Durch Fehlerkorrekturen (Wartung) und geplante Weiterentwicklung (Pflege) wird ein Softwareprodukt im Laufe seines Lebenszyklus immer wieder geändert oder erweitert. Jede dieser geänderten Versionen muss erneut getestet werden. Welchen Umfang solche Regressionstests haben, muss eine individuelle Risikoabschätzung festlegen.
- Fehler und Mängel, die im Test übersehen werden und die im Produkt verbleiben, können hohe Fehlerkosten nach sich ziehen. Dabei wird unterschieden zwischen direkten Fehlerkosten, indirekten Fehlerkosten und Korrekturkosten. Bei der Wahl der Teststrategie wird ein optimales Verhältnis zwischen Testkosten, verfügbaren Ressourcen und drohenden Fehlerkosten angestrebt. Einfluss auf die Testkosten haben folgende Faktoren: Reifegrad des Softwareentwicklungsprozesses, Testbarkeit, Mitarbeiterqualifikation, Qualitätsziele und die gewählte Teststrategie.
- Im Testkonzept beschreibt und begründet der Testmanager, wie er die anstehenden Testaufgaben lösen will, welchen Testumfang er plant, welche Werkzeuge eingesetzt werden sollen usw. Die internationale Norm [IEEE 829] stellt eine Referenzgliederung bereit. Das Testkonzept muss mit dem übergeordneten Qualitätssicherungsplan abgestimmt sein.

# 4 Statischer Test

*Statische Untersuchungen, wie Reviews und werkzeuggestützte Codeanalysen, können sehr erfolgreich zur Qualitätssteigerung eingesetzt werden. In diesem Kapitel werden die einzelnen Möglichkeiten und Vorgehensweisen dafür vorgestellt.*

Eine oft unterschätzte Prüfmethode ist der so genannte statische Test, der oft auch als manueller Test, manuelles Prüfen oder statische Analyse bezeichnet wird. Im Gegensatz zum dynamischen Test (s. Kap. 5) wird das Testobjekt nicht mit Testdaten versehen und ausgeführt, sondern einer Analyse unterzogen. Diese kann in Form einer intensiven Betrachtung durch mehrere Personen erfolgen oder durch entsprechende Werkzeuge. Die Betrachtung kann bei allen Dokumenten, die während des Softwareentwicklungsprozesses erstellt wurden, durchgeführt werden, die werkzeuggestützte statische Analyse nur bei Dokumenten, die einer formalen Struktur unterliegen.

Ziel aller Untersuchungen ist die Ermittlung von Fehlern und Verstößen gegen die vorhandenen Spezifikationen, die einzuhaltenden Standards oder auch der Nachweis der Verletzung der Projektplanung. Die Ergebnisse der Untersuchungen werden darüber hinaus dazu benutzt, den Entwicklungsprozess zu optimieren. Grundidee ist die Prävention: Fehler und Abweichungen sollen so früh wie möglich erkannt werden, noch bevor sie im weiteren Verlauf der Softwareentwicklung zum Tragen kommen und aufwändige Nach- oder Verbesserungen nach sich ziehen.

## 4.1 Strukturierte Gruppenprüfungen

### 4.1.1 Grundlagen

*Gezielte Nutzung der menschlichen Analyse- und Denkfähigkeit*

Bei den nicht werkzeuggestützten Verfahren wird die menschliche Analyse- und Denkfähigkeit genutzt, um komplexe Sachverhalte zu prüfen und zu bewerten. Dies erfolgt durch intensives Lesen und Nachvollziehen der untersuchten Dokumente.

Es gibt verschiedene Vorgehensweisen zur Überprüfung der Dokumente, die sich durch die Intensität und die benötigten Ressourcen (Personen und Zeit) und die verfolgten Ziele unterscheiden. Im Weiteren werden die unterschiedlichen Verfahren näher erläutert. Leider ist keine einheitliche Begrifflichkeit bei den statischen Verfahren gegeben.

Die hier verwendeten Begriffe sind in Analogie zu den Begriffen in den Lehrplänen und [IEEE 1028] gewählt (s. a. Anhang Glossar). Ausführliche Beschreibungen sind in [Gilb 96], [Frühauf 91] und [Balzert 98] nachzulesen.

### 4.1.2 Reviews

Review[14] ist sowohl die Bezeichnung für ein bestimmtes Vorgehen bei der Prüfung von Dokumenten als auch ein gebräuchlicher Oberbegriff für die verschiedenen statischen Prüfverfahren, die von Personen durchgeführt werden. Ein weiterer Begriff, in meist analoger Bedeutung verwendet, ist →Inspektion. Alle Dokumente können einem Review oder einer Inspektion unterzogen werden, beispielsweise die Anforderungsbeschreibung, die Entwurfsspezifikationen, die Programmtexte, die Testpläne, das Benutzungshandbuch. Oft sind Reviews die einzige Möglichkeit, die Semantik der Dokumente zu überprüfen.

*Mittel zur Sicherung der Qualität*

Reviews sind ein effizientes Mittel zur Sicherung der Qualität der untersuchten Dokumente. Idealerweise sollten sie so früh wie möglich nach der Fertigstellung der Dokumente durchgeführt werden, um frühzeitig Fehler und Unstimmigkeiten festzustellen. So werden die verifizierenden Prüfungen bei Abschluss einer Phase im allgemeinen V-Modell in der Regel mittels Reviews geleistet. Die Beseitigung der Fehler und Unstimmigkeiten führt zu einer verbesserten Qualität der Dokumente und wirkt sich positiv auf den gesamten Entwicklungsprozess aus, da die Entwicklung mit Dokumenten fortgesetzt wird, die weniger oder sogar keine Fehler mehr enthalten.

*Positive Auswirkungen*

Neben der Reduzierung der Fehler haben Reviews noch folgende weitere positive Auswirkungen:

- Fehlerbeseitigung kostet Ressourcen. Wenn frühzeitig Fehler erkannt und beseitigt werden, führt dies zu einer Produktivitätssteigerung in der Entwicklung, da weniger Ressourcen für die Fehlererkennung und -beseitigung benötigt werden und diese somit der Entwicklung zur Verfügung stehen (s. Kap. 3.8).
- Dies hat auch zur Folge, dass sich die Entwicklungszeiträume verkürzen.
- Werden Fehler frühzeitig erkannt und behoben, führt dies zu einer Verringerung der Kosten und der benötigten Zeiten für den dyna-

---

14. **review**
(*survey*) Übersicht, *die* (**of** über + *Akk.*); Überblick, *der* (**of** über + *Akk.*); (*of past events*) Rückschau, *die* (**of** auf + *Akk.*); **be a review of sth.** einen Überblick *od.* eine Übersicht über etw. (*Akk.*) geben [Duden 99].

mischen Test (s. Kap. 5), da weniger Fehlerzustände im Testobjekt vorhanden sind.

- Durch die geringere Zahl an Fehlern und Ungenauigkeiten ist auch eine Kostenreduzierung während der Lebenszeit des Produkts zu erwarten. Beispielsweise können durch Reviews Ungenauigkeiten der Kundenwünsche in den Anforderungen aufgedeckt und geklärt werden. Absehbare Änderungswünsche nach der Inbetriebnahme des Softwaresystems können somit im Vorfeld verhindert werden.
- Es ist auch eine reduzierte Fehlerhäufigkeit im Einsatz des Systems zu erwarten.
- Da die Überprüfungen im Team durchgeführt werden, führt dies zu einem Wissensaustausch unter den beteiligten Personen. Das wird zur Verbesserung der Arbeitsmethoden der einzelnen Personen führen und somit die Qualität der nachfolgenden Produkte verbessern.
- Da mehrere Personen an einem Review beteiligt sind, ist eine klare und verständliche Darstellung der Sachverhalte notwendig. Oft bringt der Zwang zu einer klaren Darlegung den Autor bereits zu Einsichten, die er vorher nicht hatte.
- Die Verantwortung für die Qualität des untersuchten Prüfobjekts trägt das Team.

*Potenzielles Problem*

Folgende Schwierigkeit kann auftreten: Bei einer schlecht geführten Reviewsitzung kann der Autor in eine psychologisch angespannte Situation geraten, wenn er das Gefühl hat, dass er selbst und nicht das Dokument einer kritischen Untersuchung unterzogen wird.

*Was kosten Reviews und was sparen sie ein?*

Die durch Reviews anfallenden Kosten werden mit ca. 10 %–15 % (s. [Gilb 96, S. 27]) des Entwicklungsbudgets veranschlagt. Die Kosten umfassen die Aktivitäten für den Reviewprozess selbst, die Auswertung der ermittelten Ergebnisse und den Aufwand für deren Umsetzung zur Prozessverbesserung. Einsparungen werden mit 14 %–25 % (s. [Bush 90]) angegeben, wobei der Mehraufwand für die Durchführung der Reviews bereits eingerechnet ist.

Bei konsequenter Nutzung und effizienter Durchführung der Reviews lassen sich bis zu 70 % der Fehler in einem Dokument finden, bevor sie in die folgenden Arbeitsschritte des Entwicklungsprozesses unerkannt übernommen werden. Unter der Berücksichtigung, dass die Kosten der Fehlerbehebung in späteren Entwicklungsschritten erheblich steigen, sind »Angaben plausibel, die über eine Reduktion der Fehlerkosten in der Entwicklung von 75 % und mehr berichten« ([Frühauf 91, S. 114]).

*Tipp*

- Dokumente, die einer formalen Struktur unterliegen, sollten vorab mit einem diese Struktur überprüfenden Werkzeug analysiert werden. Das Werkzeug kann viele Aspekte untersuchen und Fehler oder Abweichungen feststellen, die dann nicht mehr bei einem Review zu kontrollieren sind (s. Kap. 4.2).

### 4.1.3 Grundlegende Vorgehensweise

Mit dem Begriff Review wird eine ganze Gruppe von statischen Überprüfungen bezeichnet. Die einzelnen Verfahren sind in Kapitel 4.1.5 ausführlich beschrieben. Die allen Prüfungen gemeinsam zugrunde liegende Vorgehensweise wird hier kurz in Anlehnung an den IEEE Standard for Software Reviews [IEEE 1028] erläutert.

Bei einem Review sind fünf Arbeitsschritte durchzuführen: Planung, Einführung, Vorbereitung, Reviewsitzung und Nachbereitung.

#### Planung

*Reviews sind selbstverständlich auch zu planen*

In der Planung ist vom Management zu entscheiden, welche Dokumente des Softwareentwicklungsprozesses welchem Reviewverfahren unterzogen werden sollen. Der zu veranschlagende Aufwand für die einzelnen Reviews ist bei der Projektplanung vorzusehen. Der Manager wählt die fachlich geeigneten Personen aus und stellt ein Reviewteam zusammen. Er vergewissert sich in Kooperation mit dem Autor des Prüfobjekts, dass dieses einen →reviewfähigen Status hat, d. h. die Arbeiten am Dokument einen gewissen Abschluss gefunden und eine ausreichende Vollständigkeit erreicht haben. Findet eine Vorbesprechung statt, sind Ort und Zeit festzulegen.

#### Einführung

Die Einführung (oder auch Vorbesprechung oder Initialisierung) dient zur Versorgung der am Review beteiligten Personen mit allen benötigten Informationen. Dies kann in Form einer schriftlichen Einladung[15] erfolgen, oder ein erstes Treffen des Reviewteams wird durchgeführt, um über Bedeutung, Sinn und Zweck der durchzuführenden Reviewsitzung zu informieren. Sind die beteiligten Personen mit dem Umfeld des zu prüfenden Dokuments wenig vertraut, kann auf dem Treffen neben der kurzen Vorstellung des Prüfdokuments auch seine Einbettung in das Anwendungsgebiet dargestellt werden.

15. Ein Beispiel einer Einladung zu einer Reviewsitzung ist in ([Frühauf 91, S. 102]) zu finden, ein weiteres bei [URL imbus-downloads] abrufbar.

*Prüfgrundlagen sind notwendig*

Neben dem Dokument, das einem Review unterzogen werden soll, müssen den beteiligten Personen weitere Unterlagen zur Verfügung stehen. Dazu gehören die Dokumente, die herangezogen werden müssen, um zu entscheiden, ob eine Abweichung, ein Fehler oder eine korrekte Beschreibung des Sachverhalts vorliegen. Dies sind die Dokumente (z. B. Pflichtenheft, Richtlinien oder Standards), gegen die geprüft wird. Darüber hinaus sind Prüfkriterien (z. B. in Form von Checklisten) sehr sinnvoll, die ein strukturiertes Vorgehen unterstützen.

#### Vorbereitung

Die beteiligten Personen des Reviewteams haben sich individuell auf die Sitzung vorzubereiten. Nur bei einer guten Vorbereitung aller Personen ist eine erfolgreiche Reviewsitzung überhaupt möglich.

*Intensive Auseinandersetzung mit dem Prüfobjekt*

Die Gutachter setzen sich intensiv mit dem zu prüfenden Dokument auseinander und verwenden dabei die zur Verfügung gestellten Dokumente als Prüfgrundlage. Erkannte Mängel, Fragen oder Kommentare werden notiert. Meist ist es ratsam, eine Beschränkung auf bestimmte Aspekte vorzunehmen, gegeben durch die gezielte Auswahl der verwendeten Checklisten oder durch weitere Dokumente.

#### Reviewsitzung

Die Reviewsitzung wird von einem Sitzungsleiter, auch →Moderator genannt, geführt. Sie ist in der Regel auf einen festen Zeitrahmen beschränkt. Ziel ist die Beurteilung des Prüfobjekts in Bezug auf die Einhaltung und Umsetzung der Vorgaben und Richtlinien. Die Bewertung der einzelnen Befunde und die Gesamtbeurteilung soll von allen Gutachtern getragen werden.

*Regeln der Reviewsitzung*

In [Frühauf 91, S. 107] sind folgende allgemeine Regeln für eine Reviewsitzung aufgestellt, die hier mit einigen Ergänzungen wiedergegeben werden[16]:

1. Die Reviewsitzung wird auf zwei Stunden beschränkt. Falls nötig wird eine weitere Sitzung frühestens am nächsten Tag einberufen.
2. Der Moderator hat das Recht, eine Sitzung abzusagen oder abzubrechen, wenn
   - einer oder mehrere Experten (Gutachter) nicht erschienen oder ungenügend vorbereitet sind,
   - er aus irgendeinem Grund nicht im Stande ist, die Sitzung erfolgreich und effizient zu leiten.

16. Einige der Regeln treffen allerdings für bestimmte Reviewarten nach [IEEE 1028] nicht in dieser Form zu.

3. Das Resultat (also das zu prüfende Dokument, das Prüfobjekt) und nicht der Autor stehen zur Diskussion:
   – Die Gutachter müssen auf ihre Ausdrücke und Ausdrucksweisen achten.
   – Der Autor darf weder sich noch das Resultat verteidigen (d. h., der Autor sollte keinen »Angriffen« ausgesetzt werden, die ihn zur »Verteidigung« zwingen. Eine Rechtfertigung seiner Entscheidungen wird teilweise als legitim und hilfreich angesehen).
4. Der Moderator darf nicht gleichzeitig als Gutachter fungieren.
5. Allgemeine Stilfragen (außerhalb der Richtlinien) dürfen nicht diskutiert werden.
6. Die Entwicklung und Diskussion von Lösungen ist nicht Aufgabe des Reviewteams.
7. Jeder Gutachter muss Gelegenheit haben, seine Befunde angemessen präsentieren zu können.
8. Der Konsens der Gutachter zu einem Befund ist laufend zu protokollieren.
9. Befunde sind nicht in Form von Anweisungen an den Autor zu protokollieren (zusätzliche konkrete Korrekturvorschläge werden allerdings teilweise durchaus als sinnvoll und hilfreich für die Qualitätsverbesserung angesehen)
10. Die einzelnen Befunde sind zu gewichten[17] als
    – kritischer Fehler (Prüfobjekt ist für den vorgesehenen Zweck unbrauchbar, Fehler muss vor der Freigabe behoben werden),
    – Hauptfehler (Nutzbarkeit des Prüfobjekts beeinträchtigt, Fehler sollte vor der Freigabe behoben werden),
    – Nebenfehler (geringfügige Abweichung, z. B. Rechtschreibfehler oder Mangel im Ausdruck, beeinträchtigt den Nutzen kaum),
    – gut (fehlerfrei, bei Überarbeitung nicht ändern).
11. Vom Reviewteam ist eine Empfehlung über die Annahme des Prüfobjekts abzugeben (siehe Nachbereitung):
    – akzeptieren (ohne Änderungen),
    – akzeptieren (mit Änderungen, kein weiteres Review),
    – nicht akzeptieren (weiteres Review oder andere Prüfmaßnahme erforderlich).
12. Am Schluss haben alle Sitzungsteilnehmer das Protokoll zu unterschreiben.

*Protokoll und Übersicht der Ergebnisse*

Das Protokoll enthält eine Liste aller Befunde, die in der Sitzung diskutiert wurden. In einer zusätzlichen Zusammenfassung des Reviews sind

17. Siehe hierzu auch Kapitel 6.4.3, Fehler der Klasse 2 und 3 können als Hauptfehler angesehen werden.

alle wichtigen Informationen aufzunehmen, wie die Informationen zum Prüfobjekt und den verwendeten Dokumenten, die beteiligten Personen und ihre Rollen (s. Kap. 4.1.4), eine kurze Zusammenfassung der wichtigen Punkte und das Ergebnis der Reviewsitzung mit der Empfehlung der Gutachter.

#### Nachbereitung

Der Manager entscheidet, ob er der Empfehlung folgt oder eine andere Entscheidung trifft, für die er dann aber die alleinige Verantwortung trägt. Im Regelfall wird der Autor die Mängel auf Grundlage der Reviewergebnisse beseitigen. Die ordnungsgemäße Durchführung der Überarbeitung ist zu kontrollieren, meist durch den Manager.

*Zweites Review*

War das Resultat des ersten Reviews nicht akzeptabel und muss ein weiteres Review durchgeführt werden, so ist die hier beschriebene Vorgehensweise für das zweite Review entsprechend, jedoch meist verkürzt, durchzuführen.

Eine umfassende Auswertung der jeweiligen Reviewsitzungen und deren Ergebnisse sollen dazu genutzt werden, den Reviewprozess zu verbessern und die verwendeten Dokumente (Richtlinien und Checklisten) den jeweiligen Gegebenheiten anzupassen und aktuell zu halten.

*Defizite im Entwicklungsprozess erkennen und beheben*

Wiederkehrende oder häufig vorkommende Fehlerarten weisen auf Defizite im Softwareentwicklungsprozess oder im Fachwissen der jeweiligen Personen hin. Notwendige Verbesserungen des Entwicklungsprozesses sind entsprechend zu planen und umzusetzen. Mangel an Fachwissen ist durch Schulung auszugleichen.

### 4.1.4 Rollen und Verantwortlichkeiten

Ein Großteil der Rollen und ihrer Verantwortlichkeiten im Reviewprozess geht aus der Beschreibung des allgemeinen Vorgehens bereits hervor. Hier werden die beteiligten Personen und ihre Aufgaben übersichtsartig beschrieben.

*Manager*

Der Manager, unter dessen Verantwortung das Prüfobjekt entwickelt wird, sorgt für die Auswahl der Prüfobjekte und die zu verwendenden Dokumente. Er stellt die notwendigen Ressourcen zur Verfügung und das Reviewteam zusammen. Er entscheidet über das weitere Vorgehen nach der Reviewsitzung. Ihm obliegen sämtliche Planungsaktivitäten. Jedoch sollten Vertreter der Managementebene an den Sitzungen nicht teilnehmen, da eine Bewertung der Qualifikation des Autors (und nicht eine Bewertung des Dokuments) durch das Manage-

ment nicht auszuschließen ist und eine »freie« Diskussion unter den Teilnehmenden dann möglicherweise eingeschränkt wird.

*Moderator*

Der Moderator leitet die Sitzung und ist die entscheidende Person, was den Erfolg des Reviews betrifft. Er muss bei gegensätzlichen Standpunkten vermitteln können und auch auf »Untertöne« achten. Er darf nicht voreingenommen sein und keine eigene Meinung zum Prüfobjekt äußern.

*Autor*

Der Autor ist der Ersteller des Dokuments, das einem Review unterzogen wird. Sind mehrere Personen an der Erstellung beteiligt gewesen, sollte ein Hauptverantwortlicher benannt werden, der die Rolle des Autors übernimmt. Wichtig ist, dass der Autor die geäußerte Kritik nicht als Kritik an seiner Person auffasst, sondern dass es ausschließlich um die Verbesserung der Qualität des Prüfobjekts geht.

*Gutachter*

Die Gutachter, auch Reviewer oder Inspektoren genannt, sind mehrere (maximal fünf) Fachexperten, die nach der entsprechenden Vorbereitung an der Sitzung teilnehmen. Sie haben darauf zu achten, dass die als gut befundenen Teile der Dokumente auch so benannt werden. Mangelhafte Teile des Prüfobjekts sind als solche zu kennzeichnen sowie die Mängel so zu dokumentieren, dass die Nachvollziehbarkeit und damit die Beseitigung der Mängel gegeben ist.

*Protokollant*

Der Protokollant muss knapp und präzise formulieren können. Er muss während der Sitzung meist nicht ausreichend formulierte Diskussionsbeiträge verfälschungsfrei aufschreiben. Die Rollen Protokollant und Moderator sollten nicht von einer Person wahrgenommen werden. Aus pragmatischen Gründen kann es manchmal sinnvoll sein, den Autor selbst das Protokoll führen zu lassen. Er weiß genau, was im Einzelnen und wie präzise und ausführlich die Anmerkungen der Gutachter zu protokollieren sind, um ausreichend Information für die später von ihm durchzuführenden Änderungen zu haben.

### Mögliche Schwierigkeiten

*Gründe für wenig erfolgreiche Reviews*

Wenn Reviews nicht den gewünschten Erfolg erzielen, kann es mehrere Ursachen haben:

- Das benötigte Personal ist nicht in ausreichendem Maße vorhanden oder verfügt nicht über die erforderliche Ausbildung. Schulungsmaßnahmen sind durchzuführen oder entsprechendes Personal ist von Consulting-Firmen auszuleihen. Die letztgenannte Möglichkeit trifft besonders für den Moderator zu, da dieser eher psychologische als fachbezogene Fähigkeiten mitbringen muss.
- Ist fehlende Zeit Ursache für unbefriedigende Reviewergebnisse, so liegen Fehleinschätzungen bei der Ressourcenplanung durch das

Management vor. Eventuell kann ein weniger aufwändiger Reviewtyp Abhilfe schaffen.

- Scheitern Reviews an der fehlenden Vorbereitung, so liegt es meist an der falschen Auswahl der Gutachter. Sollte fehlende Einsicht der Gutachter in die Bedeutung der Reviews und den hohen Wirkungsgrad zur Qualitätssteigerung der untersuchten Dokumente Ursache für das Scheitern sein, dann ist die Nützlichkeit der Reviews mit entsprechendem Zahlenmaterial zu belegen.
- Ein Scheitern eines Reviews kann auch an der fehlenden oder unzureichenden Dokumentation liegen. Es muss vorab geprüft werden, ob alle benötigten Dokumente in ausreichender Beschreibungstiefe vorhanden sind. Nur wenn das der Fall ist, ist ein Review durchzuführen.
- Bei fehlender Unterstützung durch das Management kann ein Reviewprozess nicht erfolgreich sein, da die benötigten Ressourcen nicht bereitgestellt und die erzielten Ergebnisse nicht zur Prozessverbesserung genutzt werden. In der Praxis ist dies leider häufig der Fall.

### 4.1.5 Reviewarten

Zwei grobe Gruppen von Reviews lassen sich anhand des untersuchten Prüfobjekts unterscheiden:

- Reviews, die sich auf Produkte oder Teilprodukte beziehen, die während des Entwicklungsprozesses erstellt werden, und
- Reviews, die Projektpläne und den Entwicklungsprozess als solches analysieren.

***Exkurs***

Die zweite Gruppe wird auch als →Managementreview ([Balzert 98], [IEEE 1028])[18] oder Projektreview bezeichnet. Zur grundlegenden Vorgehensweise (Kap. 4.1.3) ergeben sich bei einem Managementreview die folgenden Unterschiede:

In der Vorbereitung auf die Reviewsitzung können Vorträge zu ausgewählten Themen ausgearbeitet werden. Informationsgrundlage sind u. a. die Projektdokumentation und die Protokolle der bis zu diesem Zeitpunkt bereits durchgeführten Prüfungen einzelner Produkte und Dokumente. Die vorbereiteten Präsentationen werden in der Sitzung vorgetragen. Im Gegensatz zur ersten Reviewgruppe werden jedoch keine einzelnen Dokumente vorgestellt und diskutiert. Prüfobjekt ist

18. In [ISO 8402] wird ein Managementreview enger gefasst und definiert als eine formelle Bewertung des Standes und der Angemessenheit des Qualitätsmanagementsystems in Bezug auf die Qualitätspolitik sowie auf Zielsetzungen durch die oberste Leitung.

das Projekt als Ganzes und die Feststellung seines augenblicklichen Zustands. Der Projektzustand wird hinsichtlich technischer, wirtschaftlicher, zeitlicher und managementmäßiger Aspekte bewertet.

Managementreviews können weit mehr als zwei Stunden dauern, sie können sogar die Dauer von bis zu zwei Arbeitstagen annehmen. Sie werden oft beim Erreichen eines Meilensteins der Projektplanung durchgeführt, wenn eine Hauptphase im Softwareentwicklungsprozess abgeschlossen werden soll oder als Postmortem-Analyse, um aus dem beendeten Projekt zu lernen.

Im Folgenden wird die erste Gruppe der Reviews weiter präzisiert. Es können folgende Ausprägungen unterschieden werden: →Walkthrough, Inspektion, →technisches Review und →informelles Review. Bei den einzelnen Beschreibungen wird auf die hauptsächlichen Unterschiede zur grundlegenden Vorgehensweise (Kap. 4.1.3) eingegangen.

### Walkthrough

Nach [Balzert 98] ist ein Walkthrough[19] eine »manuelle, informale Prüfmethode, um Fehler, Defekte, Unklarheiten und Probleme in schriftlichen Dokumenten zu identifizieren. Der Autor präsentiert das Dokument in einer Sitzung den Gutachtern«.

*Nachvollziehen typischer Benutzungssituationen*

Schwerpunkt des Walkthrough bildet die Sitzung. Die Vorbereitung hat im Vergleich zu den anderen Reviewarten den geringsten Umfang, teilweise kann sogar auf sie verzichtet werden[20]. In der Sitzung übernimmt der Autor die Rolle des Moderators und ggfs. auch die des Protokollanten. Er stellt das Prüfobjekt den Gutachtern vor. Meist werden typische Benutzungssituationen, auch Szenarien genannt, ablauforientiert durchgespielt. Es können auch einzelne Testfälle nachgespielt werden. Die Gutachter versuchen durch meist spontane Fragen, mögliche Fehler und Probleme aufzudecken.

*Geeignet für kleine Entwicklungsteams*

Das Vorgehen ist für kleine Entwicklungsteams von bis zu fünf Personen geeignet und verursacht wenig Aufwand, da Vor- und Nachbereitungen von geringem Umfang bzw. nicht zwingend erforderlich sind. Es lässt sich bei der Prüfung von »unkritischen« Dokumenten einsetzen. Da der Autor auch die Sitzungsleitung inne hat, kann er den Verlauf der Sitzung stark beeinflussen. Dies kann sich nachteilig auf das Ergebnis auswirken, wenn der Autor die Diskussion der kritischen Stellen des Prüfobjekts nicht intensiv genug oder erst gar nicht durchführen lässt. Für die Nacharbeit ist der Autor verantwortlich, eine Kontrolle findet nicht statt.

---

19. *walk through* – durchgehen, oft auch als »Structured Walkthrough« bezeichnet.
20. Nach [IEEE 1028] sollten die Teilnehmer die Dokumente vorab erhalten und vorbereitet an der Sitzung teilnehmen.

### Inspektion

*Formaler Ablauf*

Eine Inspektion wird formaler durchgeführt als ein Walkthrough. Die Ziele der Inspektion werden bei der Planung festgelegt, und es wird eine begrenzte Anzahl von Aspekten behandelt, auf die sich die Gutachter, oft auch als Inspektoren bezeichnet, vorbereiten. Häufig wird diese Reviewart auch als Design- oder Code-/Softwareinspektion bezeichnet. Der Name weist damit auf die Dokumente hin, die einer Inspektion unterzogen werden (s. [Fagan 76]).

Die Sitzung wird von einem Moderator geleitet und beginnt mit einer kurzen Einführung. Ein Gutachter[21] trägt in einer straffen und möglichst logischen Weise die Inhalte des Prüfobjekts vor. Wenn es sinnvoll erscheint, werden auch Passagen vorgelesen. Die Gutachter stellen dabei ihre Fragen, wobei die ausgewählten Aspekte der Prüfung intensiv diskutiert werden. Der Autor beantwortet an ihn gerichtete Fragen, bleibt aber ansonsten passiv.

Wenn die Diskussion abschweift, ist es die Aufgabe des Moderators einzugreifen. Alle festgestellten Unstimmigkeiten und Fehler werden vom Protokollanten erfasst. Am Ende der Sitzung werden alle aufgezeichneten Fehler vorgestellt und von allen Beteiligten auf Vollständigkeit geprüft.

*Zusätzliche Bewertung des Entwicklungs- und Inspektionsprozesses*

Bei einer Inspektion werden auch Daten gesammelt, die zur Qualitätsbeurteilung des Entwicklungs- und Inspektionsprozesses herangezogen werden. Die Inspektion dient somit neben der Beurteilung der untersuchten Dokumente auch zur Optimierung des Entwicklungsprozesses. Die gesammelten Daten werden dahingehend analysiert, die Ursachen für Schwachstellen im Entwicklungsprozess zu finden. Nach einer Verbesserung des Prozesses wird der Erfolg der Änderung kontrolliert, indem die gesammelten Daten vor der Änderung mit den aktuellen Daten verglichen werden.

Eine Inspektion folgt einem vorgeschriebenen formalen Ablauf. Jede beteiligte Person hat eine vorgeschriebene Rolle auszufüllen. Der Ablauf ist durch Regeln definiert, und es werden zu den einzelnen Aspekten Prüfkriterien, meist in Form von Checklisten, von den Gutachtern verwendet. Für die einzelnen Prüfkriterien sind Eintritts- und Austrittskriterien definiert.

### Technisches Review

*Erfüllt das Prüfobjekt seinen Zweck?*

Bei einem technischen Review, auch als »Peer Review« bezeichnet, steht die Übereinstimmung des zu prüfenden Dokuments mit der Spe-

21. In [IEEE 1028] als Leser (*reader*) benannt.

zifikation, die Eignung für den beabsichtigten Einsatz und die Einhaltung von Standards im Mittelpunkt der Untersuchungen. Das Prüfobjekt wird in der Vorbereitungsphase durch die Gutachter einer individuellen Prüfung anhand der festgelegten Prüfkriterien unterzogen.

*Fachexperten als Gutachter*

Es ist darauf zu achten, dass Fachexperten als Gutachter am technischen Review teilnehmen. Die Anmerkungen der Gutachter zu den einzelnen Prüfkriterien werden schriftlich festgehalten und dem Sitzungsleiter vorab übergeben[22]. Er priorisiert diese nach vermuteter Wichtigkeit, und in der Sitzung werden dann nur die wesentlichen Anmerkungen diskutiert.

*Hoher Vorbereitungsaufwand*

Es ist ein hoher Aufwand in der Vorbereitungsphase gegeben. In der Sitzung dokumentiert der Protokollant alle Aussagen und erstellt die abschließende Ergebnisdokumentation.

Das Ergebnis muss einstimmig im Gesamturteil gefällt und von allen Personen getragen und unterzeichnet werden. Eventuelle Sondervoten können zu Protokoll gegeben werden. Es ist nicht Aufgabe der am technischen Review Beteiligten, über die Konsequenzen des Ergebnisses zu entscheiden. Bei einem hohen Formalisierungsgrad des Reviews können auch Eintritts- und Austrittskriterien der Prüfkriterien definiert sein.

#### Informelles Review

Das informelle Review ist eine abgeschwächte Version eines Reviews, die keinem formalisiertem Prozess unterliegt und bei der das Ergebnis nicht explizit zu dokumentieren ist. Das informelle Review hat wegen des geringeren Aufwands eine hohe Akzeptanz und weite Verbreitung in der Praxis gefunden.

#### Kriterien zur Auswahl

Wann welche Reviewart eingesetzt werden soll, hängt stark von der geforderten Qualität und dem einzusetzenden Aufwand ab. Das Projektumfeld ist entscheidend, direkte Empfehlungen können nicht gegeben werden. Es ist im Einzelfall zu entscheiden, welche Reviewart in der jeweiligen Projektsituation angemessen ist. Nachfolgend sind einige Fragen und Kriterien zur Erleichterung der Auswahl der Arten aufgeführt:

- Die Form, in der das Ergebnis der Prüfung vorliegen soll, kann zur Auswahl herangezogen werden. Ist eine ausführliche Dokumenta-

22. In [IEEE 1028] gilt das auch für die Inspektion.

tion erforderlich oder reicht eine undokumentierte Umsetzung der Prüfergebnisse?
- Lässt sich die Terminkoordination einfach oder schwierig vornehmen? Fünf bis sieben Fachkräfte für einen oder mehrere Termine zu verpflichten, kann sehr aufwändig sein.
- Ist die Berücksichtigung von Fachwissen aus mehreren Gebieten erforderlich?
- Wie umfangreich muss das Fachwissen der Gutachter für das jeweilige Prüfobjekt sein? Wie hoch ist das Engagement der vorgesehenen Gutachter?
- Steht der Vorbereitungsaufwand in einem angemessenen Verhältnis zum erwarteten Ergebnis?
- Wie hoch ist der Formalisierungsgrad des Prüfobjekts? Lassen sich werkzeuggestützte Analysen durchführen?

#### Anmerkungen

Wie am Anfang des Kapitels bereits erwähnt, gibt es keine einheitlichen Beschreibungen und Abgrenzungen zu den einzelnen Reviewarten. Die Grenzen sind eher fließend, und die gleichen Begriffe werden mit unterschiedlicher Bedeutung verwendet.

*Firmenspezifische Ausprägungen*

Generell lässt sich feststellen, dass Reviews firmenspezifischen Ausprägungen unterliegen. Sie werden auf die jeweiligen Bedürfnisse und Erfordernisse der Projekte zugeschnitten. Ihr Wirkungsgrad wird dadurch positiv beeinflusst.

Allgemein ist eine kooperative Zusammenarbeit zwischen den an der Softwareentwicklung beteiligten Personen als qualitätssteigernd einzustufen. Wenn Arbeitsergebnisse untereinander begutachtet und kontrolliert werden, können Fehler und Unstimmigkeiten aufgedeckt werden. In diesem Sinn kann das paarweise Programmieren, wie es im →Extreme Programming vorgeschlagen wird, auch als eine permanente Form eines »Zweipersonenreviews« angesehen werden ([Beck 00]).

## 4.2 Statische Analyse

*Analyse ohne Programmausführung*

Das Ziel der statischen Analyse ist, ähnlich dem der Reviews, die Aufdeckung vorhandener Fehler oder fehlerträchtiger Stellen in einem Dokument. So sind beispielsweise auch Rechtschreibprüfprogramme als eine Art statische Analysatoren anzusehen, die Fehler in Texten nachweisen und damit zur Qualitätsverbesserung beitragen. Die Bezeichnung »statische Analyse« weist darauf hin, dass diese Form der Prüfung auch keine Ausführung der Prüfobjekte (eines Programms) be-

inhaltet. Ein weiteres Ziel ist die Ermittlung von Messgrößen oder Metriken, um eine Qualitätsbewertung durchführen und somit Qualität messen und nachweisen zu können.

*Formale Dokumente*

Das zu analysierende Dokument muss allerdings nach einem vorgegebenen Formalismus aufgebaut sein, also einer formalen Struktur unterliegen, um durch ein Werkzeug überprüft werden zu können. Die Anwendung der statischen Analyse ist nur mit entsprechender Werkzeugunterstützung sinnvoll. Dokumente, die einem Formalismus unterliegen können, sind beispielsweise die technischen Anforderungen, die Softwarearchitektur oder der Softwareentwurf. In der Praxis ist leider oft der Programmcode das erste und einzige formale Dokument der Softwareentwicklung, das einer statischen Analyse unterzogen werden kann.

*Statische Analyse und Reviews*

Statische Analyse und Reviews stehen in einem engen Zusammenhang. Wird vor dem Review eines formalen Dokuments eine statische Analyse durchgeführt, können bereits eine Anzahl von Fehlern und Unstimmigkeiten nachgewiesen werden, und die Menge der im Review zu berücksichtigenden Aspekte ist erheblich geringer. Da die statischen Analysen werkzeuggestützt durchgeführt werden, ist der Aufwand wesentlich niedriger als bei einem Review.

***Tipp***

- Haben Dokumente einen Grad an Formalität erreicht, der es erlaubt, eine statische Analyse werkzeuggestützt durchzuführen, dann sollte dies auf jeden Fall vor den Reviews der Dokumente erfolgen, da dadurch relativ kostengünstig Fehler und Unstimmigkeiten aufgedeckt und die Reviews verkürzt werden.

Mit der statischen Analyse lassen sich allerdings nicht alle Fehler und Unstimmigkeiten nachweisen. Es gibt Fehler oder präziser Fehlerzustände, die erst bei der Ausführung, also zur Laufzeit des Programms, zur Wirkung kommen und vorher nicht ermittelbar sind. Wird beispielsweise bei einer Division der Wert des Divisors in einer Variablen gehalten, so kann diese Variable zur Laufzeit den Wert null annehmen, was zu einer Fehlerwirkung führt. Statisch ist dieser Fehlerzustand meist nicht erkennbar, es sei denn, der Variablen wird eine Konstante mit dem Wert null zugewiesen

*Compiler ist ein Analysewerkzeug.*

Alle Compiler führen eine statische Analyse des Programmtextes durch, indem sie die Einhaltung der Syntax der jeweiligen Programmiersprache prüfen. Die meisten Compiler bieten darüber hinaus noch zusätzliche Informationen, die durch die statische Analyse ermittelt werden. Neben den Compilern gibt es Werkzeuge, so genannte

→Analysatoren, die speziell zur gezielten Durchführung einzelner oder ganzer Gruppen von Analysen eingesetzt werden.

Fehlerzustände bzw. fehlerträchtige Situationen, die mit der statischen Analyse von Programmcode nachgewiesen werden können, sind beispielsweise

- Verletzung der Syntax,
- Konventions- und Standardabweichungen,
- →Kontrollflussanomalien und
- →Datenflussanomalien.

### 4.2.1 Compiler als statisches Analysewerkzeug

Die Verletzung der Syntax der Programmiersprache wird durch statische Analyse erkannt und als Fehler oder Warnung gemeldet. Viele Compiler erstellen darüber hinaus weitere Informationen und führen Überprüfungen durch.

Beispiele hierfür sind:

- Erstellung der Verwendungsnachweise (auch mit *cross reference list* bezeichnet) der einzelnen Programmelemente (z. B. Variablen, Funktionen)
- Prüfung der typgerechten Verwendung der Daten und Variablen bei den strenggetypten Programmiersprachen
- Ermittlung von nicht deklarierten Variablen
- Nicht erreichbarer Code
- Über- oder Unterschreitung von Feldgrenzen (bei statischer Adressierung)
- Prüfung der Konsistenz von Schnittstellen
- Prüfung der Verwendung aller Marken als Sprunganweisung und Sprungziel

Die Informationen werden meist in Form von Listen zur Verfügung gestellt. Nicht immer kann anhand der Ergebnisse sofort auf einen Fehlerzustand geschlossen werden, es sind dann weitere Untersuchungen anzuschließen.

### 4.2.2 Prüfung der Einhaltung von Konventionen und Standards

Mithilfe von Analysatoren kann auch die Einhaltung von Konventionen und Standards überprüft werden. Ob beispielsweise alle Vorschriften der Programmierkonventionen eingehalten wurden, lässt sich durch die entsprechenden Analysatoren nachweisen. Diese Art der Überprüfung benötigt wenig Zeit und so gut wie keine Personalres-

sourcen. Es sollten überhaupt nur solche Richtlinien in einem Projekt zugelassen werden, zu denen Prüfwerkzeuge vorhanden sind. Alle anderen Vorschriften erweisen sich in der Praxis ohnehin schnell als bürokratischer Ballast. Darüber hinaus ist meist ein weiterer Vorteil gegeben: Ist den Programmierern bewusst, dass der Programmcode von einem Werkzeug auf Einhaltung der Programmierrichtlinien überprüft wird, ist deren Bereitschaft, die Richtlinien umzusetzen, bedeutend höher als ohne automatische Prüfung.

### 4.2.3 Durchführung der Datenflussanalyse

*Analyse der Verwendung der Daten*

Eine weitere Möglichkeit der Fehleraufdeckung ist die →Datenflussanalyse. Hierbei wird die Verwendung von Daten auf →Pfaden durch den Programmcode untersucht. Nicht immer können Fehlerzustände nachgewiesen werden, oft wird in diesem Zusammenhang dann von →Anomalien oder Datenflussanomalien gesprochen. Mit Anomalie wird eine Unstimmigkeit bezeichnet, die zu einer Fehlerwirkung führen kann, aber nicht zwingend zu einer Fehlerwirkung führt.

Datenflussanomalien sind beispielsweise die referenzierende Verwendung einer Variablen ohne vorherige Initialisierung oder die Nichtverwendung eines Wertes einer Variablen. Zur Analyse wird der Gebrauch jeder einzelnen Variablen untersucht. Folgende drei Verwendungen oder Zustände der Variablen werden unterschieden:

- **definiert (d):** Die Variable erhält einen Wert zugewiesen.
- **referenziert (r):** Der Wert der Variablen wird gelesen bzw. verwendet.
- **undefiniert (u):** Die Variable hat keinen definierten Wert.

*Datenflussanomalien*

Dabei lassen sich drei Arten von Datenflussanomalien unterscheiden:

- **ur-Anomalie:** Ein undefinierter Wert (u) einer Variablen wird auf einem Programmpfad gelesen (r).
- **du-Anomalie:** Die Variable erhält einen Wert (d), der allerdings ungültig (u) wird, ohne dass er zwischenzeitlich verwendet wurde.
- **dd-Anomalie:** Die Variable erhält auf einem Programmpfad ein zweites Mal einen Wert (d), ohne dass der erste Wert (d) verwendet wurde.

---

***Beispiel für die Anomalien***

Die unterschiedlichen Anomalien sollen an einem Beispiel erläutert werden. Gegeben sei folgende Funktion, die die ganzzahligen Werte der Parameter `Max` und `Min` unter Zuhilfenahme der Variablen `Hilf` tauschen soll, falls der Wert der Variablen `Min` größer ist als der Wert der Variablen `Max`:

```
void tausch (int Min, int Max) {
    int Hilf;
    if (Min > Max){
    Max  = Hilf;
    Max  = Min;
    Hilf = Min;
   }
}
```

Nach der Analyse des Gebrauchs der einzelner Variablen lassen sich folgende Anomalien feststellen:

- **ur-Anomalie der Variablen `Hilf`**: Der Gültigkeitsbereich der Variablen ist auf die Funktion beschränkt. Die erste Verwendung der Variablen ist auf der rechten Seite einer Zuweisung. Die Variable hat zu diesem Zeitpunkt noch einen undefinierten Wert, der an dieser Stelle referenziert wird. Eine Initialisierung bei der Deklaration der Variablen wurde nicht vorgenommen (diese Anomalie erkennen auch übliche C++-Compiler, sofern eine entsprechend hohe Warnstufe aktiviert ist).
- **dd-Anomalie der Variablen `Max`**: Die Variable wird hintereinander jeweils auf der linken Seite einer Zuweisung verwendet, bekommt somit zweimal einen Wert zugewiesen. Entweder kann die erste Zuweisung entfallen oder eine Verwendung des ersten Wertes ist vergessen worden.
- **du-Anomalie der Variablen `Hilf`**: In der letzten Anweisung der Funktion bekommt die Variable Hilf noch einen Wert zugewiesen, der nirgends verwendet werden kann, da die Variable nur innerhalb der Funktion gültig ist.

---

*Datenflussanomalien sind meist nicht so offensichtlich*

Im Beispiel sind die Anomalien sehr offensichtlich. Zu bedenken ist aber, dass zwischen den jeweils betroffenen Anweisungen, die zu den Anomalien führen, beliebig viele andere Anweisungen mit anderen Variablen stehen können. Die Anomalien sind dann nicht mehr so offensichtlich und können bei einer manuellen Prüfung, z. B. einem Review, leicht übersehen werden. Ein Werkzeug zur Datenflussanalyse deckt die Anomalien auf.

Nicht jede Anomalie führt direkt zu einem fehlerhaften Verhalten. Beispielsweise hat eine du-Anomalie nicht immer direkte Auswirkungen, das Programm kann korrekt laufen. Es stellt sich nur die Frage, warum die Zuweisung an dieser Stelle des Programms vor dem Ende des Gültigkeitsbereichs der Variablen vorhanden ist. Meist lohnt sich eine genauere Untersuchung der anomalen Programmstellen, um weitere Unstimmigkeiten ausfindig zu machen.

### 4.2.4 Durchführung der Kontrollflussanalyse

*Kontrollflussgraph*

In Abbildung 4-1 ist ein Programmstück als Kontrollflussgraph dargestellt. In dem gerichteten Graphen sind die Anweisungen des Programms als Knoten dargestellt. Sequenzen von Anweisungen werden ebenfalls als ein einziger Knoten dargestellt, da es innerhalb der Sequenz zu keiner Änderung des Programmablaufs kommen kann. Wenn die erste Anweisung der Sequenz zur Ausführung kommt, dann auch alle weiteren.

Änderungen des Ablaufs im Programmtext werden durch Entscheidungen, z. B. in IF-Anweisungen, herbeigeführt. Ist der ermittelte Wahrheitswert der Bedingung »wahr«, dann wird die Ausführung des Programms mit dem THEN-Teil der IF-Anweisung fortgeführt, im anderen Fall (»unwahr«) wird der ELSE-Teil durchlaufen. Schleifen führen zu vorherigen Anweisungen zurück, wodurch ein Teil des Graphen mehrfach durchlaufen wird.

*Kontrollflussanomalien*

Durch die Anschaulichkeit des Kontrollflussgraphen lassen sich die Abläufe durch ein Programmstück leicht erfassen und mögliche Anomalien erkennen. Dies können Sprünge aus Schleifen heraus sein oder ein Programmstück, das mehrere Ausgänge hat. Diese Unstimmigkeiten müssen keine Fehlerzustände sein, widersprechen aber den Grundsätzen der strukturierten Programmierung. Voraussetzung ist, dass der Graph nicht manuell, sondern von einem Werkzeug erstellt wird, das eine Eins-zu-eins-Abbildung zwischen Programmtext und Graph gewährleistet.

Sind Teile des Graphen oder der ganze Graph sehr unübersichtlich und die Zusammenhänge und der Ablauf kaum nachvollziehbar, sollte eine Überarbeitung des Programmtextes erfolgen, da komplexe Ablaufstrukturen oft fehlerträchtig sind.

***Exkurs: Vorgänger-Nachfolger-Tabelle***

Neben den Graphen können auch Vorgänger-Nachfolger-Tabellen erstellt werden, aus denen hervorgeht, wie jede Anweisung mit anderen Anweisungen in Beziehung steht. Gibt es zu einer Anweisung keine Vorgänger, so ist diese Anweisung nicht erreichbar (so genannter *dead code*). Ein Fehlerzustand oder zumindest eine Unstimmigkeit ist erkannt. Nur für die erste und letzte Anweisung eines Programms gelten die Ausnahmen, dass es keine Vorgänger- bzw. Nachfolgeanweisung geben darf. Für Programme mit mehreren Einstiegs- bzw. Ausstiegspunkten gilt Entsprechendes.

### 4.2.5 Ermittlung von Metriken

*Messung von Qualitätsmerkmalen*

Neben den Analysen des Daten- und Kontrollflusses und der Einhaltung der Konventionen werden von den Werkzeugen der statischen

Analyse auch Messwerte geliefert. Mit den Maßzahlen oder Metriken werden Qualitätsmerkmale gemessen. Die gemessenen Werte müssen allerdings dahingehend beurteilt werden, ob sie den spezifizierten Anforderungen genügen (s. [ISO 9126]). Eine Übersicht gängiger Maßzahlen ist in [Fenton 91] zu finden.

»Die Definition von Metriken für bestimmte Eigenschaften von Software basiert auf dem Wunsch, eine quantitative Aussage bezüglich des von Natur aus abstrakten Produktes Software zu gewinnen« (aus [Liggesmeyer 90, S. 215]). Eine Metrik liefert somit immer nur Aussagen bezüglich eines untersuchten Aspekts, und die ermittelten Maßzahlen sind erst im Vergleich zu den Zahlen aus anderen untersuchten Programm(teil)en aussagekräftig.

*Zyklomatische Zahl*

Auf eine Maßzahl, die →zyklomatische Zahl [McCabe 76], soll im Weiteren näher eingegangen werden. Die zyklomatische Zahl misst die strukturelle Komplexität von Programmcode. Grundlage zur Berechnung der zyklomatischen Zahl ist der Kontrollflussgraph.

Für einen Kontrollflussgraphen G eines Programms oder einer Programmkomponente wird die zyklomatische Zahl nach folgender Formel errechnet:

$$v(G) = e - n + 2p$$

v(G) – zyklomatische Zahl des Graphen G
e – Anzahl der Kanten des Kontrollflussgraphen
n – Anzahl der Knoten des Kontrollflussgraphen
p – Anzahl der verbundenen Komponenten[23]

---

***Beispiel zur Berechnung der zyklomatischen Zahl***

Nehmen wir an, das Programmstück zum Graphen der Abb. 4-1 ist eine Komponente, z. B. eine Funktion, die aufgerufen wird, so ergibt sich die zyklomatische Zahl für diese Funktion als:

$$v(G) = e - n + 2p = 17 - 13 + 2*1 = 6$$

mit
e (Anzahl der Kanten des Graphen) = 17,
n (Anzahl der Knoten des Graphen) = 13 und
p (Anzahl der Komponenten) = 1.

23. Eine verbundene Komponente ist ein einziger Kontrollflussgraph, z. B. der Graph einer Funktion.

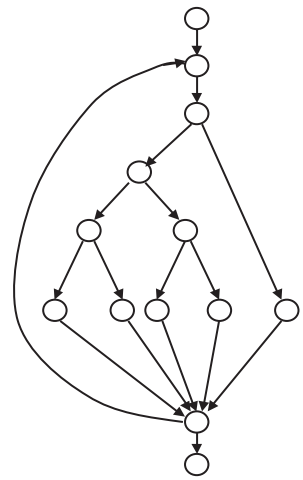


***Abb. 4–1*** *Kontrollflussgraph zur Berechnung der zyklomatischen Zahl (identisch mit Abb. 2-1)*

Der Messwert 6, den McCabe für akzeptabel hält, liegt im mittleren Bereich. Er geht davon aus, dass ein Messwert höher als 10 nicht tolerabel ist und eine Überarbeitung des Programmteils erfolgen sollte.

*Die zyklomatische Zahl gibt Auskunft über den Testaufwand*

Die zyklomatische Zahl kann genutzt werden, um Abschätzungen in Bezug auf die Testbarkeit und die Wartbarkeit des Programmteils vorzunehmen. Die zyklomatische Zahl gibt die Anzahl der unabhängigen Pfade[24] im Programmstück an. Soll eine Zweigüberdeckung (s. Kap. 5.2.2) zu 100 % erfüllt werden, dann sind diese Pfade des Kontrollflussgraphen mindestens einmal zur Ausführung zu bringen. Die zyklomatische Zahl liefert somit wichtige Hinweise zum Umfang des Tests.

Für die Wartbarkeit ist die Verständlichkeit eines Programmstücks von entscheidender Bedeutung. Je höher die ermittelte zyklomatische Zahl ist, je schwieriger ist ein Nachvollziehen des Ablaufs des Programmstücks und damit der Verständlichkeit.

***Exkurs***

Die zyklomatische Zahl ist nach ihrer Veröffentlichung viel diskutiert worden, unter anderem deshalb, weil die Komplexität der Bedingungen, die zur Auswahl des Kontrollflusses führen, nicht in das Maß eingeht. Ob es sich um eine einfache Bedingung oder um eine Bedingung handelt, die aus mehreren Teilbedingungen verbunden mit logischen Operatoren besteht, hat keinen Einfluss auf die Berechnung der zyklomatischen Zahl. Viele Erweiterungen und Ergänzungen sind in diesem Zusammenhang veröffentlicht worden.

24. Gemeint sind alle linear unabhängigen vollständigen Pfade (s. [Liggesmeyer 90, S. 219]).

## 4.3 Zusammenfassung

- Mehrere Augenpaare sehen mehr als ein Augenpaar – auch in der Softwareentwicklung. Reviews zur Kontrolle und Qualitätssteigerung nehmen diesen Grundsatz auf. Dokumente werden von mehreren Personen begutachtet und in einer Sitzung diskutiert. Die Ergebnisse werden protokolliert.
- Ein fundamentaler Reviewprozess besteht aus den folgenden Aktivitäten: Planung, Einführung, Vorbereitung, Reviewsitzung und Nachbereitung. Rollen und damit zu verteilende Aufgaben sind: Manager, Moderator, Autor, Gutachter und Protokollant.
- Es gibt eine ganze Reihe von unterschiedlichen Ausprägungen oder Arten von Reviews und leider keine einheitliche, durchgängige Begriffsfestlegung zwischen den verschiedenen Standards und Normen.
- Das Walkthrough ist eine informelle Vorgehensweise, bei der der Autor sein Dokument den Gutachtern in der Sitzung präsentiert. Die Vorbereitung hat eine geringen Umfang. Das Walkthrough ist besonders für kleine Entwicklungsteams geeignet.
- Die Inspektion wird formaler durchgeführt. Eine Vorbereitung erfolgt anhand von Checklisten, für die einzelnen Prüfkriterien sind Eintritts- und Austrittskriterien definiert, die Sitzung wird von einem geschulten Moderator geleitet.
- Bei dem technischen Review sind vorab die Ergebnisse der Gutachter in schriftlicher Form dem Sitzungsleiter zu übergeben, damit er eine Priorisierung nach vermuteter Wichtigkeit vornehmen kann.
- Das informelle Review unterliegt keinem formalisiertem Prozess, und auch die Form der Ergebnisdokumentation ist nicht vorgeschrieben. Wegen des geringeren Aufwands hat diese Form des Reviews eine hohe Akzeptanz und weite Verbreitung in der Praxis gefunden.
- Allgemein unterliegen Reviews firmenspezifischen Ausprägungen. Sie werden auf die jeweiligen Bedürfnisse und Erfordernisse zugeschnitten, wodurch ihr Wirkungsgrad erhöht wird. Wichtig ist, dass eine kooperative Zusammenarbeit zwischen den an der Softwareentwicklung beteiligten Personen etabliert ist.
- Neben den Reviews lassen sich auch werkzeuggestützt eine ganze Reihe von Ermittlungen an den Dokumenten durchführen, die einen gewissen Formalisierungsgrad haben. Die Untersuchungen werden unter dem Oberbegriff statische Analysen zusammengefasst und ohne Ausführung des Test- oder Prüfobjekts durchgeführt.

- Der Compiler ist wohl das gebräuchlichste Analysewerkzeug und deckt Fehlerzustände in der Syntax des Programmcodes auf. Compiler bieten aber meist noch weitere Untersuchungen und Informationsaufbereitungen an.
- Der Verstoß gegen Standards und andere Konventionen kann ebenfalls mithilfe eines Analysewerkzeugs ermittelt werden.
- Zum Nachweis von Anomalien im Daten- und Kontrollfluss der Programmtexte stehen Werkzeuge zur Verfügung. Hilfreiche Aussagen zum Ablauf und zur Datenverwendung werden ermittelt, die oft auf fehlerträchtige Stellen hinweisen.
- Der Einsatz von Metriken dient zur Messung von Qualität. Die zyklomatische Zahl errechnet die Anzahl der unabhängigen Pfade im untersuchten Programmtext. Aussagen zum strukturellen Aufbau und zum Testaufwand lassen sich gewinnen.
- Generell sollten erst die statischen Analysen durchgeführt werden, bevor das Dokument einem Review unterzogen wird. Es werden relativ kostengünstig Fehlerzustände und Unstimmigkeiten aufgedeckt und die Reviews in ihrem Umfang verkürzt.

# 5 Dynamischer Test

*In diesem Kapitel wird das Testen von Software durch Ausführen der Testobjekte auf einem Rechner detailliert beschrieben. Die unterschiedlichen Vorgehensweisen zur Spezifikation der Testfälle und zur Festlegung von Kriterien zum Beenden der Tests werden erläutert und anhand von Beispielen erklärt. Es werden unterschiedliche Verfahren zur Ermittlung der Testfälle vorgestellt. Die Verfahren lassen sich in Black-box- und White-box-Verfahren unterteilen. Ergänzende Verfahren zur Testfallermittlung runden das Kapitel ab.*

*Ausführen des Testobjekts auf einem Rechner*

Meist wird unter Testen von Software das Ausführen des Testobjekts auf einem Rechner verstanden. Zur Verdeutlichung wird auch der Begriff dynamischer Test verwendet. Das Testobjekt wird mit Eingabedaten versehen und zur Ausführung gebracht. Es muss ein ablauffähiges Programm vorliegen. Da dies in den unteren Teststufen (Komponenten- und Integrationstest) nicht der Fall ist, muss dort das Testobjekt in einen Testrahmen (engl. *test bed,* s. Abb. 5-1) eingebettet werden, um ein ablauffähiges Programm zu erhalten.

*Testrahmen erforderlich*

Das Testobjekt wird meist weitere Programmteile über definierte Schnittstellen aufrufen. Diese Programmteile werden durch Platzhalter (→Stellvertreter, →*stub*) realisiert, wenn sie noch nicht fertig implementiert und damit einsatzbereit sind oder für diesen Test des Testobjekts nur simuliert werden sollen. Platzhalter simulieren das Ein-/Ausgabe-Verhalten des eigentlich aufzurufenden Programmteils[25].

Weiterhin ist vom Testrahmen das Testobjekt mit den Eingabedaten zu versorgen. Meist muss dazu ein das Testobjekt aufrufendes Programmteil simuliert werden. Ein Testtreiber übernimmt diese Aufgabe. Treiber und Stellvertreter bilden zusammen den Testrahmen, der mit dem Testobjekt zusammen das ablauffähige Programm bildet.

Der Testrahmen ist häufig vom Tester zu entwickeln bzw. durch Ausbau von Standardtestrahmen auf die Schnittstellen des Testobjekts hin anzupassen. Es können auch Testrahmengeneratoren eingesetzt werden (s. Kap. 7.1.3). Nachdem so ein lauffähiges Testobjekt vorliegt, kann der dynamische Test durchgeführt werden.

---

25. Im Gegensatz zum Stub, mit seiner rudimentären Funktionalität, bietet ein →Dummy einen nahezu vollwertigen Ersatz für die echte Implementierung. Ein →Mock unterscheidet sich vom Dummy durch zusätzliche Funktionalität für Testzwecke (nach [Link 02]).

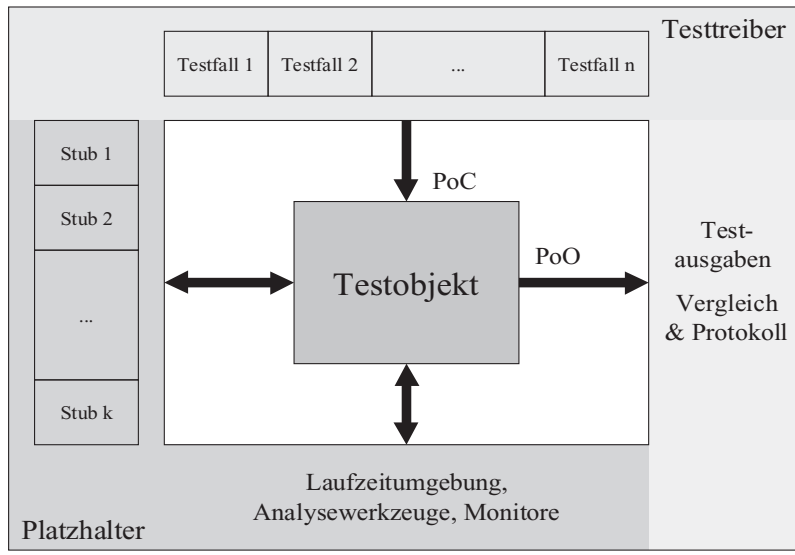



***Abb. 5–1***
*Testrahmen*

*Systematisches Vorgehen bei der Erstellung der Testfälle*

Ziel des Testens ist der Nachweis der Erfüllung der festgelegten Anforderungen durch das implementierte Programmstück und die Aufdeckung von eventuellen Abweichungen und Fehlerwirkungen. Dabei sollen mit möglichst wenig Aufwand möglichst viele Anforderungen überprüft bzw. Fehlerwirkungen nachgewiesen werden. Um diesem Ziel nahe zu kommen, muss ein systematisches Vorgehen bei der Erstellung der Testfälle gewählt werden. Ein unstrukturiertes Testen, meist »aus dem Bauch heraus«, bietet keine Gewähr dafür, dass möglichst viele, wenn es geht, sogar alle Situationen getestet wurden, die vom Testobjekt unterschiedlich verarbeitet werden.

*Black-box- und White-box-Testverfahren*

Es gibt eine ganze Reihe von unterschiedlichen Verfahren zur Überprüfung des Testobjekts, die zu einer systematischen Erstellung der Testfälle verwendet werden können. Die Verfahren lassen sich in zwei Kategorien einteilen: Black-box- und White-box[26]-Verfahren. Präziser ausgedrückt: Black-box- und White-box-Testfallentwurfsverfahren, denn die Verfahren dienen zur Ermittlung der jeweiligen Testfälle.

Bei den Black-box-Verfahren wird das Testobjekt als schwarzer Kasten angesehen. Über den Programmtext und den inneren Aufbau sind keine Informationen notwendig. Beobachtet wird das Verhalten des Testobjekts von außen (→PoO – *Point of Observation* liegt außerhalb des Testobjekts). Es ist keine Steuerung des Ablaufs des Testobjekts au-

26. Manchmal auch als »Glass-box-Verfahren« benannt, weil in einen weißen Kasten auch nicht hineingesehen werden kann. Dieser Begriff hat sich allerdings nicht durchgesetzt.

ßer durch die entsprechende Wahl der Eingabetestdaten möglich (auch der →PoC – *Point of Control* liegt außerhalb des Testobjekts). Die Überlegungen zu den Testfällen beruhen auf der Spezifikation bzw. den Anforderungen an das Testobjekt. Bei den White-box-Verfahren wird auch auf den Programmtext zurückgegriffen. Während der Ausführung der Testfälle wird der innere Ablauf im Testobjekt analysiert (der Point of Observation liegt innerhalb des Testobjekts). Ein Eingriff in den Ablauf des Testobjekts ist in Ausnahmefällen möglich, z. B. wenn Negativtests durchzuführen sind und über die Komponentenschnittstelle die zu provozierende Fehlbedienung nicht ausgelöst werden kann (der Point of Control kann innerhalb des Testobjekts liegen). Testfälle können aufgrund der Programmstruktur des Testobjekts gewonnen werden (s. Abb. 5-2).

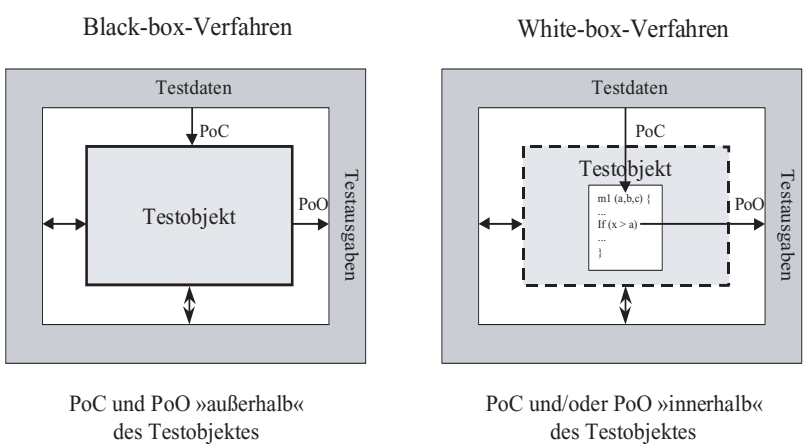



***Abb. 5–2***
*PoC und PoO bei Black-box- und White-box-Verfahren*

Die White-box-Verfahren werden auch als →strukturelle Testverfahren bezeichnet, da sie die Struktur (Komponentenhierarchie, Kontrollfluss, Datenfluss) des Testobjekts berücksichtigen, und die Black-box-Verfahren als funktionale Testverfahren, da die Prüfung des Ein-/Ausgabeverhaltens, also die Funktionalität des Testobjekts, im Mittelpunkt steht.

Die White-box-Testverfahren lassen sich auf den unteren Teststufen anwenden. Ein Systemtest, der sich am Programmtext orientiert, ist in der Regel wenig sinnvoll. Die Black-box-Verfahren eigenen sich überwiegend für die höheren Teststufen, wobei auch beim Komponententest ein Einsatz sinnvoll ist.

## 5.1 Black-box-Verfahren

Da bei Black-box-Verfahren der innere Aufbau des Testobjekts nicht bekannt ist bzw. nicht herangezogen wird, werden Testfälle aus der Spezifikation abgeleitet oder liegen bereits als Teil derselben vor. Ein Test mit allen möglichen Eingabedaten und deren Kombination wäre ein vollständiger Test. Dies ist aber wegen der großen Zahl von möglichen Eingabewerten und Kombinationen unrealistisch. Eine sinnvolle Auswahl aus den möglichen Testfällen muss getroffen werden. Dazu gibt es mehrere Verfahren, die im Folgenden vorgestellt werden.

### 5.1.1 Äquivalenzklassenbildung

*Eingabewertebereiche werden in Äquivalenzklassen unterteilt*

Die Menge der möglichen konkreten Eingabewerte für ein Eingabedatum wird in →Äquivalenzklassen unterteilt (→Äquivalenzklassenbildung). Die zugrunde liegende Idee dabei ist, dass das Testobjekt sich bei allen Eingabedaten, die zu einer Äquivalenzklasse gehören, gleich verhält. Der Test eines Repräsentanten einer Äquivalenzklasse wird als ausreichend angesehen, da davon ausgegangen wird, dass das Testobjekt für alle anderen Eingabewerte derselben Äquivalenzklasse keine andere Reaktion zeigt.

Neben den Äquivalenzklassen, die gültige Eingaben umfassen, sind auch solche für ungültige Eingaben zu berücksichtigen.

---

***Beispiel Äquivalenzklassenbildung***

Das Beispiel zur Berechnung der Weihnachtsgratifikation aus Kapitel 2.2.2 soll hier erneut aufgegriffen werden und den Sachverhalt verdeutlichen. Zur Erinnerung: Eine Firma hat ein Programm in Auftrag gegeben, das die Weihnachtsgratifikation der Mitarbeiter in Abhängigkeit von der Firmenzugehörigkeit berechnen soll. In der Beschreibung der Anforderungen findet sich folgende Textpassage: »Mitarbeiter erhalten ab einer Zugehörigkeit zur Firma von mehr als drei Jahren 50 % des Monatsgehalts als Weihnachtsgratifikation. Mitarbeiter, die länger als fünf Jahre in der Firma tätig sind, erhalten 75 %. Bei einer Firmenzugehörigkeit von mehr als acht Jahren wird eine Gratifikation von 100 % gewährt.«

Es lassen sich sehr einfach vier unterschiedliche Äquivalenzklassen mit gültigen Werten (gültige Äquivalenzklasse, gÄK) für die Berechnung der Gratifikation anhand der Dauer der Firmenzugehörigkeit ableiten:

***Tab. 5–1*** *Gültige Äquivalenzklassen und Repräsentanten*

| Parameter | Äquivalenzklasse | Repräsentant |
|---|---|---|
| Gratifikation | $gÄK_1$: `0 <= x <= 3` | 2 |
| | $gÄK_2$: `3 < x <= 5` | 4 |
| | $gÄK_3$: `5 < x <= 8` | 7 |
| | $gÄK_4$: `x > 8` | 12 |

In Kapitel 2.2.2 wurden die Eingabewerte 2, 4, 7 und 12 (vgl. Tab. 2-2) gewählt. Jeder Wert ist ein Repräsentant aus einer der vier Äquivalenzklassen. Es wird davon ausgegangen, dass Testläufe mit beispielsweise den Eingabewerten 1, 6, 9 und 17 zu keinen weiteren Erkenntnissen führen, also auch keine weiteren Fehlerwirkungen aufdecken und damit nicht durchgeführt werden brauchen.

*Äquivalenzklassen mit ungültigen Werten*

Neben den gültigen Eingabewerten müssen auch ungültige Eingaben beim Test überprüft werden. Es müssen also auch Äquivalenzklassen für ungültige Werte überlegt und Testläufe mit Repräsentanten dieser Klassen durchgeführt werden.

In dem Beispiel sind es die folgenden beiden ungültigen Äquivalenzklassen (uÄK):

***Tab. 5–2*** *Ungültige Äquivalenzklassen und Repräsentanten*

| Parameter | Äquivalenzklasse | Repräsentant |
|---|---|---|
| Gratifikation | `uÄK1: x < 0`<br>(»negative« – also falsche – Betriebszugehörigkeit) | `-3` |
| | `uÄK2: x > 70`<br>(unrealistisch lange, falsche Betriebszugehörigkeit[a]) | `80` |

a. Der Wert 70 ist hier relativ willkürlich gewählt. Es muss mit dem Kunden abgestimmt werden, was für ihn der maximale Wert für die Firmenzugehörigkeit ist.

*Systematische Herleitung der Testfälle*

Um die Testfälle systematisch herzuleiten, wird wie folgt vorgegangen. Für jede zu testende Eingabevariable (z. B. Funktions-/Methodenparameter beim Komponententest oder Maskenfeld beim Systemtest) wird der Definitionsbereich ermittelt. Dieser Definitionsbereich ist die Äquivalenzklasse aller zulässigen bzw. erlaubten Eingabewerte. Diese Werte muss das Testobjekt gemäß der Spezifikation verarbeiten. Die Werte außerhalb des Definitionsbereichs werden als Äquivalenzklassen mit unzulässigen Werten betrachtet. Auch für diese Werte ist zu prüfen, wie das Testobjekt sich verhält.

*Weitere Aufteilung der Äquivalenzklassen*

Als nächster Schritt sind die Äquivalenzklassen zu verfeinern. Äquivalenzklassenelemente, die vom Testobjekt laut Spezifikation unterschiedlich verarbeitet werden, sind einer neuen (Unter-)Äquivalenzklasse zuzuordnen. Die Äquivalenzklassen werden so lange aufgeteilt, bis sich alle unterschiedlichen Anforderungen mit den jeweiligen Äquivalenzklassen decken. Für jede einzelne Äquivalenzklasse ist dann ein Repräsentant für einen Testfall auszuwählen.

Zur Vervollständigung der Testfälle muss zu jedem Repräsentanten auch das erwartete Ergebnis und ggf. die Vorbedingungen für den Testlauf festgelegt werden.

*Äquivalenzklassen für Ausgabewerte*

Das Prinzip der Zerlegung kann genauso für die Ausgabedaten genutzt werden. Die Ermittlung der einzelnen Testfälle ist allerdings aufwändiger, da für jeden »Ausgaberepräsentanten« (Ergebniswert) die ihn erzeugenden Eingabewerte zu ermitteln sind. Auch bei den Ausgabewerten sind die Äquivalenzklassen mit ungültigen Werten nicht zu vernachlässigen.

Die Aufteilung in Äquivalenzklassen und die Wahl der Repräsentanten sind sehr sorgfältig vorzunehmen, da es stark von der Güte der Aufteilung abhängt, welche Testfälle durchgeführt und mit welcher Wahrscheinlichkeit Fehlerwirkungen gefunden werden. In der Regel ist es nicht trivial, die Äquivalenzklassen aus der Spezifikation oder aus anderen Dokumenten herzuleiten.

*Grenzen der Äquivalenzklassen*

Aussichtsreiche Testfälle sind sicherlich solche, die die Grenzen der Äquivalenzklassen prüfen. Oft sind an diesen Stellen Missverständnisse oder Ungenauigkeiten in den Anforderungen gegeben, da aus einer umgangssprachlichen Formulierung nicht (im mathematischen Sinne) genau hervorgeht, welcher Grenzwert einer Äquivalenzklasse zuzuordnen ist oder nicht. Eine umgangssprachliche Formulierung »... mehr als drei Jahre ...« in den Anforderungen kann den exakten Wert 3 innerhalb (ÄK: x >= 3) oder auch außerhalb der Äquivalenzklasse (ÄK: x > 3) meinen. Ein zusätzlicher Testfall mit x=3 kann eine mögliche Fehlinterpretation und damit eine Fehlerwirkung aufdecken. Im Kapitel 5.1.2 wird auf die Analyse der Grenzwerte der Äquivalenzklassen ausführlich eingegangen.

---

***Beispiel: Äquivalenzklassenbildung für ganzzahlige Eingabewerte***

Zur Verdeutlichung des Vorgehens bei der Bildung der Äquivalenzklassen sollen die möglichen Äquivalenzklassen für einen ganzzahligen Eingabewert ermittelt werden. Für den ganzzahligen Parameter `extras` der Methode `calculate_price()` ergeben sich folgende Äquivalenzklassen:

***Tab. 5–3*** *Äquivalenzklassen für ganzzahlige Eingabewerte*

| Parameter | Äquivalenzklasse |
|---|---|
| `extras` | `gÄK`$_1$`: [MIN_INT, ... , MAX_INT]`[a]<br>`uÄK`$_1$`: NaN` |

a. Mit MIN_INT und MAX_INT ist jeweils die kleinste und größte ganze darstellbare Zahl im Rechner gemeint. Diese können je nach verwendeter Hardware von Rechner zu Rechner unterschiedlich sein.

Dabei ist darauf zu achten, dass der Wertebereich im Gegensatz zur reinen Mathematik im Rechner durch die Maxima und Minima beschränkt ist.

Über- oder Unterschreitungen dieser Werte führen häufig zu Fehlerwirkungen, da mögliche Überschreitungen nicht abgefangen werden.

Die Äquivalenzklasse unzulässiger Werte ergibt sich aus folgender Überlegung: Unzulässige Werte sind Zahlen, die größer oder kleiner sind als die jeweiligen Intervallgrenzen, oder sämtliche nicht numerischen Werte[27]. Wird angenommen, dass die Reaktion der Methode auf einen unzulässigen Wert immer gleich ist (z. B. eine Ausnahmebehandlung, die den Fehlercode `NOT_VALID` liefert), genügt es, alle möglichen Arten unzulässiger Werte auf eine gemeinsame Äquivalenzklasse (hier bezeichnet mit `NaN` für »Not a Number«) abzubilden.

Da negative und positive Werte erfahrungsgemäß unterschiedlich behandelt werden, macht es Sinn, die Äquivalenzklasse mit den gültigen Werten ($gÄK_1$) weiteraufzuteilen. Auch ist die Null ein Eingabewert, der häufig zu Fehlerwirkungen führt.

| Parameter | Äquivalenzklasse | Repräsentant |
|---|---|---|
| `extras` | $gÄK_1$: `[MIN_INT,...,0[`[a]<br>$gÄK_2$: `[0,...,MAX_INT]`<br>$uÄK_1$: `NaN` | `-123`<br>`654`<br>`"f"` |

a. ´[´ bezeichnet ein offenes Intervall ohne Intervallgrenze.

***Tab. 5–4*** *Äquivalenzklassen und Repräsentanten für ganzzahlige Eingabewerte*

Als Repräsentant wurde relativ willkürlich je ein Wert aus den insgesamt drei Äquivalenzklassen gewählt. Hinzu kommen noch die Grenzwerte (s. a. Kap. 5.1.2) der jeweiligen Äquivalenzklassen: `MIN_INT, 0, MAX_INT.` Für die Äquivalenzklasse der ungültigen Werte gibt es im engeren Sinne keine Grenzwerte.

Damit ergeben sich nach der Äquivalenzklassenmethode mit Berücksichtigung der Grenzwerte für den Test des ganzzahligen Parameters `extras` folgende sechs zu prüfenden Werte:

```
{"f", MIN_INT, -123, 0, 654, MAX_INT}.
```

Zu jedem dieser Eingabewerte sind die erwarteten Ausgaben bzw. Reaktionen des Testobjekts festzulegen, um nach dem Testlauf entscheiden zu können, ob ggfs. eine Fehlerwirkung vorliegt.

27. Es hängt von der gewählten Programmiersprache und dem eingesetzten Compiler ab, ob und welche unzulässigen Werte der Compiler erkennt, z. B. beim Versuch, die Methode vom Testtreiber aufzurufen. Im Beispiel wird angenommen, dass der Compiler unzulässige Parameterwerte nicht erkennt und deren Verarbeitung daher im dynamischen Test mit untersucht werden muss.

*Äquivalenzklassen von Eingabedaten, die keine Basisdatentypen sind*

Bei den im Beispiel verwendeten Eingabedaten mit ganzzahligem Wertebereich lassen sich sehr leicht Äquivalenzklassen bilden und die entsprechenden Repräsentanten auswählen. Es können als Wertebereiche außer den elementaren Datentypen auch zusammengesetzte Datenstrukturen oder Objektmengen zugrunde gelegt werden. Es ist dann jeweils zu entscheiden, mit welchen Repräsentanten ein Testfall durchzuführen ist.

***Beispiel für Eingabedaten, die aus einer Menge auszuwählen sind***

Folgendes Beispiel soll den Sachverhalt verdeutlichen. Eine reisende Person kann ein Kind, ein Jugendlicher, ein Erwachsener, ein Student, ein Sozialhilfeempfänger oder ein Rentner sein. Wenn das Testobjekt auf die jeweilige Ausprägung der reisenden Person unterschiedlich reagiert bzw. reagieren sollte, dann sind alle Möglichkeiten mit einem extra Testfall zu überprüfen. Wird kein unterschiedliches Verhalten gefordert oder erwartet, kann auch ein Testfall ausreichend sein und ein beliebiger der möglichen Werte für eine reisende Person gewählt werden.

Ist das Testobjekt die Komponente, die den Fahrpreis ermittelt, so sind sicherlich sechs unterschiedliche Testfälle für die reisende Person vorzusehen. Der Fahrpreis wird bei den unterschiedlichen reisenden Personen vermutlich unterschiedlich berechnet. Genaues muss aus den Anforderungen hervorgehen. Jede Berechnung ist einem Test zu unterziehen, um die Korrektheit der Berechnung zu überprüfen und ggfs. Fehlerwirkungen nachzuweisen.

Beim Test der Komponente, die die Platzreservierung vornimmt, kann es ausreichend sein, dass für eine reisende Person nur ein Repräsentant, zum Beispiel ein Erwachsener, gewählt wird. Vermutlich – auch hier muss Genaues in den Anforderungen zum Sachverhalt definiert sein – spielt es bei der Platzreservierung keine Rolle, ob der Platz von einem Jugendlichen oder von einem Rentner eingenommen wird. Dem Tester muss aber klar sein, dass er, wenn er nur den Testfall mit dem Eingabedatum »Erwachsener« durchführt, keinerlei Aussagen treffen kann, ob die Platzreservierung auch für die anderen fünf reisenden Personengruppen ordnungsgemäß durchgeführt wird.

***Tipp zur Ermittlung der Äquivalenzklassen***

Als Richtlinie für die Ermittlung der Äquivalenzklassen können folgende Hinweise dienen:

- Ermitteln der spezifizierten Einschränkungen und Bedingungen aus der Spezifikation sowohl für die Eingaben als auch für die Ausgaben.
- Für jede Einschränkung bzw. Bedingung ist die Äquivalenzklassenbildung vorzunehmen:
  - Ist ein zusammenhängender Wertebereich spezifiziert, dann sind eine gültige und zwei ungültige Äquivalenzklassen zu berücksichtigen.
  - Ist spezifiziert, dass eine Anzahl von Werten einzugeben ist, sind eine gültige (mit allen möglichen gültigen Werten) und zwei ungültige Äqui-

valenzklassen (Unter- und Überschreitung der gültigen Anzahl) zu bilden.
  - Ist eine Menge von Werten spezifiziert, die möglicherweise unterschiedlich zu behandeln sind, so ist für jeden Wert der Menge eine gültige Äquivalenzklasse (bestehend aus diesem einen Wert) vorzusehen und eine zusätzliche ungültige Äquivalenzklasse.
  - Falls die Einschränkung oder Bedingung eine Situation beschreibt, die zwingend erfüllt werden muss, ist jeweils eine gültige und ungültige Äquivalenzklasse zu berücksichtigen.
- Bestehen Zweifel an der Gleichbehandlung von Werten innerhalb einer Äquivalenzklasse, sollte die Äquivalenzklasse weiter unterteilt werden.

---

**Testfälle**

*Kombination der Repräsentanten*

In der Regel besitzt ein Testobjekt mehr als nur einen Eingabeparameter. Die Äquivalenzklassenmethode liefert für jeden einzelnen dieser Parameter des Testobjekts mindestens zwei Äquivalenzklassen (eine gültige und eine ungültige). Damit gibt es je Parameter mindestens zwei Repräsentanten, die als Testeingaben zu verwenden sind.

Zur Spezifikation eines Testfalls muss jedem Parameter ein Eingabewert zugeordnet werden. Dazu muss entschieden werden, welche der verfügbaren Repräsentanten miteinander zu einem Eingabedatensatz zu kombinieren sind. Um alle (durch die vorgenommene Äquivalenzklassenzerlegung modellierten) Testobjektreaktionen sicher auszulösen, sind die Eingabewerte, also die Repräsentanten der jeweiligen Äquivalenzklassen, nach folgenden Regeln zu kombinieren:

*Regeln zur Testfallerstellung*

- Die Repräsentanten aller gültigen Äquivalenzklassen sind zu Testfällen zu kombinieren, d. h., alle möglichen Kombinationen der jeweiligen Repräsentanten sind vorzusehen. Jede dieser Kombinationen bildet einen »gültigen Testfall«.
- Der Repräsentant einer ungültigen Äquivalenzklasse ist nur mit Repräsentaten von anderen gültigen Äquivalenzklassen zu kombinieren. Für jede ungültige Äquivalenzklasse ist somit ein extra »Negativ«-Testfall zu spezifizieren.

*Einschränkung der Testfallmenge*

Die Anzahl der »gültigen« Testfälle ergibt sich also aus dem Produkt der Zahl gültiger Äquivalenzklassen je Parameter. Wegen dieser multiplikativen Kombination können sich schon bei wenigen Parametern sehr schnell einige hundert »gültige Testfälle« ergeben. Da es selten machbar ist, so viele Testfälle zu beachten, werden weitere Regeln benötigt, wie die Menge »gültiger« Testfälle reduziert werden kann:

*Regeln zur Testfalleinschränkung*

- Die Testfälle aus allen Repräsentanten kombinieren und anschließend nach »Häufigkeit« sortieren (typische Benutzungsprofile). Testfälle in dieser Reihenfolge priorisieren. Zum Test herangezogen werden so nur die »benutzungsrelevanten« Testfälle (oder häufig vorkommende Kombinationen oder Benutzungen).
- Es werden Testfälle bevorzugt, die Grenzwerte oder Grenzwertkombinationen enthalten.
- Sicherstellen, dass jeder Repräsentant einer Äquivalenzklasse mit jedem Repräsentanten der anderen Äquivalenzklassen in einem Testfall zur Ausführung kommt (d. h. paarweise Kombination statt vollständiger Kombination).
- Als Minimalkriterium sicherstellen, dass jeder Repräsentant einer Äquivalenzklasse in mindestens einem Testfall vorkommt.
- Repräsentanten ungültiger Äquivalenzklassen nicht mit Repräsentanten anderer ungültiger Äquivalenzklassen kombinieren.

Die Repräsentanten ungültiger Äquivalenzklassen werden nicht »multiplikativ« kombiniert. Ein ungültiger Wert soll nur mit »gültigen« kombiniert werden. Denn ein ungültiger Parameterwert löst eine Ausnahmebehandlung aus, unabhängig davon, welche Werte die übrigen Parameter haben. Kombiniert ein Testfall mehrere ungültigen Werte, kann es leicht zu einer gegenseitigen Fehlermaskierung kommen, und nur eine der möglichen Ausnahmesituationen wird ausgelöst. Auch ist dann beim Auftreten einer Fehlerwirkung nicht klar, welcher ungültige Wert die Wirkung ausgelöst hat. Vermeidbarer Aufwand zur Fehleranalyse ist dann notwendig.

---

***Beispiel: Test der DreamCar-Preisberechnung***

Im Folgenden dient wieder die Methode `calculate_price()` des VSR-Teilsystems *DreamCar* als Testobjekt (s. Kap. 3.2.3). Zu testen ist, ob die Methode aus ihren Eingabedaten stets einen gemäß Spezifikation korrekten Gesamtpreis berechnet. Es wird angenommen, dass der innere Aufbau der Methode nicht bekannt ist, sondern nur die funktionale Spezifikation der Methode und die Methodenschnittstelle:

```
double calculate_price (
double baseprice,      // Grundpreis des Fahrzeugs
double specialprice,   // Sondermodellaufschlag
double extraprice,     // Preis der Zusatzausstattung
int extras,            // Anzahl Zusatzausstattungen
double discount        // Haendlerrabatt
)
```

Zur Herleitung der nötigen Testfälle aus den Eingabeparametern wird die Äquivalenzklassenmethode eingesetzt. Im ersten Schritt wird für jeden Eingabeparameter sein Definitionsbereich ermittelt. Hieraus resultieren je Parameter Äquivalenzklassen gültiger und ungültiger Werte:

*Schritt 1: Definitionsbereich ermitteln*

***Tab. 5–5*** *Gültige und ungültige Äquivalenzklassen der Parameter der Methode*

| Parameter | Äquivalenzklasse |
|---|---|
| `baseprice` | gÄK$_{11}$: `[MIN_DOUBLE, ... , MAX_DOUBLE]`<br>uÄK$_{11}$: `NaN` |
| `specialprice` | gÄK$_{21}$: `[MIN_DOUBLE, ... , MAX_DOUBLE]`<br>uÄK$_{21}$: `NaN` |
| `extraprice` | gÄK$_{31}$: `[MIN_DOUBLE, ... , MAX_DOUBLE]`<br>uÄK$_{31}$: `NaN` |
| `extras` | gÄK$_{41}$: `[MIN_INT, ... , MAX_INT]`<br>uÄK$_{41}$: `NaN` |
| `discount` | gÄK$_{51}$: `[MIN_DOUBLE, ... , MAX_DOUBLE]`<br>uÄK$_{51}$: `NaN` |

Damit wurden, alleine aus der Schnittstellenspezifikation, je Parameter eine gültige und eine ungültige Äquivalenzklasse abgeleitet (Testdatengeneratoren gehen entsprechend vor; s. Kap. 7.1.2)

*Schritt 2: Äquivalenzklassen verfeinern, basierend auf der Spezifikation*

Um diese Äquivalenzklassen weiter zu zerlegen, wird Information über die Funktionalität der Methode benötigt. Diese Information liefert die Methoden-Spezifikation (s. Kap. 3.2.3.). Aus dieser können folgende testrelevanten Aussagen gefolgert werden:

- Die Parameter 1 bis 3 sind (Fahrzeug-)Preise. Preise sind nicht negativ. Die Spezifikation legt keine Preisgrenzen fest.
- Abhängig vom Wert `extras` berechnet sich der Rabatt auf die Zusatzausstattung (10 % falls `extras` >=3 bzw. 15 % falls `extras` >=5). `extras` stellt die Anzahl der gewählten Zusatzausstattungsteile dar und ist somit nicht negativ. Die Spezifikation legt keine Anzahlobergrenze fest.
- Der Parameter `discount` ist ein Rabatt, er wird prozentual angegeben und liegt zwischen 0 und 100. Da im Spezifikationstext die Zubehörrabattgrenzen in Prozent angegeben sind, kann der Tester annehmen, dass auch der Händlerrabatt prozentual eingegeben wird. Sicherheit gibt eine Rücksprache beim Kunden.

Diese Überlegungen basieren nicht nur auf der funktionalen Spezifikation. Vielmehr bringt die Analyse einige Lücken in der Spezifikation ans Licht. Diese Lücken »füllt« der Tester, indem er plausible Annahmen trifft, basierend auf Alltagswissen und seiner Testerfahrung. In Zweifelsfällen ist eine Rücksprache beim Kunden ratsam. Aufgrund der Analyse können die bereits gefundenen Äquivalenzklassen verfeinert werden. Je feiner die Äquivalenzklassenzerlegung, umso genauer wird der Test. Sind alle aus der Spezifikation

ersichtlichen Bedingungen für die einzelnen Eingabeparameter und auch das Erfahrungswissen des Testers berücksichtigt, ist die Zerlegung abgeschlossen.

***Tab. 5–6***
*Weitere Aufteilung der Äquivalenzklassen der Parameter der Methode `calculate_price()` mit Repräsentanten*

| Parameter | Äquivalenzklasse | Repräsentant |
|---|---|---|
| baseprice | $gÄK_{11}$: [0,..., MAX_DOUBLE] | 20000.00 |
| | $uÄK_{11}$: [MIN_DOUBLE,...,0[<br>$uÄK_{12}$: NaN | -1.00<br>"abc" |
| specialprice | $gÄK_{21}$: [0,..., MAX_DOUBLE] | 3450.00 |
| | $UÄK_{21}$: [MIN_DOUBLE,...,0[<br>$UÄK_{22}$: NaN | -1.00<br>"abc" |
| extraprice | $gÄK_{31}$: [0,..., MAX_DOUBLE] | 6000.00 |
| | $uÄK_{31}$: [MIN_DOUBLE,...,0[<br>$uÄK_{32}$: NaN | -1.00<br>"abc" |
| extras | $gÄK_{41}$: [0,...,2]<br>$gÄK_{42}$: [3,4]<br>$gÄK_{43}$: [5,..., MAX_INT] | 1<br>3<br>20 |
| | $uÄK_{41}$: [MIN_INT,...,0[<br>$uÄK_{42}$: NaN | -1<br>"abc" |
| discount | $gÄK_{51}$: [0,...,100] | 10.00 |
| | $uÄK_{51}$: [MIN_DOUBLE,...,0[<br>$uÄK_{52}$: ]100,...,MAX_DOUBLE]<br>$uÄK_{53}$: NaN | -1.00<br>101.00<br>"abc" |

Als Resultat ergeben sich insgesamt 18 Äquivalenzklassen, 7 für gültige Parameterwerte und 11 für ungültige.

*Schritt 3: Repräsentanten auswählen*

Um Eingabedaten zu erhalten, muss für jede Äquivalenzklasse ein Repräsentant ausgewählt werden. Dazu kann laut Äquivalenzklassentheorie jeder beliebige Wert einer Äquivalenzklasse herangezogen werden. In der Praxis gelingt die Zerlegung aber selten perfekt. Aus Mangel an Detailinformation, aus Zeitmangel oder einfach weil der Aufwand gescheut wird, wird die Zerlegung auf einer bestimmten Stufe abgebrochen. Einzelne Äquivalenzklassen überschneiden sich eventuell[28]. Bei der Repräsentantenwahl muss daher beachtet werden, dass es innerhalb einer Äquivalenzklasse Werte geben kann, auf die das Testobjekt doch unterschiedlich reagieren könnte, oder Werte, die im Produkteinsatz häufiger vorkommen als andere.

Im Fallbeispiel werden deshalb aus den gültigen Äquivalenzklassen Repräsentanten ausgesucht, die plausible Werte darstellen und im Produkteinsatz vermutlich häufig vorkommende Fälle abdecken. Für die ungültigen

28. Im Idealfall sind die ermittelten Klassen (wie Äquivalenzklassen in der Mathematik) überschneidungsfrei (disjunkt). Das Zerlegungsverfahren garantiert dies aber nicht.

Äquivalenzklassen werden möglichst »unkomplizierte« Repräsentanten ausgesucht. Die gewählten Repräsentanten sind in obiger Tabelle aufgeführt.

*Schritt 4: Testfälle kombinieren*

Im nächsten Schritt müssen die Repräsentanten zu Testfällen kombiniert werden. Nach obigen Grundregeln ergeben sich 1*1*1*3*1 = 3 »gültige« Testfälle und 2+2+2+2+3 = 11 Negativtests.

| | Parameter | | | | | |
|---|---|---|---|---|---|---|
| **Testfall** | **baseprice** | **special-price** | **extra-price** | **extras** | **discount** | **result** |
| 1 | 20000.00 | 3450.00 | 6000.00 | 1 | 10.00 | 27450.00 |
| 2 | 20000.00 | 3450.00 | 6000.00 | 3 | 10.00 | 26850.00 |
| 3 | 20000.00 | 3450.00 | 6000.00 | 20 | 10.00 | 26550.00 |
| 4 | -1.00 | 3450.00 | 6000.00 | 1 | 10.00 | NOT_VALID |
| 5 | "abc" | 3450.00 | 6000.00 | 1 | 10.00 | NOT_VALID |
| 6 | 20000.00 | -1.00 | 6000.00 | 1 | 10.00 | NOT_VALID |
| 7 | 20000.00 | "abc" | 6000.00 | 1 | 10.00 | NOT_VALID |
| 8 | 20000.00 | 3450.00 | -1.00 | 1 | 10.00 | NOT_VALID |
| 9 | 20000.00 | 3450.00 | "abc" | 1 | 10.00 | NOT_VALID |
| 10 | 20000.00 | 3450.00 | 6000.00 | -1 | 10.00 | NOT_VALID |
| 11 | 20000.00 | 3450.00 | 6000.00 | "abc" | 10.00 | NOT_VALID |
| 12 | 20000.00 | 3450.00 | 6000.00 | 1 | -1.00 | NOT_VALID |
| 13 | 20000.00 | 3450.00 | 6000.00 | 1 | 101.00 | NOT_VALID |
| 14 | 20000.00 | 3450.00 | 6000.00 | 1 | "abc" | NOT_VALID |

***Tab. 5–7***
*Testfälle für die Methode* `calculate_price()`

Bei den gültigen Äquivalenzklassen sind jeweils die gleichen Repräsentanten verwendet worden, um sicherzustellen, dass nur die Veränderung des einen Parameters die Reaktion des Testobjekts bewirkt.

Da vier der fünf Parameter nur jeweils eine gültige Äquivalenzklasse aufweisen, ergeben sich nur wenige »gültige« Testfälle. Es besteht kein Anlass, die Testfallmenge weiter zu reduzieren.

Sind die Testfälle aufgestellt, muss zu jedem Testfall das Sollergebnis ermittelt werden. Im Falle obiger Negativtests ist dies einfach. Hier muss nur der vom Testobjekt zu generierende Fehlercode eingetragen werden. Bei den »gültigen« Tests muss (z. B. per Tabellenkalkulation) das erwartete Ergebnis berechnet werden.

### Festlegung der Testendekriterien

Ein Endekriterium für den Test nach der Äquivalenzklassenbildung lässt sich anhand der durchgeführten Tests der Repräsentanten der jeweiligen Äquivalenzklassen im Verhältnis zur Gesamtzahl aller definierten Äquivalenzklassen festlegen:

ÄK-Überdeckung = (Anzahl getestete ÄK / Gesamtzahl ÄK) * 100 %

Sind wie im Beispiel 18 Äquivalenzklassen aus den Anforderungen bzw. der Spezifikation für ein Eingabedatum ermittelt worden und sind von diesen 18 nur 15 in den Testfällen getestet worden, so ist eine Äquivalenzklassen-Überdeckung von 83 % erreicht.

ÄK-Überdeckung = (15 / 18) * 100% = 83,33 %

*Überdeckungsgrad legt Testintensität fest*

Je intensiver ein Testobjekt getestet werden soll, je höher ist der zu erreichende Überdeckungsgrad anzusetzen. Er dient dabei vor der Testausführung zur Festlegung des Kriteriums, wann die Testintensität als ausreichend angesehen werden soll, und nach der Testdurchführung zur Prüfung, ob das geforderte Ziel erreicht wurde.

Wird in dem oben gegebenen Beispiel eine Überdeckung der Äquivalenzklassen von 80 % in der Testplanung festgelegt, so ist diese mit dem Test von 14 der insgesamt 18 Äquivalenzklassen erreicht, und der Test nach Äquivalenzklassenbildung kann beendet werden. Ein messbares Kriterium zur Beendigung der Tests ist somit gegeben.

An dem Beispiel wird auch deutlich, wie wichtig die Ermittlung der jeweiligen Äquivalenzklassen ist. Werden nicht alle Äquivalenzklassen erkannt und somit zu wenige ermittelt und mit Repräsentanten getestet, wird zwar ein hoher Überdeckungsgrad erreicht, der aber aufgrund einer falschen Gesamtzahl der Äquivalenzklassen berechnet wird. Das vermeintlich gute Ergebnis spiegelt aber nicht die tatsächliche Testintensität wider. Die Testfallermittlung mithilfe der Äquivalenzklassenbildung ist nur so gut, wie sorgfältig die Bildung der Äquivalenzklassen vorgenommen wurden.

### Bewertung der Methode

Die Systematik bei der Äquivalenzklassenbildung trägt dazu bei, dass spezifizierte Bedingungen und Einschränkungen beim Test nicht übersehen und keine →unnützen Testfälle durchgeführt werden. Unnütz sind Testfälle, deren Daten aus derselben Äquivalenzklasse kommen und somit ein gleiches Verhalten des Testobjekts bewirken.

Es werden allerdings nur einzelne Ein- oder Ausgabebedingungen betrachtet, mögliche Abhängigkeiten oder Wechselwirkungen zwi-

schen den Bedingungen werden nicht berücksichtigt bzw. sind sehr aufwändig zu berücksichtigen (durch weitere Aufteilung der Äquivalenzklassen und entsprechende Kombinationen).

In Kombination mit fehlerorientierten Verfahren, wie der Grenzwertanalyse, ist die Äquivalenzklassenanalyse jedoch ein sehr wirkungsvolles Verfahren.

### 5.1.2 Grenzwertanalyse

*Sinnvolle Ergänzung*

Die →Grenzwertanalyse liefert eine sehr sinnvolle Ergänzung zu den Testfällen, die durch die Äquivalenzklassen ermittelt wurden. Denn Fehlerzustände in Programmen treten häufig an den Grenzbereichen der Äquivalenzklassen auf, da hier fehlerträchtige Fallunterscheidungen in den Programmen vorherzusehen sind. Ein Test mit Grenzwerten deckt daher oft Fehlerwirkungen auf. Die Methode lässt sich nur anwenden, wenn die Menge der Daten, die in eine Äquivalenzklasse fallen, geordnet ist und sich Grenzen identifizieren lassen.

Bei der Grenzwertanalyse werden die »Ränder« der Äquivalenzklassen einer Überprüfung unterzogen. An jedem Rand wird der exakte Grenzwert und die beiden (innerhalb und außerhalb der Äquivalenzklasse) benachbarten Werte getestet. Dabei sollte das kleinste mögliche Inkrement in beiden Richtungen verwendet werden, um die Grenzen einem genauen Test zu unterziehen. Für jede Grenze ergeben sich somit drei Testfälle. Fällt die obere Grenze einer Äquivalenzklasse mit der unteren Grenze der benachbarten Äquivalenzklasse zusammen, dann fallen auch die entsprechenden Testfälle zusammen.

---

***Beispiel: ganzzahlige Eingabe***

Für das oben angegebene Beispiel für den Test eines ganzzahligen Eingabewertes ergeben sich somit weitere Testfälle mit folgenden Testeingabedaten; sechs weitere kommen hinzu, so dass es insgesamt zwölf Testfälle sind:

```
{"f",
MIN_INT-1, MIN_INT, MIN_INT+1,
-123,
-1, 0, 1,
654,
MAX_INT-1, MAX_INT, MAX_INT+1}
```

Der Testfall mit dem Eingabewert `-1` prüft den größten Wert der Äquivalenzklasse $ÄK_1$: `[MIN_INT, ... , 0]`, der kleiner als der obere Grenzwert `(0)` selbst ist. Dieser Testfall prüft aber auch die kleinste Unterschreitung der unteren Grenze `(0)` der Äquivalenzklasse $ÄK_2$: `[0, ... , MAX_INT]`. Von dieser Äquivalenzklasse aus betrachtet liegt der Wert außerhalb. Entsprechendes gilt für den Eingabewert `1`. Zu beachten ist hierbei, dass Werte ober-

halb der oberen Grenze bzw. unterhalb der unteren Grenze aus technischen Gründen manchmal nicht als konkrete Eingabewerte eingegeben werden können.

Im Beispiel sind nur die Testwerte für die Eingabevariable angegeben. Damit die Testfälle komplett sind, muss zu jedem der zwölf einzelnen Werte das jeweils erwartete Verhalten des Testobjekts bzw. die jeweils erwarteten Ausgaben unter Verwendung des Testorakels sowie Vor- und Nachbedingungen aufgelistet werden.

*Testaufwand gerechtfertigt?*

Auch hier ist wieder zu entscheiden, ob der Testaufwand gerechtfertigt ist und jede Grenze mit den benachbarten Werten jeweils für einen extra Testfall berücksichtigt werden soll. Es können auch die Testfälle mit den Repräsentanten der Äquivalenzklassen weggelassen werden, die keine Grenzen prüfen. Im Beispiel wären das die Testfälle mit den Eingabetestdaten `-123` und `654`. Es wird dann davon ausgegangen, dass die Testfälle mit den Werten aus der Mitte einer Äquivalenzklasse keine zusätzlichen Erkenntnisse liefern, da Testfälle mit dem maximalen und dem minimalem Wert innerhalb der Äquivalenzklasse in jeweils einem Testfall überprüft werden. Im Beispiel sind das die innerhalb der Äquivalenzklasse liegenden Werte `MIN_INT+1`, `-1` und `1`, `MAX_INT-1`.

---

*Grenzwerte bei Mengen nicht vorhanden*

Für das oben erwähnte Beispiel mit dem Eingabedatum »reisende Person« lassen sich keine Grenzen für den Eingabewertebereich angeben. Die Möglichkeiten für das Eingabedatum sind im mathematischen Sinne aus einer Menge zu wählen, die aus sechs Elementen (Kind, Jugendlicher, Erwachsener, Student, Sozialhilfeempfänger, Rentner) besteht. Grenzen lassen sich hier nicht identifizieren. Eine mögliche Ordnung über das Alter der Personen ist nicht eindeutig festzulegen, da beispielsweise Sozialhilfeempfänger jedes Alter haben können.

Die Grenzwertanalyse kann selbstverständlich auch für solche Äquivalenzklassen angewendet werden, die für Ausgabedaten gebildet wurden.

### Testfälle

In Analogie zur Testfallermittlung bei der Äquivalenzklassenbildung können die gültigen Grenzen innerhalb einer Äquivalenzklasse miteinander zu Testfällen kombiniert werden. Die ungültigen Grenzen sind wieder separat zu überprüfen und können nicht mit anderen ungültigen Grenzen kombiniert werden.

Wie im Beispiel oben erwähnt, müssen die Werte aus dem mittleren Bereich einer Äquivalenzklasse nicht für einen extra Testfall verwendet werden, wenn die beiden Randwerte innerhalb der Äquivalenzklasse für Testfälle vorgesehen sind.

Wird die Grenzwertanalyse auf die gültigen Äquivalenzklassen zur Prüfung der Methode `calculate_price()` angewendet, ergeben sich die in folgender Tabelle aufgelisteten grenzwertbasierten Testdaten je Parameter:

***Beispiel: Grenzwertetests für*** *`calculate_price()`*

***Tab. 5–8*** *Grenzwerte der Parameter der Methode* *`calculate_price()`*

| Parameter | (untere und obere) Grenzwerte |
|---|---|
| baseprice | 0-δ[a]/0/0+δ MAX_DOUBLE-δ/MAX_DOUBLE/MAX_DOUBLE+δ |
| specialprice | s.o. |
| extraprice | s.o. |
| extras | -1/0/1 1/2/3<br>2/3/4 3/4/5<br>4/5/6 MAX_INT–1/MAX_INT/MAX_INT+1 |
| discount | 0-δ/0/0+δ 100-δ/100/100+δ |

a. Die berücksichtigende Genauigkeit δ hängt von der Aufgabenstellung und der Zahlendarstellung des Rechners ab.

Nach Streichung von Wiederholungen der Grenzwerte (`2, 3, 4, 5`) und solchen Grenzwerten, die in ungültigen Äquivalenzklassen liegen (`0-δ, MAX_DOUBLE+δ; -1, MAX_INT+1; 0-δ, 100+δ`), ergeben sich 4 + 4 + 4 + 9 + 4 = 25 grenzwertbasierte Repräsentanten, von denen 2 (`extras: 1, 3`) schon aus der ursprünglichen Äquivalenzklassenzerlegung bekannt sind. Ein erfahrener Tester wird von den 23 neuen Repräsentanten folgende als »interessant« betrachten und zur Spezifikation weiterer Testfälle nutzen:

***Tab. 5–9*** *Weitere Testfälle für die Methode* *`calculate_price()`*

| | Parameter | | | | | |
|---|---|---|---|---|---|---|
| **Test-fall** | **baseprice** | **special-price** | **extra-price** | **extras** | **discount** | **result** |
| 15 | 20000.00 | 3450.00 | 6000.00 | 0 | 0.00 | 29450.00 |
| 16 | 20000.00 | 3450.00 | 6000.00 | 5 | 0.00 | 28550.00 |
| 17 | 20000.00 | 3450.00 | 6000.00 | 0 | 100.00 | 9459.00 |
| 18 | MAX_DOUBLE | MAX_DOUBLE | MAX_DOUBLE | 0 | 0.00 | >MAX_DOUBLE |
| 19 | MAX_DOUBLE | 3450.00 | 6000.00 | 0 | 0.00 | >MAX_DOUBLE |
| 20 | 20000.00 | MAX_DOUBLE | 6000.00 | 0 | 0.00 | >MAX_DOUBLE |
| 21 | 20000.00 | 3450.00 | MAX_DOUBLE | 0 | 0.00 | >MAX_DOUBLE |
| 22 | 0.00 | 3450.00 | 6000.00 | 0 | 0.00 | NOT_VALID |
| 23 | 20000.00 | 0.00 | 6000.00 | 0 | 0.00 | 26000.00 |
| 24 | 20000.00 | 3450.00 | 0.00 | 0 | 0.00 | 23450.00 |
| 25 | 0.00 | 0.00 | 0.00 | 0 | 0.00 | NOT_VALID |

- Testfall 15 prüft, ob ein Preis ohne Sonderausstattung korrekt berechnet wird
- Testfall 16 prüft die 15%-Rabattgrenze
- Testfall 17 prüft, ob ein Fahrzeug kostenlos abgegeben werden kann
- Testfall 18 prüft auf Wertebereichsüberlauf nach Addition der Einzelpreise
- Die Testfälle 19 bis 21 prüfen Wertebereichsüberlauf durch einen Parameter
- Die Testfälle 22 bis 24 prüfen, ob Teilpreise »null« sein dürfen
- Testfall 25 prüft, ob ein Gesamtpreis »null« erkannt wird.

Die 23 zusätzlich gefundenen Repräsentanten sind zu 11 weiteren Testfällen kombiniert worden. Das Risiko, einen Defekt in der Preisberechnung im Test zu übersehen, erscheint dem Tester damit als gering. Er entscheidet, dass keine weiteren Testfälle benötigt werden.

***Tipp zur Testfallerstellung nach der Grenzwertanalyse***

- Bei einem Eingabewertebereich sind die Grenzen und die benachbarten Werte außerhalb des Bereichs heranzuziehen. Bereich: [-1.0; +1.0], Testdaten: -1.0, +1.0 und -1.001, +1.001[29].
- Sind für eine Eingabedatei die Anzahl der Werte vorgegeben, so ergeben sich nach der Grenzwertanalyse folgende Überlegungen: Eingabedatei hat zwischen 1 und 100 Eingabesätze, Testdaten: 1, 100 und 0, 101.
- Dienen Ausgabebereiche als Grundlage, kann beispielsweise wie folgt vorgegangen werden: Die Ausgabe des Testobjekts ist ein ganzzahliger Wert zwischen 500 und 1000. Zu erzielende Ergebnisse: 500, 1000 und 499, 1001. Es kann allerdings einen gewissen Aufwand erfordern, die entsprechenden Eingabetestdaten zu ermitteln, um genau die geforderten Ausgaben zu erhalten.
- Ist bei den Ausgabewerten die erlaubte Anzahl entscheidend, ist in Analogie zur Anzahl bei den Eingabewerten zu verfahren: Ausgabe von 1 bis 4 Daten, zu erzeugende Ausgaben: 1, 4 und 0, 5 Daten.
- Bei geordneten Mengen sind das erste und das letzte Element für den Test von besonderem Interesse.
- Sind als Ein- oder Ausgaben komplexe Datenstrukturen gegeben, so können beispielsweise eine leere Liste oder die Nullmatrix als Grenzwerte angesehen und für den Test berücksichtigt werden.
- Bei numerischen Berechnungen sollten als Grenzen sowohl eng beieinander liegende Werte als auch weit entfernte Werte für die Testfallermittlung herangezogen werden.
- Für ungültige Äquivalenzklassen ist eine Grenzwertanalyse nur dort sinnvoll, wo abhängig von einer Äquivalenzklassengrenze eine unterschiedliche Ausnahmebehandlung des Testobjekts erwartet wird.

29. Die zu wählende Genauigkeit hängt von der spezifizierten Aufgabe ab.

#### Festlegung der Testendekriterien

In Analogie zum Testendekriterium bei der Äquivalenzklassenbildung lässt sich auch eine anzustrebende Überdeckung der Grenzwerte (GW) vorab festlegen und nach der Durchführung der Tests berechnen.

GW-Überdeckung = (Anzahl getestete GW / Gesamtzahl GW) * 100 %

Wobei zu beachten ist, dass nicht nur der direkte Grenzwert, sondern auch die jeweiligen benachbarten Werte ober- und unterhalb der Grenze zu berücksichtigen sind. Es werden allerdings nur unterschiedliche Werte zur Berechnung herangezogen. Zusammenfallende Werte benachbarter Äquivalenzklassen werden als ein Wert gezählt, da ja auch nur ein Testfall mit dem entsprechenden Eingabetestwert durchgeführt wird.

#### Bewertung der Methode

*In Verbindung mit der Äquivalenzklassenbildung*

Die Grenzwertanalyse sollte in Verbindung mit der Äquivalenzklassenbildung durchgeführt werden, da an den Grenzen der Äquivalenzklassen häufiger Fehlerwirkungen nachzuweisen sind als innerhalb der Klassen. Beide Verfahren lassen sich sehr sinnvoll kombinieren, geben aber dennoch genügend Freiheitsgrade in der Wahl der konkreten Testdaten.

Das Verfahren erfordert eine hohe Kreativität, um die entsprechenden Testdaten an den Grenzen festzulegen. Dieser Aspekt wird oft nicht entsprechend beachtet, da das Verfahren als sehr einfach erscheint, obwohl die Bestimmung der relevanten Grenzen nicht trivial ist.

### 5.1.3 Zustandsbezogener Test

*Historie berücksichtigen*

Bei einer ganzen Reihe von Systemen hat neben den Werten der Eingaben auch der bisherige Ablauf des Systems Einfluss auf die Berechnung der Ausgaben bzw. auf das Systemverhalten. Die Historie ist zu berücksichtigen, die das System durchlaufen hat. Zur Veranschaulichung der Historie werden →Zustandsmodelle verwendet, die Grundlage für den Test sind (→zustandsbezogener Test).

Das System oder Testobjekt kann beginnend von einem Startzustand unterschiedliche Zustände annehmen. Zustandsänderungen oder -übergänge werden durch Ereignisse, zum Beispiel Funktionsaufrufe, ausgelöst. Bei den Zustandsänderungen können Aktionen durchzuführen sein. Neben dem Startzustand gibt es als weiteren speziellen Zustand den Endzustand. Modelliert wird das Verhalten in →Zustandsautomaten oder →endlichen Automaten.

*Definition Zustandsautomat*

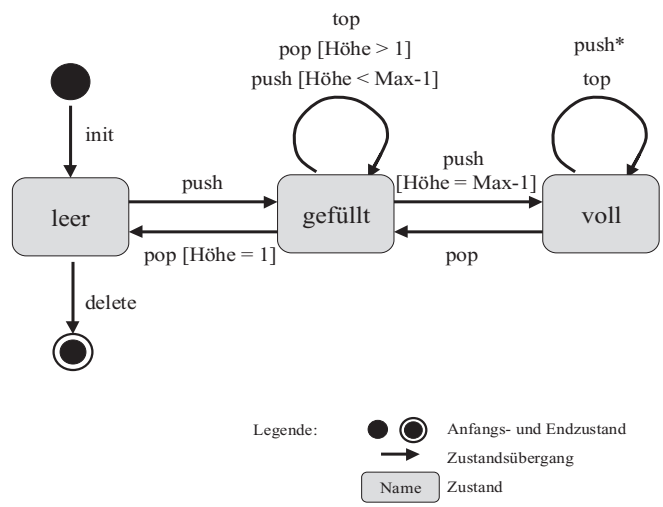
[Balzert 00] gibt folgende Definition: »Ein endlicher Automat oder ein Zustandsautomat (*finite automaton, finite state machine, sequential machine*) besteht aus einer endlichen Anzahl von internen Konfigurationen – Zustände genannt. Der Zustand eines Systems beinhaltet implizit die Informationen, die sich aus den bisherigen Eingaben ergeben haben und die benötigt werden, um die Reaktion des Systems auf noch folgende Eingaben zu bestimmen.«

---

***Beispiel: Stapel***

Das verbreitete Beispiel eines Stapels (s. Abb. 5-3) soll den Sachverhalt verdeutlichen. Der Stapel, zum Beispiel ein Tellerstapel in einem Wärmegerät, kann drei unterschiedliche Zustände annehmen: ein leerer Stapel, ein gefüllter Stapel und ein voller Stapel.

***Abb. 5–3***
*Zustandsdiagramm eines Stapels*




Der Stapel ist nach der Initialisierung (init), bei der die maximale Höhe (Max) festgelegt wird, leer (aktuelle Höhe = 0). Durch Hinzufügen eines Elements (Aufruf von push) ändert sich der Zustand des Stapels in gefüllt, und die aktuelle Höhe wird inkrementiert. In diesem Zustand können weitere Elemente hinzugefügt (push, Höhe inkrementieren) oder weggenommen (pop, Höhe dekrementieren) werden. Auch kann das oberste Element angezeigt (top, Höhe unverändert) werden. Das Anzeigen bewirkt allerdings keine Änderung des Stapels, es wird also kein Element entfernt. Ist die aktuelle Höhe einen Wert vom Maximum entfernt (Höhe = Max-1) und wird ein Element hinzugefügt (push), ändert sich der Zustand des Stapels von gefüllt in voll. Es können keine weiteren Elemente mehr hinzugefügt werden. Wird in diesem Zustand ein Element entfernt (pop), ändert sich der Zustand von voll zurück zu gefüllt. Eine Zustandsänderung von gefüllt nach leer tritt ein, wenn der Stapel aus nur einem Element besteht und dieses entfernt (pop) wird. Nur im leeren Zustand kann der Stapel gelöscht werden.

Je nach Spezifikation kann festgelegt sein, welche Funktionen (pop, push, top …) bei welchen Zuständen des Stapels aufgerufen werden können. Zu klären ist, was passiert, wenn ein Element zu einem vollen Stapel hinzugefügt werden soll (push*). In diesem Fall muss die Funktion anders reagieren als bei einem gefüllten Stapel. Die Funktionen müssen sich also in Abhängigkeit vom Zustand des Stapels unterschiedlich verhalten. Der Zustand des Testobjekts spielt somit eine entscheidende Rolle und muss beim Test entsprechend berücksichtigt werden.

*Ein möglicher konkreter Testfall*

Ein möglicher Testfall mit Vor- und Nachbedingung für einen Stapel, der Zeichenketten (Typ: string) aufnehmen kann, ist folgender:

- Vorbedingung: Stapel ist initialisiert; Zustand: leer
- Eingabe: push ("hello")
- Sollreaktion: Stapel enthält »hello«
- Nachbedingung: Zustand des Stapels: gefüllt

Weitere Funktionen des Stapels (Anzeigen der aktuellen Höhe, Anzeigen der maximalen Höhe, Abfrage, ob der Stapel leer ist …) werden in diesem Beispiel nicht berücksichtigt, da sie keine Änderung des Zustands bewirken.

*Testobjekt beim zustandsbezogenen Test*

Das Testobjekt kann beim zustandsbezogenen Test ein komplettes System mit unterschiedlichen Systemzuständen aber auch eine Klasse mit verschiedenen Zuständen in einem objektorientierten System sein. Immer wenn die Historie zu unterschiedlichem Verhalten führt, ist ein zustandsbezogener Test durchzuführen.

---

***Weitere Testfälle für das Stapelbeispiel***

Beim zustandsbezogenen Test können unterschiedliche Abstufungen der Testintensität festgelegt werden. Eine minimale Forderung ist die Erreichung aller möglichen Zustände. In dem gegebenen Stapelbeispiel sind dies die Zustände leer, gefüllt und voll[30]. Bei einer angenommenen maximalen Höhe von 4 sind alle drei Zustände nach dem Aufruf der folgenden Funktionen erreicht:

Testfall 1[31]: init [leer], push [gefüllt], push, push, push [voll]

Es sind allerdings nicht einmal alle Funktionen des Stapels im Test aufgerufen worden.

Eine andere Forderung für den Test ist der Aufruf aller Funktionen. Bei gleichen Randbedingungen ist im Beispiel folgende Reihenfolge von Funktionsaufrufen ausreichend zur Erfüllung der Forderung:

Testfall 2: init [leer], push [gefüllt], top, pop [leer], delete

30. Um den Testaufwand gering zu halten, ist der Maximalwert des Stapels nicht allzu hoch anzusetzen, da ja eine entsprechende Anzahl von push-Funktionen aufzurufen ist, um den Zustand voll zu erreichen.
31. Die folgenden Testfälle sind vereinfacht dargestellt, um sie übersichtlich zu halten.

Bei dieser Reihenfolge sind allerdings nicht einmal alle Zustände des Stapels im Test erreicht worden.

*Testkriterien*

Ein zustandsbezogener Test sollte bei jedem Zustand alle für diesen Zustand spezifizierten Funktionen mindestens einmal zur Ausführung bringen. Die Übereinstimmung zwischen dem im Zustandsautomaten spezifizierten gewünschten Verhalten und dem tatsächlichen Verhalten des Testobjekts kann dann festgestellt werden.

*Übergangsbaum erstellen*

Zur Ermittlung der notwendigen Testfälle wird aus dem Zustandsdiagramm ein so genannter Übergangsbaum erstellt, der bestimmte Folgen von Zustandswechseln enthält (s. a. [Sneed 02]). Aus dem zyklischen Zustandsdiagramm mit potenziell unendlich vielen Folgen von Zuständen wird im Übergangsbaum eine repräsentative Menge von Zuständen ohne Zyklen. Dabei sollen mindestens alle Zustände erreicht und alle Übergänge des Zustandsdiagramms vorkommen.

Nach Sneed und Winter wird der Übergangsbaum aus einem Zustandsdiagramm wie folgt gebildet:

1. Der Anfangszustand wird die Wurzel des Baums.
2. Für jeden möglichen Übergang vom Anfangszustand zu einem Folgezustand im Zustandsdiagramm erhält der Übergangsbaum von der Wurzel aus eine Verzweigung zu einem Knoten, der den Nachfolgezustand repräsentiert.
3. Der letzte Schritt wird für jedes Blatt des Übergangsbaums so lange wiederholt, bis eine der beiden Endbedingungen eintritt:
   - Der dem Blatt entsprechende Zustand ist auf dem Weg von der Wurzel zum Blatt bereits einmal im Baum enthalten. Diese Endbedingung entspricht einem Durchlauf von einem Zyklus im Zustandsdiagramm.
   - Der dem Blatt entsprechende Zustand ist ein Endzustand und hat somit keine weiteren Übergänge, die zu berücksichtigen wären.

Für den Stapel ergibt sich der in Abb. 5-4 dargestellte Übergangsbaum.

Es ergeben sich acht unterschiedliche Pfade von der Wurzel zu jeweils einem Blatt. Jeder dieser Pfade präsentiert einen Testfall, also eine Abfolge von Funktionsaufrufen. Dabei wird jeder Zustand mindestens einmal erreicht und jede Funktion wird gemäß der Spezifikation durch das Zustandsdiagramm in den jeweiligen Zuständen aufgerufen.

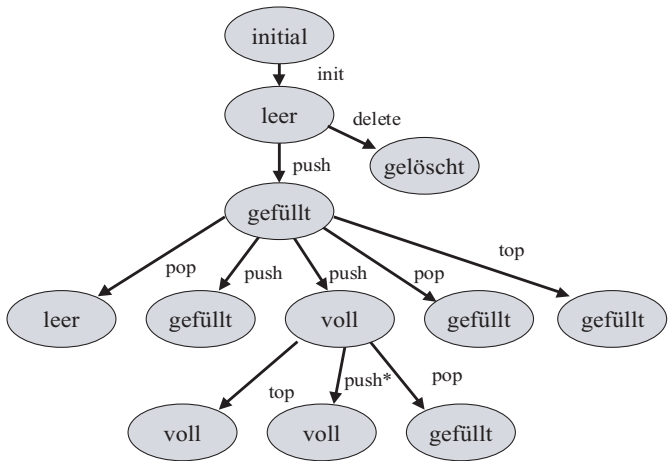



***Abb. 5–4***
*Übergangsbaum des Stapel-Beispiels*

*Fehlerhafte Verwendung der Funktionen*

Darüber hinaus ist die Reaktion auf spezifikationsverletzende Benutzungen zu prüfen. Eine Verletzung der Spezifikation ist gegeben, wenn Funktionen in Zuständen aufgerufen werden, in denen sie nicht verwendet werden sollten (z. B. das Löschen des Stapels im gefüllten Zustand). Es ist ein Robustheitstest, der prüft, wie sich das Testobjekt bei fehlerhafter Verwendung verhält. Getestet wird, ob unerwartete Übergänge auftreten können. Der Test ist in Analogie zum Test von unerwarteten Eingabewerten zu sehen.

Der Übergangsbaum ist dahingehend zu erweitern, dass von jedem Zustand aus alle Funktionen ausgeführt werden oder zumindest der Versuch unternommen wird (s. Abb. 5-5).

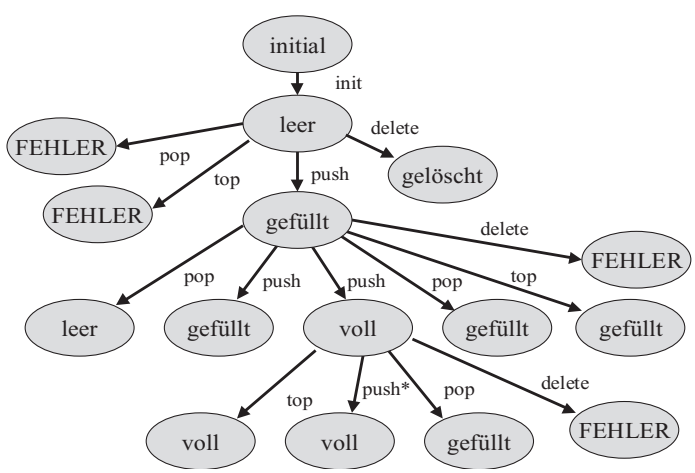



***Abb. 5–5***
*Erweiterter Übergangsbaum für den Robustheitstest*

Zustandbezogenes Testen eignet sich auch gut als Technik beim Systemtest, wenn Tests der grafischen Bedienoberfläche des Testobjekts erfolgen. Die Bedienoberfläche besteht in der Regel aus einer Menge von Masken oder Dialogboxen, zwischen denen durch Eingabeaktionen (Menüauswahl, »OK«-Button usw.) hin und her gewechselt werden kann. Werden Masken bzw. Dialogboxen als Zustände aufgefasst und Eingabeaktionen als Zustandsübergänge, so können die Navigationsmöglichkeiten durch die Bedienoberfläche als Zustandsautomat modelliert werden. Geeignete Testfälle und die Testüberdeckung können mit oben beschriebener Methode des zustandsbasierten Tests ermittelt werden.

***Beispiel: Test der DreamCar-GUI***

Beim Test der *DreamCar*-GUI kann dies beispielsweise wie folgt aussehen:

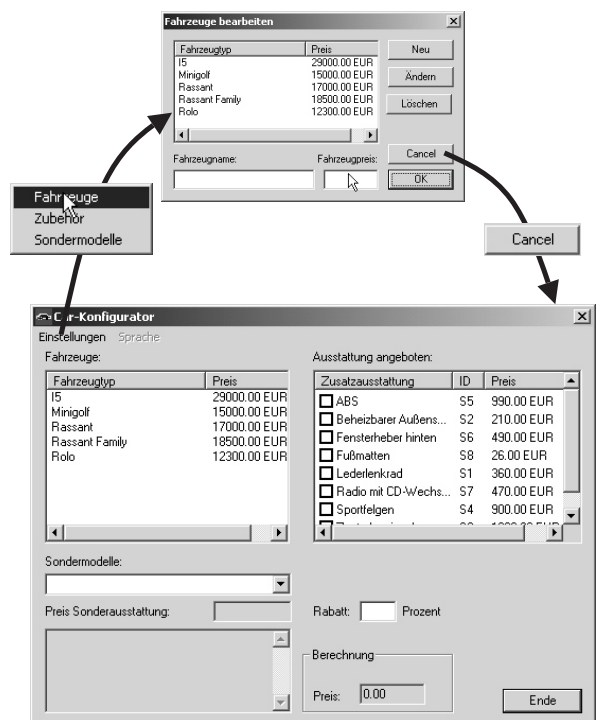



***Abb. 5–6***
*GUI-Navigation als Zustandsgraph*

Der Test beginnt in der *DreamCar*-Hauptmaske (Zustand 1). Über die Aktion[32] »Einstellungen/Fahrzeuge« erfolgt der Wechsel in den Dialog »Fahrzeuge bearbeiten« (Zustand 2). Die Aktion »Cancel« beendet diesen Dialog, und es erfolgt der Rücksprung in Zustand 1. Innerhalb eines Zustands kön-

32. Die zweistufige Menüauswahl wird hier als eine einzige Aktion aufgefasst.

nen dann »zustandslokale« Tests erfolgen, die die eigentliche Funktionalität der angesteuerten Maske prüfen. Auf analoge Weise kann die Navigation durch beliebige komplexe Dialogketten dargestellt werden. Das Zustandsmodell der Bedienoberfläche hilft sicherzustellen, dass alle Dialoge im Test beachtet und geprüft werden.

---

#### Testfälle

Zur vollständigen Definition eines zustandsbezogenen Testfalls gehören folgende Informationen:

- der Anfangszustand des Testobjekts (Komponente oder System),
- die Eingaben für das Testobjekt,
- die erwarteten Ausgaben bzw. das erwartete Verhalten,
- der erwartete Endzustand.

Ferner sind für jeden im Testfall erwarteten Zustandsübergang folgende Aspekte festzulegen:

- den Zustand vor dem Übergang,
- das auslösende Ereignis, das den Übergang bewirkt,
- die erwartete Reaktion, ausgelöst durch den Übergang,
- den nächsten erwarteten Zustand.

Es ist nicht immer einfach, die Zustände eines Testobjekts zu ermitteln. Oft manifestiert sich der Zustand nicht in einer einzigen Variablen, sondern der Zustand ergibt sich aus der entsprechenden Konstellation der unterschiedlichen Werte der Variablen. Die Überprüfung und Bewertung der einzelnen Testfälle kann dadurch sehr aufwändig werden.

---

***Tipp***

- Das Zustandsdiagramm bereits bei der Spezifikation unter Testgesichtspunkten bewerten. Bei einer hohen Anzahl von Zuständen und Übergängen auf den erhöhten Testaufwand hinweisen und auf eine Vereinfachung dringen, soweit dies möglich ist.
- Ebenfalls bei der Spezifikation darauf achten, dass die unterschiedlichen Zustände leicht zu ermitteln sind und sich nicht aus einer vielfältigen Kombination von Werten von unterschiedlichen Variablen ergeben.

---

#### Festlegung der Testendekriterien

Kriterien zur Intensität und zur Beendigung lassen sich auch beim zustandsbezogenen Test angeben:

- Jeder Zustand wurde mindestens einmal erreicht.
- Jeder Zustandsübergang wurde mindestens einmal ausgeführt
- Alle spezifikationsverletzenden Zustandsübergänge wurden geprüft.

Entsprechende Prozentzahlen lassen sich durch das Verhältnis der jeweils möglichen zu den tatsächlich durchgeführten Anforderungen in Analogie zu den anderen Überdeckungsmaßen angeben.

*Weitgehende Prüfungen*

Bei hoch kritischen Anwendungen lassen sich noch weitgehendere zustandsbezogene Prüfungen vornehmen:

- Alle Kombination von Zustandsübergängen,
- alle Zustandsübergänge in jeder beliebigen Reihenfolge mit allen möglichen Zuständen, auch mehrfach hintereinander.

Allerdings ist das Erreichen einer ausreichenden Überdeckung durch die große Zahl an benötigten Testfällen meist nicht möglich. Sinnvoll ist dann eine Einschränkung der Anzahl von zu prüfenden Kombinationen bzw. Reihenfolgen.

#### Bewertung der Methode

Der zustandsbezogene Test ist überall dort anzuwenden, wo Zustände eine Rolle spielen und wo die Funktionalität durch den jeweiligen Zustand des Testobjekts unterschiedlich beeinflusst wird. Die anderen bisher vorgestellten Testverfahren berücksichtigen diese Aspekte nicht, da sie nicht auf das unterschiedliche Verhalten von Funktionen in Abhängigkeit vom Zustand eingehen.

*Besonders geeignet zum Test objektorientierter Systeme*

In objektorientierten Systemen können Objekte unterschiedliche Zustände annehmen. Die jeweiligen Methoden zur Manipulation der Objekte müssen dann entsprechend auf die unterschiedlichen Zustände reagieren. Beim objektorientierten Testen hat der zustandsbezogene Test deshalb eine herausgehobene Bedeutung, da er diesen speziellen Aspekt der Objektorientierung berücksichtigt.

### 5.1.4 Allgemeine Bewertung der Black-box-Verfahren

Es werden hier bei weitem nicht alle Black-box-Verfahren beschrieben, die zur Prüfung eines Testobjekts verwendet werden können. Die gängigsten Verfahren wurden ausgewählt. Weitere Verfahren finden sich in [Myers 82], [Pol 00], [Riedemann 97] und [Sneed 02]. Im Folgenden werden weitere Verfahren kurz beschrieben, um bei deren Auswahl eine erste Hilfestellung zu geben.

### Weitere Black-box-Verfahren

Bei den bisher vorgestellten Verfahren werden die verschiedenen Eingabedaten relativ unabhängig voneinander betrachtet, und die Werte der Eingaben werden jeweils separat für einen Testfall herangezogen. Abhängigkeiten zwischen den Eingaben untereinander, aber auch zwischen den Ein- und Ausgaben werden nicht explizit berücksichtigt.

*Ursache-Wirkungs-Graph–Analyse*

[Myers 82] beschreibt ein Verfahren, das die Abhängigkeiten bei der Ermittlung der Testfälle berücksichtigt: die →Ursache-Wirkungs-Graph-Analyse (s.a. [Sneed 02]). Die logischen Beziehungen zwischen Ursachen und deren Wirkungen in einer Komponente oder einem System werden im Ursache-Wirkungs-Graph dargestellt. Jede Ursache wird als eine Bedingung beschrieben, die aus Eingabedaten (oder Kombinationen von Eingabedaten) besteht. Die Eingaben werden über logische Operatoren (z. B. AND oder OR ) verknüpft. Die Bedingung, und damit die Ursache, kann `wahr` oder `falsch` sein. Die Wirkungen werden analog behandelt und im Graphen vermerkt. Der Graph ist in eine →Entscheidungstabelle umzuformen, aus der dann die Testfälle abzulesen sind.

*Syntaxtest*

Mit →Syntaxtest wird ein Verfahren zur Ermittlung der Testfälle bezeichnet, das bei Vorliegen einer formalen Spezifikation der Syntax der Eingaben angewendet werden kann. Die Regeln der syntaktischen Beschreibung werden genutzt, um Testfälle zu spezifizieren, die sowohl die Einhaltung als auch die Verletzung der syntaktischen Regeln für die Eingaben berücksichtigen.

*Zufallstest*

Der →Zufallstest wählt aus der Menge der möglichen Werte eines Eingabedatums zufällig Repräsentanten für die Testfälle aus. Ist eine statistische Verteilung der Werte gegeben (z. B. eine Normalverteilung), so soll diese auch für die Wahl der Repräsentanten herangezogen werden, um möglichst realitätsnahe Testfälle und Aussagen zur Zuverlässigkeit des Systems zu erhalten.

*Smoke-Test*

Unter →Smoke-Test wird mehr ein »Ausprobieren« des Testobjekts verstanden, das vorwiegend die Robustheit prüft. Es wird ein in der Regel automatisierter Test durchgeführt, der möglichst alle Hauptfunktionen des Testobjekts auslöst. Die Testobjekt-Ausgaben werden nicht ausgewertet. Es wird also kein Testorakel zur Ermittlung der Sollergebnisse benötigt. Stattdessen wird versucht, einen offensichtlichen Absturz des Testobjekts zu erzeugen. Beim Ausfall steigt dann Rauch (engl. s*moke*) aus dem Testobjekt auf (in Analogie zum Ausfall bei älteren elektrischen Geräten).

### Vor- und Nachteile der Black-box-Verfahren

*Fehlerhafte Spezifikation wird nicht erkannt*

Grundlage aller Black-box-Verfahren sind die Anforderungen sowie die Spezifikation des Systems bzw. der einzelnen Komponenten und ihres Zusammenwirkens. Sind fehlerhafte Festlegungen in den Anforderungen getroffen oder war eine fehlerhafte Spezifikation Grundlage für die Implementierung, so können diese Fehler nicht erkannt werden, da es keine Abweichungen zwischen der fehlerhaften Anforderung und dem tatsächlichen Verhalten gibt. Das Testobjekt verhält sich so, wie die Spezifikation (bzw. die Anforderungen) es fordert, auch wenn diese fehlerhaft ist. Zur Auffindung von Unstimmigkeiten und Fehlern in der Spezifikation sind Reviews (s. Kap. 4.1.2) durchzuführen.

*Nicht geforderte Funktionalität wird nicht erkannt*

Mit den Black-box-Verfahren kann auch nicht festgestellt werden, ob das Testobjekt nicht weitere Funktionalität bereitstellt, die über die Spezifikation hinausgeht. Diese zusätzlichen Funktionen sind weder spezifiziert noch vom Kunden gefordert. Testfälle, die diese zusätzlichen Funktionen zur Ausführung bringen, werden – wenn überhaupt – nur zufällig durchgeführt. Die Überdeckungskriterien, die zur Beendigung des Tests dienen, werden ausschließlich auf der Grundlage der Anforderungen bzw. der Spezifikation ermittelt und nicht auf der Grundlage von nicht beschriebenen und nur vermuteten Funktionen.

*Prüfung der Funktionalität*

Im Mittelpunkt aller Black-box-Verfahren steht die Prüfung der Funktionalität des Testobjekts. Es ist sicherlich unumstritten, dass das korrekte Funktionieren eines Softwaresystems höchste Priorität hat und somit auch die Black-box-Testverfahren stets einzusetzen sind.

## 5.2 White-box-Verfahren

*Codebasierte Testverfahren*

Grundlage der White-box-Verfahren ist der Programmtext des Testobjekts. Die Verfahren werden daher oft auch als →codebasierte Testverfahren bezeichnet. Der Programmtext muss vorliegen und unter Umständen auch manipuliert, d. h. ergänzt werden können.

*Alle Codeteile sollen zur Ausführung kommen*

Die grundlegende Idee der White-box-Verfahren ist, dass alle Codeteile eines Testobjekts mindestens einmal zur Ausführung gebracht werden sollen. Aufgrund der Programmlogik werden ablauforientierte Testfälle ermittelt und ausgeführt. Dabei ist selbstverständlich auch die Spezifikation zu berücksichtigen, um den Testfall zu erstellen und insbesondere nach seiner Ausführung bewerten und entscheiden zu können, ob ein fehlerhaftes Verhalten vorliegt.

Betrachtungsgegenstand eines White-box-Verfahrens können beispielsweise die Anweisungen eines Testobjekts sein. Ziel der Methode ist dann das Erreichen einer zuvor festgelegten Überdeckung der An-

weisungen beim Testen, d. h. möglichst viele der im Programm enthaltenen Anweisungen zur Ausführung zu bringen.

Es lassen sich folgende White-box-Testfallentwurfsmethoden unterscheiden:

- →Anweisungsüberdeckung
- →Zweigüberdeckung
- Test der Bedingungen
  - →Einfache Bedingungsüberdeckung
  - →Mehrfachbedingungsüberdeckung
  - →Minimale Mehrfachbedingungsüberdeckung
  - →Pfadüberdeckung

Im Folgenden werden die Verfahren näher beschrieben.

### 5.2.1 Anweisungsüberdeckung

Die einzelnen Anweisungen (engl. *statements*) des Testobjekts stehen im Mittelpunkt der Untersuchung. Es sind Testfälle zu identifizieren, die eine zuvor festgelegte Mindestquote oder auch alle Anweisungen des Testobjektes zur Ausführung bringen. Als ersten Schritt ist der Programmtext in einen Kontrollflussgraphen zu transferieren. Anhand des Graphen werden dann die geforderten Überdeckungen genauer spezifiziert. Im Graphen werden die Anweisungen als Knoten und der Kontrollfluss zwischen den Anweisungen als Kanten repräsentiert. Sind Sequenzen von Anweisungen im Programmstück vorhanden, werden diese als ein Knoten dargestellt, da bei Ausführung der ersten Anweisung der Sequenz alle weiteren Anweisungen ausgeführt werden.

Es ist nach Ausführung der Testfälle nachzuweisen, welche einzelnen Anweisungen ausgeführt wurden (s. Kap. 5.2.6). Wenn der zuvor festgelegte Überdeckungsgrad erreicht ist, wird der Test als ausreichend angesehen und beendet.

---

***Beispiel***

Ein Beispiel soll das Vorgehen verdeutlichen. Für das Beispiel wird ein sehr einfaches Programmstück gewählt, das lediglich aus zwei Abfragen und einer Schleife besteht (s. Abb. 5-7).

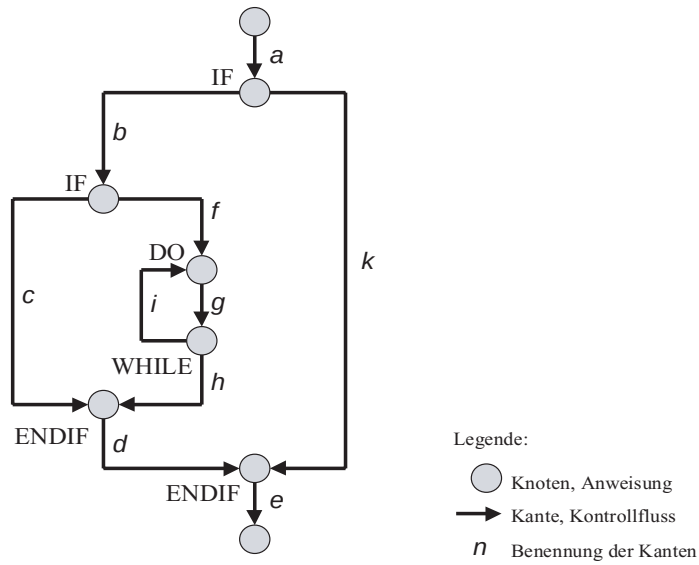



***Abb. 5–7***
*Kontrollflussgraph eines Programmstücks*

### Testfälle

*Überdeckung der Kanten des Kontrollflussgraphen*

Im Beispiel können alle Anweisungen (alle Knoten) bereits durch einen einzigen Testfall erreicht werden. Bei diesem Testfall müssen die folgenden Kanten des Graphen durchlaufen werden, d. h. die Anweisungen (Knoten) in der entsprechenden Reihenfolge zur Ausführung kommen:

a, b, f, g, h, d, e

*Ein Testfall reicht aus*

Nach dem Durchlaufen der Kanten sind alle Anweisungen einmal ausgeführt worden. Die vollständige Überdeckung lässt sich sicherlich auch durch andere Kombinationen von Kanten des Graphen erreichen. Beim Testen ist jedoch immer der Aufwand zu minimieren, d. h. mit möglichst wenigen Testfällen das zu erfüllende Ziel zu erreichen.

Für die einzelnen Testfälle sind auch die erwarteten Ergebnisse und das erwartete Verhalten des Testobjekts vorab anhand der Spezifikation durch ein Testorakel zu bestimmen. Nach der Ausführung ist das erwartete und das tatsächliche Ergebnis bzw. Verhalten zu vergleichen, um Abweichungen und Fehlerwirkungen festzustellen.

**Festlegung der Testendekriterien**

Kriterien zur Beendigung der Tests lassen sich hier sehr einsichtig definieren:

Anweisungsüberdeckung =
(Anzahl durchlaufene Anweisungen / Gesamtzahl Anweisungen) * 100 %

*C0-Maß*

Das obige Maß zur Berechnung der Überdeckung der Anweisungen ist auch als C0-Maß bekannt. Es ist ein in der Aussage sehr schwaches Kriterium. Manchmal ist eine 100%ige Überdeckung der Anweisungen schwer erreichbar, z. B. wenn Ausnahmebedingungen im Programm vorkommen, die während der Testphase nur mit erheblichem Aufwand oder gar nicht herzustellen sind.

**Bewertung der Methode**

*Nicht erreichbarer Code erkennbar*

Wird eine vollständige Überdeckung aller Anweisungen gefordert und können Anweisungen durch keine Testfälle zur Ausführung gebracht werden, kann das ein Hinweis auf nicht erreichbaren Programmtext (tote Programmanweisung) sein.

*Leere ELSE-Teile bleiben unberücksichtigt*

Ist bei einer Bedingung (IF) nur bei deren Erfüllung (THEN-Teil) die Ausführung von Anweisungen vorgesehen, so ist im entsprechenden Kontrollflussgraphen eine von der Bedingung ausgehende (THEN-) Kante mit (mindestens) einem Knoten und eine zweite ausgehende (ELSE-)Kante ohne Knoten vorhanden. Der Kontrollfluss beider Kanten wird beim abschließenden (ENDIF-)Knoten wieder vereint. Für die Anweisungsüberdeckung ist eine leere (ELSE-)Kante (zwischen IF und ENDIF) ohne Bedeutung. Möglicherweise fehlende Anweisungen in diesem Programmteil werden nicht erkannt!

Die Messung der Überdeckung der Anweisungen wird mittels entsprechender Werkzeuge durchgeführt (s. Kap. 7.1.4)

### 5.2.2 Zweigüberdeckung

Ein weitergehendes Kriterium im White-box-Test ist die Überdeckung der Zweige des Kontrollflussgraphen, d. h., die Kanten im Graphen stehen im Mittelpunkt der Untersuchung. Nicht die Ausführung der einzelnen Anweisungen wird betrachtet, sondern die Auswertung einer Bedingung. Aufgrund des Ergebnisses der Auswertung wird entschieden, welche Anweisung als nächste ausgeführt wird. Dies soll beim Testen berücksichtigt werden.

*Leere ELSE-Teile werden berücksichtigt*

Dabei spielt es im Gegensatz zur Knotenüberdeckung für die Zweigüberdeckung keine Rolle, ob beispielsweise bei einer IF-Anweisung sowohl im THEN- als auch im ELSE-Teil Anweisungen auszufüh-

ren sind. Ein leerer ELSE-Teil ist bei der Zweigüberdeckung ebenfalls zu berücksichtigen. Die Zweigüberdeckung verlangt, dass bei einer Verzweigung des Kontrollflusses beide (bei einer Bedingung) bzw. alle (bei einer CASE-Anweisung) Möglichkeiten und bei Schleifen die Umgehung des Schleifenkörpers und ein Rücksprung zum Schleifenanfang zu berücksichtigen sind.

#### Testfälle

*Zusätzliche Testfälle erforderlich*

Im Beispiel (s. Abb. 5-7) sind nun weitere Testfälle erforderlich, wenn alle Zweige des Kontrollflussgraphen beim Test berücksichtigt werden sollen. Für eine 100%ige Anweisungsüberdeckung war folgende Kantenabfolge ausreichend:

a, b, f, g, h, d, e

Die Kanten c, i und k sind durch diesen Testfall nicht ausgeführt worden. Die Kante c und k sind leere Zweige einer Bedingung, die Kante i ist der Rücksprung zum Anfang einer Schleife. Es sind zusätzlich drei Testfälle erforderlich:

a, b, c, d, e
a, b, f, g, i, g, h, d, e
a, k, e

*Überdeckung der Kanten des Kontrollflussgraphen*

Alle vier Testfälle zusammen ergeben eine vollständige Überdeckung der Kanten des Kontrollflussgraphen, womit alle möglichen Verzweigungen des Kontrollflusses im Programmtext des Testobjektes durch einen Testfall überprüft worden sind.

Es sind jetzt Kanten mehrfach ausgeführt worden, dies lässt sich allerdings nicht immer vermeiden. Im Beispiel werden die Kanten a und e bei jedem Testfall ausgeführt, da es keine Alternativen zu diesen Kanten gibt.

Für die einzelnen Testfälle sind, neben den Vor- und Nachbedingungen, auch hier wieder die erwarteten Ergebnisse und das erwartete Verhalten des Testobjekts vorab zu bestimmen und danach mit dem tatsächlichen Ergebnis bzw. Verhalten zu vergleichen. Darüber hinaus ist es sinnvoll festzuhalten, welche Zweige bei welchem Testfall durchlaufen werden sollen, um Abweichungen im Ablauf feststellen zu können.

#### Festlegung der Testendekriterien

In Analogie zur Anweisungsüberdeckung wird der Überdeckungsgrad der Zweigüberdeckung wie folgt definiert:

Zweigüberdeckung =
(Anzahl durchlaufene Zweige / Gesamtzahl Zweige) * 100 %

*C1-Maß*

Mit C1-Maß ist die Überdeckung der Zweige benannt. Bei der Berechnung wird nur gezählt, ob ein Zweig bei der Ausführung überhaupt durchlaufen wurde, die Häufigkeit der Ausführung spielt keine Rolle, so sind im Beispiel die Zweige a und e jeweils viermal, nämlich bei jedem Testlauf einmal, durchlaufen worden.

Unter der Annahme, dass im obigen Beispiel der letzte Testfall mit der Kante k nicht ausgeführt wurde, ergibt sich eine Zweigüberdeckung von

$$9/10 = 90\ \%$$

Zum Vergleich: Eine 100%ige Anweisungsüberdeckung ist bereits nach dem ersten Testfall erreicht.

Je nach Kritikalität des Testobjekts und je nach erwartetem Risiko im Fehlerfall kann das Testendekriterium unterschiedlich festgelegt werden. Es können beispielsweise 85 % Zweigüberdeckung einer Komponente in einem Projekt als ausreichend angesehen werden, während für eine andere Komponente eine Überdeckung von 100 % erreicht werden muss. Wie im Beispiel zu sehen, ist der Testaufwand bei hoher geforderter Überdeckung entsprechend umfangreich.

#### Bewertung der Methode

*Mehr Testfälle erforderlich*

Bei der Zweigüberdeckung ist die Ausführung von mehr Testfällen als bei der Anweisungsüberdeckung gefordert. Wie viel mehr, hängt von der Struktur des Testobjekts ab. Mit der Zweigüberdeckung können fehlende Anweisung in leeren Zweigen im Gegensatz zur Anweisungsüberdeckung erkannt werden.

Die einzelnen Zweige werden unabhängig voneinander betrachtet, und es werden keine bestimmten Kombinationen der einzelnen Zweige gefordert. Die Abfolge der Zweige erfolgt relativ willkürlich.

***Tipp***

- Eine Zweigüberdeckung von 100 % ist anzustreben.
- Nur wenn neben allen Anweisungen auch jede mögliche Verzweigung des Kontrollflusses in der Testphase berücksichtigt wird, kann der Test als ausreichend eingestuft werden.

*Für objektorientierte Systeme unzureichend*

Sowohl die Anweisungs- als auch die Zweigüberdeckung ist für objektorientierte Systeme nur unzureichend geeignet, da die Methoden in den Klassen normalerweise wenig umfangreich und von geringer Komplexität sind. Die geforderten Überdeckungen lassen sich dann mit wenig Aufwand erreichen. Die Komplexität bei objektorientierten Systemen ist meist in den Beziehungen zwischen den Klassen verborgen. Da oft eine Werkzeugunterstützung zur Ermittlung der Überdeckungswerte

vorhanden ist, kann diese allerdings genutzt werden, um nicht aufgerufene Methoden oder Programmteile zu erkennen.

### 5.2.3 Test der Bedingungen

*Berücksichtigung der Komplexität bei zusammengesetzten Bedingungen*

Bei der Zweigüberdeckung wird ausschließlich der ermittelte Ergebniswahrheitswert einer Bedingung berücksichtigt (`wahr`/`TRUE` oder `falsch`/`FALSE`). Anhand dieses Wertes wird entschieden, welche Verzweigung im Kontrollflussgraphen verfolgt wird bzw. welche Anweisung als nächste im Programm zur Ausführung kommt. Setzt sich eine Bedingung aus mehreren Teilbedingungen zusammen, die über logische Operatoren miteinander verknüpft sind, so muss im Test diese Komplexität der Bedingung berücksichtigt werden. Unterschiedliche Anforderungen und damit auch Abstufungen der Testintensität unter Berücksichtigung der zusammengesetzten Bedingungen werden im Folgenden beschrieben.

**Einfache Bedingungsüberdeckung**
(*branch condition testing*)[33]

Ziel der einfachen Bedingungsüberdeckung ist, dass jede →atomare Teilbedingung im Test sowohl den Wert `wahr` als auch den Wert `falsch` angenommen hat.

*Definition atomare Teilbedingung*

Eine atomare Teilbedingung ist eine Bedingung, die keine logischen Operatoren wie AND, OR oder NOT, sondern höchstens Relationssymbole wie »>« oder »=« enthält.

Eine Bedingung im Programmtext des Testobjekts kann aus mehreren atomaren Teilbedingungen zusammengesetzt sein.

---

***Beispiel für zusammengesetzte Bedingungen***

Ein Beispiel für eine zusammengesetzte Bedingung ist: x > 3 OR y < 5. Die Bedingung besteht aus zwei atomaren Teilbedingungen (x>3; y<5), die mit einem logischen ODER (OR) verknüpft sind. Ziel der einfachen Bedingungsüberdeckung ist, dass jede atomare Teilbedingung einmal jeden der beiden Wahrheitswerte annimmt. Die Testdaten x=6 und y=8 ergeben für die erste Teilbedingung (x>3) den Wahrheitswert `wahr` und für die zweite Teilbedingung (y<5) den Wahrheitswert `falsch`. Der Wahrheitswert der gesamten Bedingung ist `wahr` (`wahr` ODER `falsch` = `wahr`). Das zweite Testdatenpaar mit den Werten x=2 und y=3 ergibt für die erste Teilbedingung den Wahrheitswert `falsch` und für die zweite den Wahrheitswert `wahr`. Der Wert der Gesamtbedingung ergibt sich erneut als `wahr` (`falsch` ODER

33. Die entsprechenden englischen Bezeichnungen sind zur Verdeutlichung hier mit aufgenommen worden.

`wahr = wahr`). Beide Teilbedingungen haben jeweils die beiden Wahrheitswerte angenommen. Die Auswertung der Gesamtbedingung ist allerdings bei beiden Kombinationen gleich.

*Schwaches Kriterium*

Die einfache Bedingungsüberdeckung ist somit ein schwächeres Kriterium als die Anweisungs- oder auch Zweigüberdeckung, da nicht verlangt wird, dass unterschiedliche Wahrheitswerte bei der Auswertung der Gesamtbedingung im Test zu berücksichtigen sind.

---

**Mehrfachbedingungsüberdeckung**
(*branch condition combination testing*)

*Alle Kombinationen der Wahrheitswerte*

Bei der Mehrfachbedingungsüberdeckung wird gefordert, dass Kombinationen der Wahrheitswerte der atomaren Teilbedingungen berücksichtigt werden. Es sollen möglichst alle Variationen gebildet werden.

---

***Fortführung des Beispiels***

Für das obige Beispiel sind bei den zwei atomaren Teilbedingungen (x>3, y<5) vier Kombinationen von Testfällen mit den obigen Testdaten möglich:

x=6 (T), y=3 (T), x>3 OR y<5 (T)
x=6 (T), y=8 (F), x>3 OR y<5 (T)
x=2 (F), y=3 (T), x>3 OR y<5 (T)
x=2 (F), y=8 (F), x>3 OR y<5 (F)

*Mehrfachbedingungsüberdeckung subsumiert Anweisungs- und Zweigüberdeckung*

Bei der Auswertung der Gesamtbedingung ergeben sich nun auch beide Wahrheitswerte. Die Mehrfachbedingungsüberdeckung erfüllt somit auch die Kriterien der Anweisungs- und Zweigüberdeckung. Sie ist ein umfassenderes Kriterium, da sie auch die Komplexität bei zusammengesetzten Bedingungen berücksichtigt. Allerdings ist sie auch sehr aufwändig, da bei steigender Anzahl der atomaren Bedingungen die Zahl der möglichen Kombinationen exponentiell ansteigt (auf $2^n$ bei n atomaren Bedingungen).

---

*Nicht alle Kombinationen sind immer möglich*

Ein weiteres Problem ergibt sich daraus, das nicht immer alle Kombinationen auch durch Testdaten realisierbar sind.

---

***Beispiel für nicht realisierbare Kombinationen von Teilbedingungen***

Ein Beispiel soll das verdeutlichen. Für die zusammengesetzte Bedingung 3<=x AND x<5 lassen sich nicht alle Kombinationen mit den entsprechenden Werten für die Variable x herstellen, da die Teilbedingungen nicht unabhängig voneinander sind:

x=4 - 3<=x (T), x<5 (T), 3<=x AND x<5 (T)
x=8 - 3<=x (T), x<5 (F), 3<=x AND x<5 (F)
x=1 - 3<=x (F), x<5 (T), 3<=x AND x<5 (F)
x=? - 3<=x (F), x<5 (F), Kombination nicht möglich, weil der Wert für x kleiner als 3 und zugleich größer oder gleich 5 sein müsste)

---

### Minimale Mehrfachbedingungsüberdeckung

(*modified branch condition decision testing*)

*Beschränkung der Kombinationen*

Die minimale Mehrfachbedingungsüberdeckung beseitigt die oben geschilderten Probleme. Es müssen nicht alle Kombinationen berücksichtigt werden, sondern nur jede mögliche Kombination von Wahrheitswerten, bei denen die Änderung des Wahrheitswertes einer atomaren Bedingung den Wahrheitswert der logischen Verknüpfung ändern kann (s. a. [Riedemann 97]).

---

***Fortführung des Beispiels***

Zur Verdeutlichung wird das Beispiel mit den zwei atomaren Teilbedingungen (x>3, y<5) erneut betrachtet. Vier Kombinationen sind möglich ($2^2$):

1) x=6 (T), y=3 (T), x>3 OR y<5 (T)
2) x=6 (T), y=8 (F), x>3 OR y<5 (T)
3) x=2 (F), y=3 (T), x>3 OR y<5 (T)
4) x=2 (F), y=8 (F), x>3 OR y<5 (F)

*Änderung einer Teilbedingung ohne Auswirkung*

Für die erste Kombination gilt Folgendes: Wird der Wahrheitswert für die erste Teilbedingung fehlerhaft berechnet, ist also eine falsche Bedingung realisiert, so kann sich der Wahrheitswert der ersten Teilbedingung von `wahr` (T) durch den Fehlerzustand auf `falsch` (F) ändern, allerdings bleibt das Ergebnis der Gesamtbedingung unverändert (T). Analoges gilt für die zweite Teilbedingung.

Bei der ersten Kombination werden somit fehlerhafte Auswertungen der jeweiligen Teilbedingung maskiert, da sie keine Auswirkungen auf das Ergebnis der Gesamtbedingung haben und Fehlerwirkungen somit nicht auftreten. Der Test der ersten Kombination kann somit entfallen.

Wird beim zweiten Testfall der Wahrheitswert der ersten Teilbedingung durch einen Fehlerzustand `falsch` berechnet, ändert sich der Wert von `wahr` (T) zu `falsch` (F). Hier kommt es zu einer Fehlerwirkung, da sich auch der Wert der Gesamtbedingung ändert. Analoges gilt für den dritten Testfall und dort die zweite Teilbedingung. Im vierten Testfall wird eine falsche Realisierung der Teilbedingungen ebenfalls erkannt, da sich dann auch der Wahrheitswert der Gesamtbedingung ändert.

---

*Geringe Anzahl der Testfälle*

Für jede logische Verknüpfung der zusammengesetzten Bedingung ist zu entscheiden, welche Testfälle sensitiv auf Fehlerzustände reagieren und bei welchen Kombinationen Fehlerzustände maskiert werden können und die Kombinationen somit im Test nicht berücksichtigt werden brauchen. Die Anzahl der Testfälle ist erheblich geringer als bei der Mehrfachbedingungsüberdeckung, die Anzahl liegt zwischen n+1 und 2n, mit n=Anzahl der booleschen Operanden der Bedingung.

### Testfälle

Bei den Testfällen ist jeweils zu berücksichtigen, welche Eingabedaten zu welcher Auswertung der Bedingungen bzw. Teilbedingungen führen und welche Programmteile nach der Auswertung zur Ausführung kommen sollen. Die erwartete Ausgabe bzw. das erwartete Verhalten des Testobjekts ist ebenfalls vorab festzulegen, um das korrekte bzw. fehlerhafte Verhalten festzustellen.

***Tipp***

- Wegen der geringen Aussagekraft soll auf die einfache Bedingungsüberdeckung verzichtet werden.
- Bei zusammengesetzten Bedingungen die minimale Mehrfachbedingungsüberdeckung anwenden, da die Komplexität der Bedingung von dem Verfahren zur Testfallerstellung berücksichtigt wird und das Verfahren die Anweisungs- und Zweigüberdeckung subsumiert. Diese beiden Verfahren sind dann nicht zusätzlich durchzuführen.

Bei der minimalen Mehrfachbedingungsüberdeckung kann es allerdings sehr aufwändig sein, die Eingabedaten so zu wählen, dass eine bestimmte Teilbedingung den im Testfall geforderten Wahrheitswert annimmt.

### Festlegung der Testendekriterien

Wie bei den bisherigen Verfahren auch, lassen sich als Kriterien für die Beendigung der Tests die Verhältnisse zwischen den bereits erreichten und allen geforderten Wahrheitswerten der (Teil-)Bedingungen bilden. Bei den Verfahren, welche die Komplexität der Bedingungen im Programmtext des Testobjekts in den Mittelpunkt der Prüfung stellen, ist es sinnvoll, eine vollständige Prüfung (100 % Überdeckung) anzustreben. Ist die Komplexität der Bedingungen im Test ohne Bedeutung, kann eine Zweigüberdeckung als ausreichend angesehen werden.

### Bewertung der Methode

*Komplexe Bedingungen sind oft fehlerhaft*

Sind komplexe Bedingungen im Programmtext vorhanden, dann müssen diese intensiv getestet werden, um mögliche Fehlerzustände aufzudecken. Gerade bei der Kombination von logischen Ausdrücken werden oft Fehlhandlungen begangen, weshalb der Test sehr wichtig ist. Die minimale Mehrfachbedingungsüberdeckung ist allerdings auch ein sehr aufwändiges Verfahren zur Testfallerstellung.

*Tipp*

- Es kann auch sinnvoll sein, komplexe zusammengesetzte Bedingungen in verschachtelte, einfache Abfragen aufzuteilen und dann für diese Abfolge von Abfragen einen Zweigüberdeckungstest durchzuführen.
- Auf die intensive Prüfung oder Aufteilung von komplexen Bedingungen kann möglicherweise ganz verzichtet werden, wenn diese vor dem dynamischen Test einem Review (s. Kap. 4.1.5) unterzogen werden und deren Korrektheit dort nachgewiesen wird.

*Exkurs*

Ein Nachteil der Bedingungsüberdeckungen ist, dass sie boolesche Ausdrücke beispielsweise nur innerhalb einer IF-Anweisung prüft. Bei dem folgenden Beispiel eines Programmstücks wird nicht erkannt, dass die IF-Bedingung aus mehreren Teilbedingungen zusammengesetzt ist und die minimale Mehrfachbedingungsüberdeckung angewendet werden sollte.

```
 ...
Flag = (A || (B && C));
if (Flag)
        ...;
else    ...;
...
```

Werden im Programm alle vorkommenden booleschen Ausdrücke für die Erstellung der Testfälle herangezogen, kann dieser Nachteil ausgeglichen werden.

*Compiler bricht Auswertung von Ausdrücken ab*

Ein weiteres Problem tritt bei der Messung der Überdeckung der Teilbedingungen auf. Einige Compiler verkürzen die Auswertung von booleschen Ausdrücken, sobald das Ergebnis feststeht. Zum Beispiel ist bei einer AND-Verknüpfung von zwei Teilbedingungen für eine Teilbedingung der Wert FALSE ermittelt, dann ist die Gesamtbedingung ebenfalls FALSE, egal welchen Wert die zweite Teilbedingung liefert. Einige Compiler ändern auch die Reihenfolge der Auswertung in Abhängigkeit von den booleschen Operatoren, um möglichst schnell ein Endergebnis zu erhalten und die weiteren Teilbedingungen nicht auswerten zu müssen. Testfälle, die eine Überdeckung von 100 % erreichen sollen, können zwar ausgeführt werden, wegen der Verkürzung der Auswertung lässt sich die Überdeckung allerdings nicht nachweisen.

### 5.2.4 Pfadüberdeckung

*Alle möglichen Pfade durch ein Testobjekt*

Bisher standen Anweisungen oder Verzweigungen des Kontrollflusses oder die Komplexität von Bedingungen im Mittelpunkt der Testfallermittlung. Enthält das Testobjekt Schleifen bzw. Wiederholungen, so reichen die bisherigen Überlegungen nicht für einen angemessenen Test aus. Die Pfadüberdeckung fordert die Ausführung aller unterschiedlichen Pfade durch ein Testobjekt.

***Beispiel für einen Pfadtest***

Anhand des Kontrollflussgraphen (Abb. 5-7) soll der Begriff Pfad näher erläutert werden. Das Programmstück, das der Graph repräsentiert, enthält eine Schleife. Die DO-WHILE-Schleife wird auf jeden Fall einmal durchlaufen. In der WHILE-Bedingung am Ende der Schleife wird ermittelt, ob eine Wiederholung der Schleife, also ein Rücksprung zum Schleifenanfang, erfolgen soll. Bei der Ermittlung der Zweigüberdeckung ist die Schleife in zwei Testfällen berücksichtigt worden:

Schleife ohne Rücksprung:

a, b, f, g, h, d, e

Schleife mit einmaligem Rücksprung (i) und einmaliger Wiederholung:

a, b, f, g, i, g, h, d, e

In der Regel wird eine Schleife aber häufiger als einmal durchlaufen werden. Mögliche weitere Abfolgen von Zweigen durch den Programmgraphen sind

a, b, f, g, i, g, i, g, h, d, e
a, b, f, g, i, g, i, g, i, g, h, d, e
a, b, f, g, i, g, i, g, i, g, i, g, h, d, e
usw.

*Kombination der Programmteile*

Ein Pfad beschreibt die mögliche Abfolge von einzelnen Programmteilen in einem Programmstück. Im Gegensatz dazu werden Zweige unabhängig voneinander betrachtet, jeder für sich. Die Pfade berücksichtigen Abhängigkeiten zwischen den Zweigen, wie zum Beispiel bei Schleifen, bei denen ein Zweig an den Anfang eines anderen Zweiges zurückführt.

*Beispiel Anweisungs- und Zweigüberdeckung* VCR

In Kapitel 5.1.1 wurden für die Methode `calculate_price()` des VSR-Teilsystems *DreamCar* Testfälle aus gültigen und ungültigen Äquivalenzklassen der Parameter hergeleitet. Im Folgenden werden Testfälle dahingehend bewertet, inwieweit sie den Programmtext überdecken, d. h. entsprechende Teile der Methode zur Ausführung gekommen sind. Es soll eine 100%ige Zweigüberdeckung erreicht werden, um sicherzustellen, dass während der Testläufe alle Zweige mindestens einmal zur Ausführung gekommen sind.

Zum besseren Verständnis wird hier der Programmtext der Methode aus Kap. 3.2.3 wiederholt:

```
double calculate_price(
        double baseprice, double specialprice,
        double extraprice, int extras, double discount)
{
        double addon_discount;
        double result;

        if (extras >= 3) addon_discount = 10;
        else if (extras >= 5) addon_discount = 15;
        else addon_discount = 0;

        if (discount > addon_discount)
                addon_discount = discount;

        result = baseprice/100.0*(100-discount)
                + specialprice
                + extraprice/100.0*(100-addon_discount);

        return (result);
}
```

Der Kontrollflussgraph der Methode `calculate_price` ist in Abb. 5-8 dargestellt.

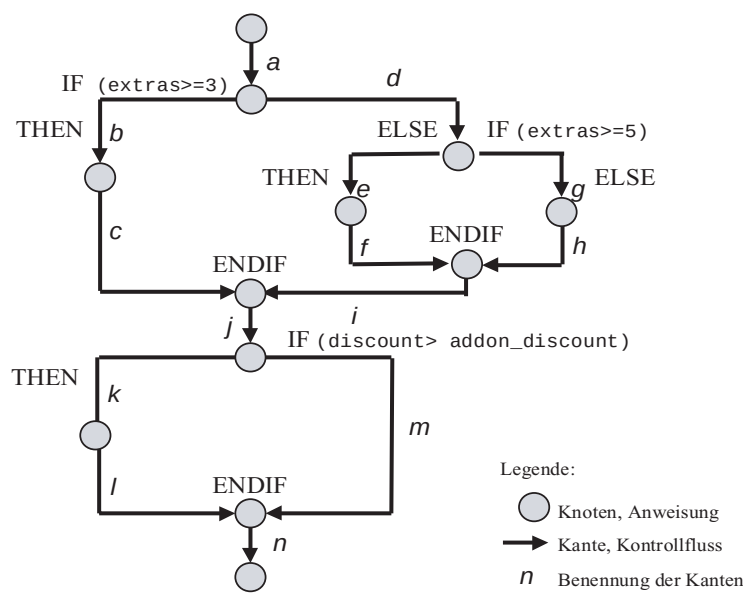



***Abb. 5–8***
*Kontrollflussgraph der Methode* `calculate_price`

In Kapitel 3.2.3 wurden folgende beide Testfälle angegeben:

```
// testcase 01
price = calculate_price(10000.00,2000.00,1000.00,3,0);
test_ok = test_ok && (price == 12900.00);

// testcase 02
price = calculate_price(25500.00,3450.00,6000.00,6,0);
test_ok = test_ok && (price == 34050,00);
```

Die Testfälle bewirken die Ausführung folgender Kanten des Graphen:

Testfall 01: a, b, c, j, m, n
Testfall 02: a, b, c, j, m, n

*46 % Zweigüberdeckung erreicht*

Die Kanten d, e, f, g, h, i, k, l sind nicht zur Ausführung gekommen. Durch die beiden Testfälle wurden insgesamt nur 43 % der Zweige (6 von insgesamt 14) überdeckt. Testfall 02 hat keine Verbesserung der Überdeckung gebracht und war aus Sicht der Zweigüberdeckung unnütz. Nach Spezifikation sollte der Testfall 02 allerdings zur Ausführung weiterer Anweisungen führen, da ein anderer Rabatt berechnet werden sollte (bei fünf oder mehr Zusatzausstattungen).

Zur Erhöhung der Überdeckung werden folgende weiteren Testfälle spezifiziert:

```
// testcase 03
price = calculate_price(10000.00,2000.00,1000.00,0,10);
test_ok = test_ok && (price == 12000.00);

// testcase 04
price = calculate_price(25500.00,3450.00,6000.00,6,15);
test_ok = test_ok && (price == 30225.00);
```

Diese Testfälle bewirken die Ausführung folgender Kanten des Graphen:

Testfall 03: a, d, g, h, i, j, k, l, n
Testfall 04: a, b, c, j, k, l, n

*86 % Zweigüberdeckung erreicht*

Die Testfälle führen zur Ausführung von weiteren Kanten (d, g, h und i) und damit zur Erhöhung der Überdeckung auf 86 %. Die Kanten e und f sind nicht ausgeführt worden.

*Auswertung der Bedingungen*

Bevor mit weiteren Testfällen versucht wird, die beiden fehlenden Kanten zu erreichen, werden die Bedingungen der IF-Anweisungen näher betrachtet, d. h., der Programmtext wird zur Ermittlung weiterer Testfälle herangezogen. Um die Kante e und f zu erreichen, muss die Bedingung der ersten Abfrage `(extras >= 3)` den Wert `FALSE` liefern, damit der ELSE-Teil ausgeführt wird. In diesem ELSE-Teil muss die Bedingung (`extras >= 5`) den Wert `TRUE` liefern. Es muss somit ein Wert für den Parameter `extras` gefunden werden, der folgender Bedingung genügt:

```
¬(extras >= 3) AND (extras >= 5)
```

Einen solchen Wert gibt es nicht, die fehlenden Kanten sind nie erreichbar. Es liegt ein Defekt im Programmtext vor.

---

***Beispiel Zusammenhang der Maße***

Der Zusammenhang zwischen Anweisungs-, Zweig- und Pfadüberdeckung soll ebenfalls an diesem Beispiel verdeutlicht werden. Das Testobjekt besteht aus insgesamt drei IF-Anweisungen, wovon zwei verschachtelt sind und die dritte separat zu den anderen steht (s. Abb. 5-8).

Alle Anweisungen (Knoten) werden durch folgende Abfolgen der Kanten des Graphen erreicht:

a, b, c, j, k, l, n
a, d, e,f, i, j, k, l, n
a, d, g, h, i, j, k, l, n

Diese Abfolgen reichen aus, um eine Anweisungsüberdeckung von 100 % zu erhalten. Alle Zweige (Kanten) wurden allerdings noch nicht überdeckt, es fehlt die Kante m. Ein Ablauf könnten wie folgt aussehen

a, b, c, j, m, n

Diese zusätzliche Abfolge sollte die obige erste Abfolge ersetzen. Mit den dann drei möglichen Testfällen, die diese drei Abfolgen bewirken, ist eine 100 %-Zweigüberdeckung erreicht.

*Weitere Pfade durch den Graphen*

Es gibt allerdings auch bei diesem einfachen Programmstück noch weitere Möglichkeiten, den Graphen zu durchlaufen und somit alle Pfade durch den Graphen zu berücksichtigen. Bisher sind die folgenden Pfade nicht zur Ausführung gekommen:

a, d, e,f, i, j, m, n
a, d, g, h, i, j, m, n

Insgesamt ergeben sich somit sechs unterschiedliche Pfade durch den Programmtext (die möglichen drei Wege durch den Graphen vor der Kante j multipliziert mit den zwei möglichen Wegen nach der Kante j). Voraussetzung ist, dass die Abfragen unabhängig voneinander und die Kanten somit frei kombinierbar sind.

---

Kommen im Programmstück noch Schleifen hinzu, so zählt jede mögliche Anzahl von Schleifenwiederholungen als ein möglicher Pfad durch das Programmstück. Es ist klar, dass eine 100%-Überdeckung aller Pfade in einem Programm während der Testphase nie erreichbar ist, sobald das Programm nicht trivial ist.

### 5.2.5 Allgemeine Bewertung der White-box-Verfahren

Auch bei den White-box-Verfahren gibt es eine ganze Reihe von weiteren Verfahren, die zur Prüfung eines Testobjekts herangezogen werden können, hier aber nicht beschrieben sind. Die gebräuchlichsten Verfah-

ren wurden vorgestellt. Weitere Verfahren sind in [Myers 82], [Pol 00], [Riedemann 97] und [Sneed 02] erklärt. Im Folgenden wird auf zwei Verfahren etwas näher eingegangen.

#### Weitere White-box-Verfahren

*Linear Code Sequence and Jump (LCASJ)*

Sind Programme zu testen, die in älteren Programmiersprachen vorliegen, so können diese Testobjekte Sprünge enthalten. Die bisher vorgestellten Testverfahren berücksichtigen den Test der Sprünge nur unzureichend. Anders das →*Linear Code Sequence and Jump* (LCASJ)-Verfahren. In dem Verfahren werden die Folgen von Anweisungen in den Mittelpunkt der Untersuchung gestellt, die sequenziell ausgeführt werden und durch eine Sprung auf eine andere Anweisung als die nächstfolgende Anweisung beendet werden. Kombinationen solcher linearer Codesequenzen mit einem Sprung am Ende sind beim Test zu berücksichtigen. Das Verfahren prüft durch die Kombination der Sequenzen mehr als die Zweigüberdeckung, ist allerdings nicht so umfangreich wie die Pfadüberdeckung. Eine ausführliche Beschreibung findet sich in [Liggesmeyer 90].

*Datenflussbasierte Verfahren*

Eine ganze Reihe von Verfahren verwenden den Datenfluss durch das Testobjekt als Grundlage für die Ermittlung der Testfälle. Es werden im Wesentlichen die Datenbenutzungen im Testobjekt überprüft. Für jede Variable wird deren Verwendung analysiert. Es werden die Definition von Variablen und die lesenden und schreibenden Zugriffe auf Variablen unterschieden. Darüber hinaus wird betrachtet, ob der Wert der Variablen zur Berechnung einer anderen Variablen oder zur Ermittlung eines Wahrheitswertes in einer Bedingung verwendet wird. Anhand dieser Information können unterschiedliche Kriterien in Bezug auf den Datenfluss festgelegt werden, die durch die Testfälle zu überdecken sind. Eine ausführlicher Beschreibung der datenflussbasierten Verfahren ist in [Balzert 98] und [Riedemann 97] zu finden.

#### Vor- und Nachteile der White-box-Verfahren

Grundlage aller White-box-Verfahren ist der vorliegende Programmtext. In Abhängigkeit der Komplexität der Programmstruktur können die adäquaten Testfallentwurfsverfahren ausgewählt und angewendet werden. Anhand des Programmtextes und der ausgewählten Verfahren wird die Intensität der Tests festgelegt.

Die White-box-Verfahren eignen sich für die unteren Teststufen. Es ist beispielsweise wenig sinnvoll, beim Systemtest eine Überdeckung einzelner Anweisungen oder Zweige zu fordern, da der Systemtest sich nicht zur Überprüfung von einzelnen Anweisungen oder Bedingungen eignet.

*»Nicht vorhandener Programmcode« bleibt unberücksichtigt*

Anforderungen, die übersehen und daher nicht realisiert wurden, werden durch die White-box-Verfahren nicht aufgedeckt. Nur die Anforderungen, die auch im Programm umgesetzt worden sind, werden bei den White-box-Verfahren überprüft.

### 5.2.6 Instrumentierung und Werkzeugunterstützung

*Ermittlung der ausgeführten Programmteile*

Bei den White-box-Verfahren wird gefordert, dass bestimmte Programmteile zur Ausführung kommen bzw. Bedingungen unterschiedliche Wahrheitswerte annehmen. Um den Test auswerten zu können, muss ermittelt werden, welche Programmteile bereits ausgeführt wurden und welche noch nicht zur Ausführung gekommen sind. Dazu muss das Testobjekt an strategisch wichtigen Stellen vor der Testausführung instrumentiert werden. Bei einer →Instrumentierung werden zusätzlich Zähler eingebaut und mit null initialisiert, die dann beim Durchlauf an den entsprechenden Stellen inkrementiert werden. Am Ende der Testläufe enthalten die Zähler die Anzahl von Durchläufen durch die jeweiligen Programmteile. Ist ein Zähler auf null geblieben, so sind die entsprechenden Programmteile nicht ausgeführt worden.

*Werkzeuge verwenden*

Die Instrumentierung, die Auswertung der Testläufe und die Ermittlung der erreichten Überdeckungen sind nicht von Hand vorzunehmen, da dies zu viele Ressourcen benötigt und eine manuelle Instrumentierung fehleranfällig wäre. Es gibt eine große Zahl von Werkzeugen, die diese Aufgaben übernehmen (s. Kap. 7.1.4). Diese Werkzeuge sind beim White-box-Test sehr wichtig, denn sie erhöhen die Produktivität und steigern dadurch auch indirekt die Qualität des Testobjekts.

## 5.3 Intuitive Testfallermittlung

Neben den methodischen Ansätzen ist die intuitive Ermittlung der Testfälle durchzuführen. Durch die →intuitive Testfallermittlung werden die systematisch ermittelten Testfälle sinnvoll ergänzt. Intuitives Testen kann Fehlerwirkungen aufdecken, die durch systematisches Testen übersehen werden können.

*Intuitive Fähigkeit und Erfahrung der Tester*

Grundlage der Überlegungen ist die intuitive Fähigkeit und die Erfahrung, Testfälle nach erwarteten Fehlerzuständen und -wirkungen auszuwählen. Ein methodisches Vorgehen hierzu kann nicht angegeben werden. Die Testfälle basieren auf der Erfahrung, wo Fehlerhandlungen in der Vergangenheit aufgetreten sind, oder der Vermutung des Testers, wo Fehlerhandlungen in Zukunft wahrscheinlich auftreten werden.

Die intuitive Testfallermittlung lässt sich nicht eindeutig den Black- oder White-box-Verfahren zuordnen, da weder die Anforderungen noch der Programmtext ausschließliche Grundlage für die Überlegungen und Prüfungen sind. Ihr Anwendungsbereich liegt aber eher in den höheren Teststufen, da auf den niedrigeren meist ausreichende Informationen für die anderen Verfahren zur Verfügung stehen, wie beispielsweise der Programmtext oder eine detaillierte Spezifikation.

*Nicht als erstes oder einziges Vorgehen verwenden*

Als primäres Testverfahren sollte die intuitive Testfallermittlung nicht eingesetzt werden, sondern sie sollte der Fertigstellung von Testfällen und zur Unterstützung von methodischen Testverfahren dienen.

### Testfälle

Das Erfahrungswissen der Tester zur Erstellung der ergänzenden Testfälle kann aus vielen Quellen gespeist werden.

***Wissen der Tester beim CarConfigurator***

Im Falle des Projekts zur Entwicklung des *CarConfigurator* sind die Tester sehr gut mit dem Vorgängersystem vertraut. Viele von ihnen haben auch dieses System schon getestet. Sie wissen, welche Schwächen dieses System hatte und kennen (aus Daten der Hotline und aus persönlichen Gesprächen) auch die Probleme der Autohäuser beim Betrieb der alten Software. Mitarbeiter aus dem Konzernmarketing wissen beim fachlichen Test, welche Fahrzeuge in welcher Konstellation oft verkauft werden und welche theoretisch möglichen Sonderausstattungsvarianten vielleicht gar nicht lieferbar sind. Diese Erfahrung nutzen sie, indem sie intuitiv ihre systematisch hergeleiteten Testfälle priorisieren und durch zusätzliche Testfälle ergänzen. Der Testmanager schließlich weiß, welche Entwicklerteams besonders unter Druck stehen und auch an Wochenenden durcharbeiten. Er wird daher die Komponenten dieser Teams intensiver testen lassen.

*Gesamtes Wissen nutzen*

Der Tester soll sein gesamtes Wissen nutzen, um zu ergänzenden Testfällen zu gelangen. Die Vor- und Nachbedingungen, die erwarteten Ausgaben und das erwartete Verhalten des Testobjekts ist selbstverständlich auch bei den intuitiven Testfällen vorab festzulegen.

***Tipp***

- Da umfangreiches Erfahrungswissen oft nur in den Köpfen der erfahrenen Tester vorhanden ist, kann das Führen einer Liste mit möglichen Fehlhandlungen, Fehlerzuständen und fehlerverdächtigen Situationen sehr nützlich sein. In der Liste werden die Erfahrungen über immer wieder auftretende Fehler vermerkt und stehen somit allen Testern zur Verfügung. Anhand der identifizierten möglichen Fehler und kritischen Situationen werden die zusätzlichen Testfälle erstellt.

- Die Liste kann auch für Entwickler von großem Nutzen sein, da vor der Implementierung schon auf eventuelle Probleme und Schwierigkeiten hingewiesen wird, die dann in der Implementierung berücksichtigt werden können und somit der Fehlervermeidung dienen.

---

**Festlegung der Testendekriterien**

*Das Endekriterium ist nicht definierbar*

Ein Endekriterium wie bei den systematischen Verfahren lässt sich nicht angeben. Existiert die oben erwähnte Liste, kann anhand der Liste eine gewisse Vollständigkeit überprüft werden.

**Bewertung der intuitiven Testfallermittlung**

*Meist erfolgreich in der Aufdeckung weiterer Fehler*

Die intuitive Testfallermittlung lässt sich meist mit gutem Erfolg einsetzen. Das wirkungsvolle Vorgehen lässt sich für alle Testobjekte und auf allen Teststufen einsetzen. Es sollte allerdings nur zur Ergänzung der systematischen Verfahren dienen. Es können keine Messungen der Intensität oder Vollständigkeit der intuitiven Testfallermittlung durchgeführt werden.

## 5.4 Zusammenfassung

*Wann welches Verfahren?*

In diesem Kapitel wurden eine ganze Reihe von Verfahren zum Testen von Software vorgestellt. Es drängt sich die Frage auf, wann welches Verfahren verwendet werden soll? Im Folgenden wird zusammenfassend eine Hilfestellung bei der Beantwortung dieser Frage gegeben und ein sinnvolles Vorgehen aufgezeigt. Allgemeines Ziel ist, mit wenig Aufwand ausreichend unterschiedliche Testfälle zu erstellen, die mit einer gewissen Wahrscheinlichkeit vorhandene Fehlerwirkungen nachweisen. Die Verfahren zur Erstellung der Testfälle sollten dabei passend zum Testobjekt ausgewählt werden.

*Prüfung der Funktionalität*

- Von zentraler Bedeutung bei einem Softwaresystem ist sicherlich die korrekte Funktionalität des Systems. Eine ausreichende Prüfung der Funktionalität des Testobjekts ist auf jeden Fall zu gewährleisten. Da bei der Aufstellung aller Testfälle, egal nach welchem Vorgehen oder Verfahren, die erwarteten Ergebnisse und Reaktionen des Testobjekts mit vermerkt werden müssen, erfolgt eine Prüfung der Funktionalität bei jeder Auswertung der durchgeführten Testfälle. Im Vergleich der erwarteten und tatsächlichen Ausgabedaten bzw. Reaktionen ist die Überprüfung der Funktionalität mit enthalten. Es kann entschieden werden, ob ein Fehlverhalten vorliegt oder die korrekte Funktionalität realisiert wurde.

*Äquivalenzklassenbildung in Kombination mit der Grenzwertanalyse*

- Bei jedem Testobjekt sollte die Äquivalenzklassenbildung in Kombination mit der Grenzwertanalyse zur Erstellung der Testfälle eingesetzt werden. Bei der Ausführung dieser Testfälle sollten die entsprechenden Werkzeuge zur Messung der codebasierten Überdeckungen bereits eingesetzt werden. So kann nach der Durchführung aller Testfälle der Äquivalenzklassenbildung und Grenzwertanalyse bereits die erreichte Anweisungs- oder Zweigüberdeckung ermittelt werden.

*Historie berücksichtigen*

- Haben unterschiedliche Zustände einen Einfluss auf den Ablauf innerhalb des Testobjekts, muss ein zustandsbezogener Test durchgeführt werden. Nur der zustandsbezogene Test überprüft das Zusammenspiel der Zustände, der Zustandsübergänge und das entsprechende Verhalten der Funktionen in adäquater Weise.
- Bisher nicht ausgeführte Teile des Testobjekts werden dann gezielt einem der White-box-Testverfahren unterzogen. Je nach Kritikalität und Beschaffenheit des Testobjekts muss ein entsprechend aufwändiges White-box-Verfahren gewählt werden.

*Minimales Kriterium: Zweigüberdeckung*

- Als minimales Kriterium sollte die Zweigüberdeckung verwendet werden. Sind komplexe Bedingungen im Testobjekt vorhanden, so ist die minimale Mehrfachbedingungsüberdeckung das adäquate Verfahren, um fehlerhafte Bedingungen zu erkennen.
- Bei den Überdeckungsmessungen sollte darauf geachtet werden, dass Schleifen auch mehr als einmal wiederholt werden. Bei kritischen Systemteilen muss die Prüfung der Schleifen anhand von entsprechenden Methoden erfolgen (siehe hierzu *boundary interior*-Pfadtest und strukturierter Pfadtest [Balzert 98]).
- Die Pfadüberdeckung ist als eher theoretisches Maß anzusehen und hat auf Grund des riesigen Aufwands für die Praxis keine Bedeutung.
- Die White-box-Verfahren lassen sich sinnvollerweise auf den unteren Teststufen einsetzen; auf den oberen Teststufen sind die Black-box-Verfahren die adäquaten Methoden.
- Auf die intuitive Ermittlung der Testfälle als Ergänzung sollte nicht verzichtet werden. Es ist immer sinnvoll, die Erfahrungen der Tester zu nutzten, um weitere Fehler aufzudecken.

---

***Tipp***

- Testen umfasst immer die Kombination von unterschiedlichen Verfahren, da es kein Testverfahren gibt, das alle Aspekte, die beim Testen zu berücksichtigen sind, gleich gut abdeckt.
- Die Auswahl der Testverfahren und die Intensität der Durchführung sind anhand der Kritikalität und dem zu erwartenden Risiko im Fehlerfall festzulegen.

- Bei den White-box-Verfahren soll die Struktur des Testobjekts die Grundlage bei der Auswahl der Verfahren sein. Sind beispielsweise keine komplexen Bedingungen im Testobjekt vorhanden, macht die Verwendung der minimalen Mehrfachbedingungsüberdeckung keinen Sinn.

# 6 Testmanagement

*Dieses Kapitel behandelt, wie Testteams organisiert sein sollten, welche Mitarbeiterqualifikationen wichtig sind, welche Aufgaben ein Testmanager hat und welche testunterstützenden Prozesse vorhanden sein müssen, um effizient testen zu können.*

## 6.1 Organisation von Testteams

Über den gesamten Lebenszyklus eines Softwareprodukts hinweg sind Testarbeiten durchzuführen (s. Kap. 3). Diese sind mit den Entwicklungsarbeiten gut und eng zu koordinieren. Die Testaktivitäten eines Entwicklungsprojekts müssen dazu geeignet organisiert sein. Grundsätzlich sind folgende Modelle einer Aufgabenteilung zwischen Entwicklung und Test möglich:

*Organisationsformen*

1. Testen liegt ausschließlich in der Verantwortung des einzelnen Entwicklers. Jeder Entwickler testet seine eigenen Programmteile.
2. Testen liegt in der Verantwortlichkeit des Entwicklungsteams. Die Entwickler testen ihre Programmteile gegenseitig.
3. Mindestens ein Mitglied des Entwicklerteams ist für Testarbeiten abgestellt. Es erledigt alle Testarbeiten des Teams.
4. Es gibt ein oder mehrere dedizierte Testteams innerhalb des Projekts (die nicht an der Entwicklung beteiligt sind).
5. Eine separate Organisation (Testabteilung, externer Testdienstleister, Testlabor) übernimmt das Testen.

In jedem dieser Modelle ist es vorteilhaft, (interne) Test-Consultants verfügbar zu haben, die mehreren Projekten beratend zur Seite stehen und diese bei ihren Testarbeiten methodisch unterstützen (Training, Coaching, Testautomatisierung usw.). Welches dieser Modelle zweckmäßig ist, hängt unter anderem von der betrachteten Teststufe ab.

**Komponententest**: Hier muss relativ entwicklungsnah gearbeitet werden (s. Kap. 3.2). Modell 1 ist, obwohl oft anzutreffen, sicher die schlechteste Wahl. Ein Wechsel zu Modell 2 kann in den meisten Fällen organisiert werden und bringt sicher Qualitätsgewinne. Modell 3 ist sinnvoll, sofern in Relation zur Entwicklungsmannschaft genügend viele Tester abgestellt werden. Bei allen drei Varianten besteht die Gefahr, dass sich die beteiligten Personen im Wesentlichen als Entwickler

verstehen und die Testarbeiten deshalb vernachlässigt werden. Um dies auszugleichen, sind zwei Maßnahmen empfehlenswert:

**Tipp**

- Projekt- oder Testleitung sollten Testpläne vorgeben und von den Entwicklern Testprotokolle einfordern.
- Zur methodischen Unterstützung sollten zumindest zeitweise Testspezialisten als Coach hinzugezogen werden.

**Integrationstest:** Sofern Komponenten zu integrieren und zu testen sind, die vom selben Team entwickelt wurden, kann ein Vorgehen analog zum Komponententest organisiert werden (Modelle 1-3).

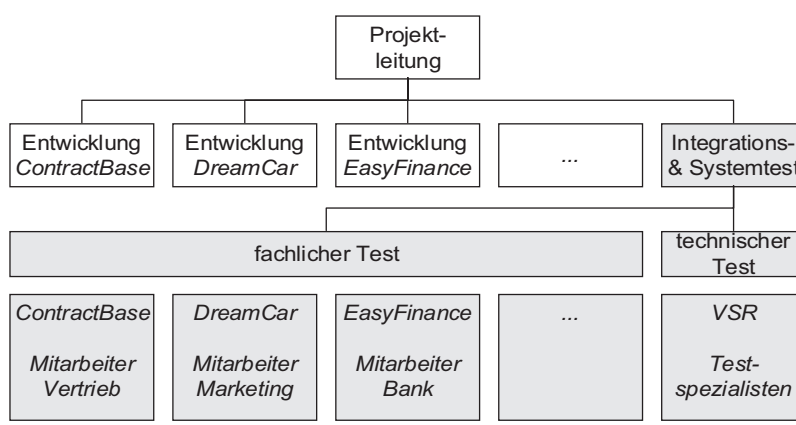
Müssen Komponenten zusammengeführt werden, die von verschiedenen Teams stammen, sollte ein gemischtes Integrationsteam mit Vertretern aus den betroffenen Entwicklungsgruppen oder ein unabhängiges Integrationsteam tätig werden. Das einzelne Entwicklungsteam wird eine einseitige Sichtweise auf die eigene Komponente haben und deshalb Fehler übersehen. Je nach Größe des Entwicklungsvorhabens und Anzahl der Komponenten kommen hier die Modelle 3–5 in Frage.

**Systemtest:** Das Gesamtprodukt muss aus dem Blickwinkel des Kunden und späteren Anwenders betrachtet werden. Unabhängigkeit von der Entwicklung ist hier unabdingbar. Damit kommen nur die Modelle 4 und 5 in Frage.

**Beispiel: Organisation der VSR-Tests**

Im *VSR*-Projekt liegen die Komponententests in der Verantwortung der jeweiligen Entwicklungsteams und sind dort individuell nach den oben beschriebenen Modellen 1–3 organisiert. Parallel zu den Entwicklungsgruppen ist eine unabhängige Testgruppe eingerichtet. Diese ist verantwortlich für den Integrations- und Systemtest. Abbildung 6-1 zeigt diese Organisation.

**Abb. 6–1**
*VSR-Projektorganisation*



Für den funktionalen oder fachlichen Test jedes Teilsystems (*ContractBase*, *DreamCar* usw.) sind zwei bis drei Mitarbeiter des jeweils zuständigen Konzernfachbereichs abgestellt (Vertrieb, Marketing u. a.). Diese kennen die vom jeweiligen Teilsystem realisierten Geschäftsprozesse und wissen, welche Anforderungen »ihr« Testobjekt aus Anwendersicht erfüllen muss. Sie sind erfahren im Umgang mit dem PC, aber keine DV-Spezialisten. Ihre Aufgabe ist es, die Testspezialisten bei der Spezifikation fachlich sinnvoller Testfälle zu unterstützen und diese Tests auch durchzuführen. Zu Beginn der Testarbeiten wurden sie in den Grundlagen des Testens geschult (Testprozess, -spezifikation, -durchführung, -protokollierung).

Zum Testpersonal gehören des Weiteren drei bis fünf IT- und Testspezialisten, die für Integrationsarbeiten, nicht funktionale Tests, Testautomatisierung und die Betreuung der Testwerkzeuge zuständig sind (»technischer Test«).

Geleitet wird die Testgruppe von einem Testmanager, der Testplanung und Teststeuerung verantwortet. Zu seinen Aufgaben gehört auch das Coaching des Testpersonals, insbesondere die Anleitung der Mitarbeiter beim Testen der fachlichen Anforderungen.

## 6.2 Mitarbeiterqualifikation

Zur Durchführung der Testarbeiten sollten Spezialisten zur Verfügung stehen, deren Know-how alle Aufgabenbereiche im Testprozess abdeckt. Folgende Rollen sind auszufüllen und im Idealfall mit speziell qualifizierten Mitarbeitern zu besetzen:

*Rollen und Qualifikationsprofile*

- **Testmanager**: Experte(n) für Testplanung und Teststeuerung mit Know-how/Erfahrung in den Bereichen Softwaretest, Qualitätsmanagement, Projektmanagement, Personalführung.
- **Testdesigner**: Experte(n) für Testmethoden und Testspezifikation mit Know-how/Erfahrung in den Bereichen Softwaretest, Softwareengineering und (formalen) Spezifikationsmethoden.
- **Testautomatisierer**: Experte(n) für →Testautomatisierung (Testgrundlagenwissen, Programmiererfahrung und mit sehr guter Kenntnis der eingesetzten Testwerkzeuge).
- **Testadministrator**: Experte(n) für Installation und Betrieb der Testumgebung (Systemadministrator-Know-how).
- **Tester**[34]: Experte(n) für Testdurchführung und Fehlerberichte (IT-Grundlagen, Testgrundlagenwissen, Bedienung der eingesetzten Testwerkzeuge, Verständnis des Testobjekts).

34. Der Begriff »Tester« wird teilweise auch als Oberbegriff für die genannten Rollen verwendet.

*Certified Tester*

Was leistet in diesem Zusammenhang die Ausbildung zum Certified Tester? Die Grundlagenausbildung (Foundation Level) qualifiziert zur Rolle des »Testers« (ohne die erforderlichen IT-Grundlagen). Das heißt, ein Certified Tester weiß, warum Disziplin und strukturiertes Vorgehen notwendig sind, und kann unter Leitung eines Testmanagers Tests manuell durchführen und dokumentieren. Basistechniken aus dem Bereich Testspezifikation und -management sind ihm bekannt. Um die Rolle eines »Testdesigners« oder »Testmanagers« aber in der Praxis ausfüllen zu können, muss entsprechende Erfahrung als Tester erst noch gesammelt werden. Die Ausbildung für diese Rollen leistet die nächste Ausbildungsstufe (Practitioner Level). Auch jeder Softwareentwickler sollte über Grundlagenkenntnisse im Testen verfügen, um die im Rahmen der Organisationsmodelle 1 bis 3 sich ergebenden Testaufgaben erledigen zu können.

*Auch soziale Kompetenz ist wichtig*

Neben seinen technischen und testspezifischen Fähigkeiten benötigt ein Tester, um erfolgreich zu sein, auch soziale Kompetenz:

- Teamfähigkeit, politisches und diplomatisches Geschick
- Bereitschaft, scheinbare Tatsachen zu hinterfragen
- Durchsetzungskraft, sicheres Auftreten
- Exaktheit und Kreativität
- Fähigkeit, sich schnell in komplexe Anwendungsgebiete und Applikationen einzuarbeiten.

*Multidisziplinäres Team*

Besonders beim Systemtest ist es oft notwendig, die Testmannschaft durch weitere IT-Spezialisten zu ergänzen, die zumindest zeitweise zuarbeiten. Dies sind zum Beispiel: Datenbankadministratoren, Datenbankdesigner oder Netzwerkspezialisten. Unverzichtbar sind oft auch Branchen- oder Fachspezialisten aus dem Anwendungsbereich des zu testenden Softwaresystems. Ein solches multidisziplinäres Testteam zu führen, ist auch für erfahrene Testmanager nicht immer einfach.

*Spezialisierte Softwaretestdienstleister*

Sind entsprechende Ressourcen firmenintern nicht ausreichend verfügbar, so können – analog zur Vergabe von Softwareentwicklungsaufträgen an externe Entwicklungshäuser – auch die Testaufgaben an spezialisierte Softwaretestdienstleister vergeben werden. Solche Testspezialisten können aufgrund ihrer Erfahrung und »schlüsselfertiger« Vorgehensweisen den für das jeweilige Projekt optimalen Testbetrieb sehr schnell aufsetzen sowie fehlendes Spezialwissen aus jedem der oben angesprochenen Qualifikationsprofile in das Projekt einbringen.

## 6.3 Management der Testarbeiten

*Aufgaben des Testmanagers*

Der Testprozess besteht nach [BS 7925-2] aus den Phasen: Testplanung, Testspezifikation, Testdurchführung, Testprotokollierung und Testbewertung (vgl. Kap. 2.2, Abb. 2-4).

Ein Durchlauf durch diese Schritte stellt einen Testzyklus dar, an dessen Ende Fehlerkorrektur- oder Änderungsaufträge für die Entwickler vorliegen. Dieser Testzyklus spielt sich auf jeder Teststufe ab. Planung und Spezifikation erfolgen dabei jeweils schon parallel zu den Entwicklungsarbeiten der jeweiligen Stufe (linker Ast im V-Modell, vgl. Kap 3.1). Diese Testzyklen zu initiieren, zu überwachen und zu steuern, ist Aufgabe des Testmanagers. Dabei kann je nach Projektgröße für jede Teststufe ein eigener Testmanager zuständig sein.

### 6.3.1 Testzyklusplanung

Die grobe Testplanung (Teststrategie und grober Zeitablauf) wird zu einem möglichst frühen Zeitpunkt im Projekt erstellt und im Testkonzept festgehalten (s. Kap. 3.8.2).

*Feinplanung je Testzyklus*

Diese Grobplanung muss durch eine Feinplanung für den jeweils anstehenden konkreten Testzyklus ergänzt und regelmäßig der aktuellen Projektsituation angepasst werden. Folgende Punkte sind zu berücksichtigen:

- **Entwicklungsstand**: Die zu Beginn des Testzyklus tatsächlich verfügbare Software kann eine gegenüber den ursprünglichen Plänen möglicherweise eingeschränkte oder veränderte Funktionalität ausweisen. Daraus resultieren unter Umständen Anpassungen von Testspezifikationen und Testfällen.
- **→Testergebnisse**: Die in vorangehenden Testzyklen aufgedeckten Probleme machen eventuell eine Änderung der Testpriorisierung notwendig. Korrigierte Defekte erfordern zusätzliche Fehlernachtests, die ebenfalls neu einzuplanen sind; zusätzliche Tests können auch notwendig sein, weil Probleme noch nicht vollständig reproduziert und analysiert werden konnten.
- **Ressourcen**: Die Planung des aktuellen Testzyklus muss mit dem aktuellen Projektplan in Einklang stehen; zu beachten sind z. B. Auswirkungen der aktuellen Personaleinsatz- und Urlaubsplanung, die momentane Verfügbarkeit der Testumgebung und spezieller Testwerkzeuge usw.

*Planung des Testaufwands*

Unter Berücksichtigung dieser Ausgangsdaten schätzt der Testmanager Aufwand und Zeitbedarf der Testarbeiten und legt im Detail fest, wel-

che Testfälle, von welchem Tester in welcher Reihenfolge, zu welchen Zeitpunkten auszuführen sind. Ergebnis dieser Feinplanung ist der (Regressions-)Testplan für den anstehenden Testzyklus.

### 6.3.2 Testzyklusüberwachung

Die Überwachung und Erfolgskontrolle der laufenden Tests geschieht anhand objektiver →Testmetriken, die im Testkonzept vereinbart wurden. Dabei ist darauf zu achten, dass nur solche Metriken verwendet werden, die regelmäßig, zuverlässig und einfach zu messen[35] sind. Zu unterscheiden sind u. a. folgende Ansätze:

*Maße für die Verfolgung des Testprozesses*

- **Fehlerbasierte Metriken:** Anzahl gefundener Fehlerzustände bzw. erstellter Fehlermeldungen (pro Testobjekt) im jeweiligen Release, in Abhängigkeit der Fehlerklasse und des Fehlerstatus, ggf. bezogen auf Größe des Testobjekts (engl. *lines of code*), Testdauer o. Ä. (vgl. Kap. 6.4).
- **Testfallbasierte Metriken:** Anzahl spezifizierter oder geplanter Tests, Anzahl →blockierter Tests (z. B. wegen nicht beseitigter Fehlerzustände), Anzahl gelaufener, nicht →Fehler aufdeckender Testfälle, Anzahl gelaufener Fehler aufdeckender Testfälle.
- **Testobjektbasierte Metriken:** Codeabdeckung, Dialogabdeckung, abgedeckte Installationsvarianten, Plattformen usw.

*Teststatusbericht*

Die jeweils aktuellen Messergebnisse listet der Testmanager in seinen Berichten auf. Nach jedem Testzyklus erstellt er einen Teststatusbericht, aus dem folgende Informationen über den Stand der Testarbeiten hervorgehen:

- Testobjekt(e), Teststufe, Testzyklus-Datum von ... bis
- Testfortschritt: geplante/gelaufene/blockierte Tests
- Fehlerstatus: neue/offene/korrigierte Fehler
- Risiken: neue/veränderte/bekannte
- Ausblick: Planung des nächsten Testzyklus
- Gesamtbewertung: Beurteilung der Freigabereife des Testobjekts

*Kriterien für das Testende*

Die gemessenen Daten dienen auf der einen Seite zur Standortbestimmung und zur Beantwortung der Frage: »Wie weit ist der Test vorangekommen?« Auf der anderen Seite dienen sie als Endekriterien und zur Klärung der Frage: »Kann der Test beendet und das Produkt ausgeliefert werden?« Welche Kriterien zur Bestimmung des Testendes sinnvoll und angemessen sind, hängt von den zu erfüllenden Qualitäts-

35. Falls die eingesetzten Testwerkzeuge diese Daten liefern, ist dies gegeben.

anforderungen (Kritikalität der Software) ab, aber auch von den verfügbaren Testressourcen (Zeit, Personal, Werkzeuge). Die im Projekt geltenden Testendekriterien werden ebenfalls im Testkonzept festgelegt. Jedes Testendekriterium muss so gewählt sein, dass es sich aus den laufend erhobenen Testmetriken berechnen lässt.

***Beispiel: Testendekriterien für den VSR-Systemtest***

Die Testfälle im *VSR*-Projekt sind in folgende drei Prioritätsstufen eingeteilt:

| Prio | Bedeutung |
|---|---|
| 1 | Testfall muss durchgeführt werden |
| 2 | Testfall sollte durchgeführt werden |
| 3 | Testfall kann durchgeführt werden |

Aufbauend auf dieser Priorisierung wurden im Testkonzept folgende testfallbasierte Testendekriterien für den *VSR*-Systemtest festgelegt:

- Alle Testfälle mit Priorität 1 sind fehlerfrei gelaufen,
- und mindestens 60 % aller Testfälle der Priorität 2 sind gelaufen.

*Produktfreigabe*

Falls die gesetzten Testendekriterien erfüllt sind, entscheidet das Projektmanagement (beraten durch den Testmanager), ob das entsprechende Testobjekt freizugeben und auszuliefern ist. Im Komponenten- und Integrationstest bedeutet »Auslieferung« die Übergabe an die jeweils nachfolgende Teststufe. Der Systemtest gibt die Software zur Auslieferung an den Kunden frei. Der Abnahmetest des Kunden schließlich gibt den Betrieb des Systems in der eigentlichen Einsatzumgebung frei.

Die Freigabe ist nicht mit »Fehlerfreiheit« gleichzusetzen. Das Produkt wird einige unentdeckte Defekte enthalten und auch einige entdeckte, die als »nicht Freigabe verhindernd« eingestuft und deshalb nicht behoben wurden. Letztere sind in der Fehlerdatenbank vermerkt und werden im Zuge der Softwarewartung (s. Kap. 3.6.1) später korrigiert.

### 6.3.3 Testzyklussteuerung

*Auf Planabweichungen reagieren*

Falls Verzögerungen gegenüber der Projekt- oder Testplanung entstanden sind, muss der Testmanager geeignete Gegenmaßnahmen einleiten. So kann es notwendig sein, zusätzliche Testressourcen (Personal, Arbeitsplätze, Werkzeuge) anzufordern und einzusetzen, um einen Rückstand gegenüber dem Testplan in den verbleibenden Zyklen aufzuholen.

Stehen keine zusätzlichen Ressourcen zur Verfügung, muss der Testplan selbst angepasst werden. Mit niedriger Priorität eingestufte Testfälle werden gestrichen. Eine weitere Möglichkeit ist, Testfälle, die in verschiedenen Varianten geplant sind, nur in einer Variante durchzu-

führen (z. B. werden Tests nur unter einem Betriebssystem ausgeführt, statt wie geplant unter verschiedenen). Durch solche Anpassungen werden zwar einige interessante Tests nicht durchgeführt, aber die gesparten Ressourcen ermöglichen es, wenigstens die Testfälle hoher Priorität durchzuführen.

Je nachdem wie gravierend die aufgedeckten Mängel und Probleme sind, ist die Testdauer zu verlängern, weil zusätzliche Testzyklen notwendig sind. Denn nach jedem Korrekturzyklus muss die geänderte Software erneut getestet werden (s. Kap. 3.6.3). Dies kann unter Umständen bedeuten, dass die Markteinführung bzw. die Auslieferung des Softwareprodukts verschoben werden muss.

*Änderungen des Testplans klar kommunizieren*

Wichtig ist, dass der Testmanager jede solche Planänderung dokumentiert und kommuniziert. Denn eine Änderung am ursprünglichen Testplan bedeutet in aller Regel eine Erhöhung des Freigaberisikos. Es ist Aufgabe des Testmanagers, dieses Risiko den Projektverantwortlichen zu jedem Zeitpunkt klar und offen darzulegen.

## 6.4 Fehlermanagement

Damit die in den verschiedenen Teststufen gefundenen Mängel, Probleme und Fehler zuverlässig und schnell korrigiert werden können, bedarf es eines gut funktionierenden Verfahrens zur Übermittlung und Verwaltung entsprechender Fehlermeldungen. Dieses Verfahren beginnt nach Ende eines Testlaufs mit der Auswertung des Testprotokolls.

### 6.4.1 Testprotokoll

*Testprotokoll analysieren*

Nach jedem Testlauf, spätestens mit Ende eines Testzyklus, erfolgt eine Auswertung der angefallenen Testprotokolle. Die Protokolle werden auf Abweichungen zwischen erwartetem Sollverhalten und beobachtetem Istverhalten untersucht. Jedes signifikante, unerwartete Ereignis, das während des Testens aufgetreten ist, kann ein Symptom sein, das auf ein Fehlverhalten des Testobjekts hindeutet. Entsprechende Stellen im Testprotokoll werden analysiert. Die Tester vergewissern sich, ob tatsächlich eine Abweichung im Sollverhalten vorliegt oder ob die Ursache ein falsch entworfener Testfall, eine fehlerhafte Testautomatisierung oder eine falsche Testausführung war (auch Testern können Fehlhandlungen unterlaufen).

*Fehler dokumentieren*

Liegt das Problem im Testobjekt[36], wird über die Beobachtung eine Problemmeldung oder Fehlermeldung erstellt. Dies geschieht für jede im Testprotokoll enthaltene und vom Sollwert bzw. Sollverhalten ab-

weichende Beobachtung. Eventuell ist die Beobachtung ein Duplikat einer schon früher erfassten Beobachtung. Dann sollte geprüft werden, ob aus der zweiten Beobachtung zusätzliche Informationen zu entnehmen sind, die die Problemursache ggf. weiter eingrenzen können. Andernfalls sollte keine zweite Meldung erfolgen, um Mehrfachnennungen ein und desselben Fehlers zu vermeiden.

*Ursachenanalyse ist Aufgabe der Entwickler*

Die Tester müssen allerdings nicht die Ursache des gemeldeten Problems herausfinden. Dies ist Aufgabe der Entwickler.

### 6.4.2 Fehlermeldung

Üblicherweise wird im Projekt eine zentrale Fehlerdatenbank eingerichtet, in der alle Probleme, Mängel oder Fehler, die im Test (oder auch später im Betrieb) entdeckt werden, erfasst und verwaltet werden. Fehlermeldungen können von allen an der Entwicklung beteiligten Personen, aber auch von Kunden- und Anwenderseite eingebracht werden[37]. Sie können sich auf Probleme in den getesteten Programm(teil)en beziehen, aber auch auf Fehler oder Mängel in Spezifikationen, Benutzerhandbüchern oder anderen Dokumenten.

Das Fehlermeldeverfahren wird oft auch Problemmeldeverfahren genannt. Denn gemeldet werden alle offenen Probleme; aber nicht jedes Problem muss ein Fehler (der Entwickler) sein. Problemmeldung klingt auch weniger nach »Schuldzuweisung«. Es ist keine Einbahnstraße, sondern jeder Entwickler kann betreffende Meldungen kommentieren, etwa um Rückfragen an den Tester zu richten oder eine unberechtigte Meldung zurückzuweisen. Nimmt ein Entwickler aufgrund einer Fehlermeldung eine Korrektur am Testobjekt vor, so wird er dies ebenfalls in der Fehlerdatenbank dokumentieren, damit der zuständige Tester diese Korrektur im nächsten Testzyklus nachvollziehen und erneut testen kann.

Testmanager und Projektmanager können sich mittels Fehlerdatenbank zu jedem Zeitpunkt ein aktuelles und vollständiges Bild über Anzahl und Status der Fehler und über den Korrekturfortschritt verschaffen. Die Datenbank stellt hierzu entsprechende Auswertungen zur Verfügung.

---

36. Liegt das Problem auf Seiten der Tester, kann die Erstellung einer Fehlermeldung natürlich auch sinnvoll sein, z. B. wenn das Problem weitere Analysen erfordert. Adressat dieser Meldung sind dann nicht die Entwickler, sondern die Tester.
37. Die folgenden Erläuterungen gehen vereinfachend davon aus, dass nur Entwickler und Tester über die Fehlerdatenbank kommunizieren.

***Tipp: Fehlerdatenbank einsetzen***

Beim Aufbau eines geordneten Testprozesses für ein Projekt sollte als einer der allerersten Schritte ein diszipliniertes Fehlermanagement eingerichtet werden. Eine leistungsfähige Fehlerdatenbank, auf die alle Projektbeteiligten entsprechend ihrer Funktion Zugriff erhalten, ist hierzu unerlässlich.

*Einheitliches Meldungsschema*

Damit die Kommunikation reibungslos funktioniert und die Meldungen statistisch ausgewertet werden können, muss jede Meldung nach einem projektweit vorgegebenen einheitlichen Schema aufgebaut sein. Dieses Schema legt der Testmanager (z. B. im Testkonzept) fest.

Typischerweise beinhaltet eine Fehlermeldung neben der eigentlichen Problembeschreibung Informationen zur Identifikation der getesteten Software, zur Testumgebung, den Namen des Testers, die Fehlerklasse und Priorisierung sowie andere Informationen mit Relevanz für das Reproduzieren und Lokalisieren des potenziellen Fehlers. Tabelle 6-1 zeigt beispielhaft ein mögliches Schema.

***Tab. 6–1*** *Aufbau einer Fehlermeldung*

| | Attribut | Bedeutung |
|---|---|---|
| Identifikation | Nummer | laufende, eindeutige Meldungsnummer |
| | Testobjekt | Bezeichnung des Testobjekts |
| | Version | Identifikation der genauen Version des Testobjekts |
| | Plattform | Identifikation der HW-/SW-Plattform bzw. der Testumgebung, in der das Problem auftritt |
| | Entdecker | Identifikation des Testers (ggf. mit Teststufe), der das Problem meldet |
| | Entwickler | Name des für das Testobjekt verantwortlichen Entwicklers oder Teams |
| | Erfassung | Datum, ggf. Uhrzeit, an dem das Problem beobachtet wurde |
| Klassifikation | Status | Bearbeitungsfortschritt der Meldung (s. Kap. 6.4.4); möglichst mit Kommentar und Datum des Eintrags |
| | Klasse | Klassifizierung der Schwere des Problems (s. Kap. 6.4.3) |
| | Priorität | Klassifizierung der Dringlichkeit einer Korrektur (s. Kap. 6.4.3) |
| | Anforderung | Verweis auf die (Kunden-)Anforderung, die wegen der Fehlerwirkung nicht erfüllt oder verletzt ist |
| | Fehlerquelle | Soweit feststellbar, die Projektphase, in der die Fehlhandlung begangen wurde (Analyse, Design, Programmierung); nützlich zur Planung prozessverbessernder Maßnahmen |

| | Attribut | Bedeutung |
|---|---|---|
| Problembeschreibung | Testfall | Beschreibung des Testfalls (Name, Nummer) bzw. der Schritte, die zur Reproduktion der Fehlerwirkung führen |
| | Problem | Beschreibung der Fehlerwirkung; erwartete vs. tatsächliche Ergebnisse bzw. Verhalten |
| | Kommentar | Stellungnahmen der Betroffenen zum Meldungsinhalt |
| | Korrektur | Korrekturmaßnahmen des zuständigen Entwicklers |
| | Verweis | Querverweise auf andere zugehörige Meldungen |

Viele Meldungsattribute können weiter verfeinert und aufgeteilt werden (s. [IEEE 1044] und [DIN 66271]). Wird die Fehlerdatenbank z. B. im Abnahmetest oder im Produktsupport eingesetzt, müssen zusätzlich Kundendaten miterfasst werden. Der Testmanager muss ein für sein Projekt geeignetes Schema festlegen.

*Alle für Reproduktion und Korrektur relevanten Informationen erfassen*

Wichtig dabei ist, dass alle Informationen erfasst werden, die für das Reproduzieren und Lokalisieren des potenziellen Fehlers notwendig sind, sowie Informationen, die eine Auswertung der Produktqualität und des Korrekturfortschritts ermöglichen.

Unabhängig vom vereinbarten Schema gilt: Eine Meldung muss so abgefasst sein, dass der zuständige Entwickler mit minimalem Aufwand das Problem nachvollziehen und dessen Ursache finden kann.

### 6.4.3 Fehlerklassifikation

Ein wichtiges Kriterium zur Beurteilung eines gemeldeten Problems ist sicherlich dessen »Schwere«, d. h. nach [DIN 66271] der Grad der Beeinträchtigung des Produkteinsatzes. Schließlich ist es ein gravierender Unterschied, ob 100 offene Fehler in der Datenbank stehen, die Systemabstürze darstellen oder lediglich Schönheitsfehler im Maskenlayout dokumentieren. Eine Klassifizierung der Fehlerschwere kann z. B. wie in Tabelle 6-2 gezeigt aussehen.

***Tab. 6–2***
*Fehlerschwere*

| Klasse | Bedeutung |
|---|---|
| 1 | Systemabsturz mit ggf. Datenverlust; das Testobjekt ist in dieser Form nicht einsetzbar. |
| 2 | Wesentliche Funktion ist fehlerhaft; Anforderung nicht beachtet oder falsch umgesetzt; das Testobjekt ist nur mit großen Einschränkungen einsetzbar. |
| 3 | Funktionale Abweichung bzw. Einschränkung (»normaler« Fehler); Anforderung fehlerhaft oder nur teilweise umgesetzt; System kann mit Einschränkungen genutzt werden. |

| Klasse | Bedeutung |
|---|---|
| 4 | Geringfügige Abweichung; System kann ohne Einschränkung genutzt werden. |
| 5 | Schönheitsfehler (z. B. Rechtschreibfehler oder Mangel im Maskenlayout); System kann ohne Einschränkung genutzt werden. |

Die »Schwere« eines Fehlers sollte aus Sicht des Anwenders bzw. späteren Nutzers des Testobjekts vergeben werden. Aus dieser Einordnung muss jedoch nicht unmittelbar folgen, wie schnell das jeweilige Problem zu beheben ist. Hier gehen zusätzlich Anforderungen des Produkt- oder Projektmanagements ein (z. B. Korrekturaufwand), aber auch Anforderungen aus Sicht der weiteren Testdurchführung (blockierte Tests). Die Frage, wie dringlich ein Defekt zu korrigieren ist, wird deshalb über ein weiteres Attribut, die »Fehlerpriorität«, gesteuert. Ein mögliches Schema zeigt Tabelle 6-3.

***Tab. 6–3***
*Fehlerpriorität*

| Priorität | Bedeutung |
|---|---|
| 1 – PATCH | Der Arbeitsablauf beim Anwender ist blockiert oder die laufenden Tests können nicht fortgesetzt werden. Das Problem muss unmittelbar, ggf. provisorisch, behoben werden; ein →Patch ist zu erstellen. |
| 2 – UPDATE | Die Fehlerkorrektur erfolgt mit der nächsten regulären Produktversion oder der nächsten Testobjektlieferung. |
| 3 – GELEGENTLICH | Die Fehlerkorrektur erfolgt, sobald die betroffenen Systemteile ohnehin überarbeitet werden. |
| 4 – OFFEN | Korrekturplanung ist noch zu treffen. |

*Meldungsauswertung zur Überwachung des Testprozesses*

Die Auswertung von Schwere und Priorität der erfassten Fehler erlaubt dem Testmanager, Aussagen über die Produktstabilität bzw. über die Lieferfähigkeit des Testobjekts zu treffen. Neben der reinen Standortbestimmung und Klärung der Fragen: »Wie viele Defekte wurden gefunden?«, »Wie viele davon sind behoben?«, »Wie viele sind noch offen?«, sind Trendanalysen wichtig, d. h. Prognosen auf Basis einer Auswertung über den zeitlichen Verlauf der Meldungseingänge zu erstellen. Die wichtigste Frage in diesem Zusammenhang ist: »Nehmen die Probleme weiter zu oder ist eine Abflachung der Trendkurve zu erkennen?«

*Meldungsauswertung zur Verbesserung des Testprozesses*

Daten aus Fehlermeldungen können auch zur Verbesserung des Testprozesses selbst herangezogen werden. Zum Beispiel kann ein Vergleich der Daten für verschiedene Testobjekte zeigen, welche Testobjekte unterdurchschnittlich viele Defekte aufweisen. Es könnte sich

herausstellen, dass dort noch Testfälle fehlen oder besonders sorgfältig programmiert wurde.

### 6.4.4 Fehlerstatus

Das Testmanagement muss nicht nur sicherstellen, dass Fehler ordentlich erfasst und verwaltet werden, es muss in Kooperation mit dem Projektmanagement dafür sorgen, dass Fehlerzustände zügig korrigiert werden und dass diese Korrekturen mit den neuen Versionen des Testobjekts verfügbar sind.

Hierzu ist eine kontinuierliche Verfolgung des Fehleranalyse- und Korrekturprozesses über alle Stadien notwendig. Diese Verfolgung geschieht anhand des Fehlerstatus. Dabei durchläuft jede Fehlermeldung (vgl. Tabelle 6-1) eine Reihe festgelegter Stati, die die erstmalige Erfassung bis zur erfolgreichen Fehlerkorrektur beinhalten. Ein beispielhaftes Schema hierfür zeigt Tabelle 6-4.

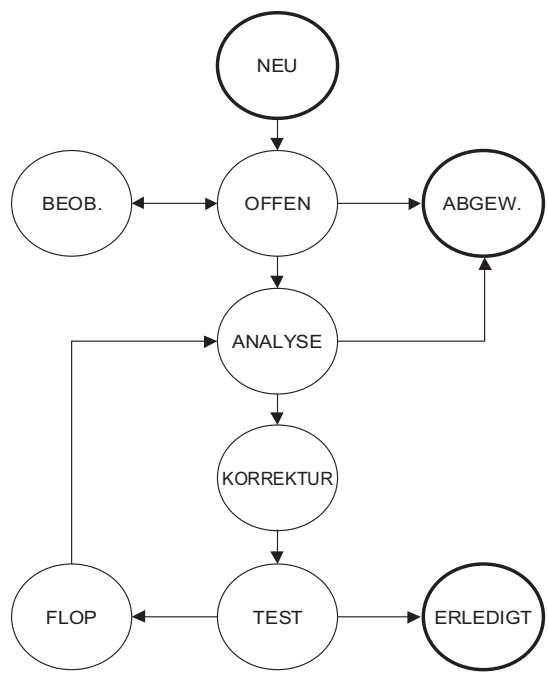
***Tab. 6–4***
*Fehlerstatusschema*

| Status (gesetzt durch) | Bedeutung |
|---|---|
| Neu (Tester) | Neue Meldung wurde erfasst. Der Verfasser hat eine aus seiner Sicht sinnvolle Beschreibung und Klassifizierung eingetragen. |
| Offen (Testmanager) | Neue Meldungen werden regelmäßig vom Testmanager gesichtet. Verständlichkeit und vollständige Vergabe der identifizierenden Attribute wird geprüft. Klassifizierende Attribute werden ggf. angepasst, so dass eine projektweit einheitliche Bewertung gegeben ist. Dubletten oder offensichtlich sinnlose Meldungen werden bereinigt/ zurückgewiesen.<br>Die Meldung wird einem zuständigen Entwickler zugewiesen und auf »OFFEN« gesetzt. |
| Abgewiesen (Testmanager) | Meldung wird als unberechtigt abgewiesen (kein Defekt im Testobjekt, Änderungswunsch, der nicht berücksichtigt wird). |
| Analyse (Entwickler) | Der zuständige Entwickler setzt die Meldung auf diesen Status, sobald er die Meldung bearbeitet. Das Ergebnis der Problemanalyse (Fehlerursache, Lösungsmöglichkeiten, geschätzter Korrekturaufwand u. Ä.) wird in Kommentaren dokumentiert. |
| Beobachtung (Entwickler) | Das geschilderte Problem kann weder nachvollzogen noch ausgeschlossen werden. Die Meldung bleibt unerledigt, bis weitere Informationen/Erkenntnisse vorliegen. |
| Korrektur (Projektmanager) | Aufgrund der Analyse entscheidet der Projektmanager, dass die Korrektur erfolgen soll, und setzt dazu den Status auf »KORREKTUR«. Der zuständige Entwickler führt die Korrekturen durch. Die Art der Korrektur wird in Kommentaren dokumentiert. |

| Status (gesetzt durch) | Bedeutung |
|---|---|
| Test (Entwickler) | Der zuständige Entwickler setzt die Meldung auf diesen Status, sobald das Problem aus seiner Sicht behoben ist. Die Softwareversion, in der die Korrektur verfügbar ist, wird angegeben. |
| Erledigt (Tester) | Problemlösungen, also Meldungen im Status »TEST«, werden vom zuständigen Tester im nächstmöglichen Testzyklus verifiziert. Dazu wird mindestens der fehleraufdeckende Test wiederholt.<br>Ergibt dieser Fehlernachtest, dass die Fehlerbeseitigung erfolgreich ist, beendet der Tester die Meldungshistorie im Endzustand »ERLEDIGT«. |
| Flop (Tester) | Ergibt der Fehlernachtest, dass die Fehlerbeseitigung erfolglos oder ungenügend war, wird »FLOP« gesetzt. Eine erneute Analyse ist notwendig. |

Abbildung 6-2 veranschaulicht diesen Ablauf.

***Abb. 6–2***
*Fehlerstatus Modell*




*Auf »ERLEDIGT« darf nur der Tester setzen*

Ein entscheidender Punkt, der oft übersehen wird, ist, dass »ERLEDIGT« nicht vom Entwickler, sondern ausschließlich vom Tester gesetzt werden darf! Und zwar erst, nachdem eine Testwiederholung (Fehlernachtest) nachgewiesen hat, dass das in der Fehlermeldung beschrie-

bene Problem nicht mehr auftritt. Falls nun nach der Korrektur als Seiteneffekt neue Fehlerwirkungen auftreten, werden diese in neuen Meldungen erfasst und behandelt.

***Beispiel: erweiterte Testendekriterien für den VSR-Systemtest***

Die Testendekriterien für den *VSR*-Systemtest sollen nicht nur den Testfortschritt widerspiegeln, sondern auch die erreichte Produktqualität. Der Testmanager erweitert die Testendekriterien deshalb wie folgt um fehler-basierte Metriken:

- Alle Fehler der Klasse 1 sind »ERLEDIGT«
- Keine nicht »ERLEDIGT«-Fehler mit Priorität »PATCH«
- Die Anzahl »NEU«-Meldungen pro Testwoche stagniert oder sinkt

Das oben beschriebene Schema kann in vielen Projekten eingesetzt werden. Die im Projekt vorhandenen oder notwendigen Entscheidungsprozesse müssen jedoch abgebildet werden. Hierzu sind geeignete Anpassungen des Modells vorzusehen. Während im oben erklärten Grundmodell alle Entscheidungen in der Hand von Einzelpersonen liegen, werden diese Prozesse in größeren Projekten von Entscheidungsgremien getroffen. Entsprechend komplexer werden die Entscheidungsprozesse, da Vertreter vieler Interessengruppen gehört werden müssen.

*change control board*

Die von den Entwicklern auszuführenden Änderungen sind in vielen Fällen keine »Fehlerkorrekturen«, sondern echte (funktionale) Erweiterungen. Da die Grenze zwischen »Fehlermeldung« und »Erweiterungswunsch« und die Beurteilung als »berechtigt« oder »unberechtigt« oft Ermessenssache sind, wird im Fehlermanagement eine Instanz benötigt, die Änderungsanforderungen genehmigt oder ablehnt. Diese Instanz heißt Änderungskontrollausschuss (*change control board*) und wird üblicherweise von Vertretern folgender Interessengruppen gebildet: Produktmanagement, Projektmanagement, Testmanagement, Kundenvertreter.

## 6.5 Anforderungen an das Konfigurationsmanagement

Ein Softwaresystem besteht aus einer Vielzahl von Einzelbausteinen, die zueinander passen müssen, damit das System als Ganzes funktioniert. Von jedem dieser Bausteine entstehen im Laufe der Entwicklung des Systems neue, korrigierte oder verbesserte Versionen oder Varianten. Da an diesem Prozess mehrere Entwickler und Tester parallel be-

teiligt sind, ist es alles andere als einfach, den Überblick zu behalten, welche Bausteine aktuell sind und zusammengehören.

*Typische Symptome von unzureichendem KM*

Wird in einem Projekt das Konfigurationsmanagement (KM) unzureichend erledigt, können typische Symptome beobachtet werden:

- Entwickler überschreiben gegenseitig ihre am Code vorgenommenen Modifikationen, da der gleichzeitige Zugriff auf gemeinsame Codedateien nicht verhindert wird.
- Die Integrationsarbeiten sind behindert,
  - weil unklar ist, welche Versionen des Codes einer bestimmten Komponente im Entwicklungsteam existieren und welche davon aktuell sind,
  - weil unklar ist, welche Versionen verschiedener Komponenten zusammengehören und zu einem größeren Teilsystem integriert werden können oder
  - weil Compiler und andere Entwicklungswerkzeuge in unterschiedlichen Versionen eingesetzt werden.
- Fehleranalyse, Fehlerkorrektur und Regressionstest sind erschwert,
  - weil unbekannt ist, wo und warum der Code einer Komponente gegenüber der Vorversion geändert wurde,
  - weil unbekannt ist, aus welchen Codedateien ein bestimmtes integriertes Teilsystem (Objektcode) hervorgegangen ist.
- Tests und Testauswertung sind behindert, weil unklar ist,
  - welche Testfälle zu welchem Versionsstand eines Testobjekts gehören,
  - welche Testergebnisse ein Testlauf an einem Testobjekt einer bestimmten Version erbracht hat.

Unzureichendes Konfigurationsmanagement führt also zu einer Vielzahl möglicher Störungen, die den Entwicklungs- aber auch den Testprozess behindern. Ist zum Beispiel in einer Teststufe unklar, ob die untersuchten Testobjekte alle in der aktuellsten Version vorliegen, dann verlieren die Tests schnell jegliche Aussagekraft. Ein guter Testprozess ist ohne verlässliches Konfigurationsmanagement nicht realisierbar.

*Anforderungen an KM*

Folgende Anforderungen sollten aus Sicht des Tests erfüllt werden:

- **Versionenverwaltung**: Katalogisieren, Speichern und Wiederabrufen von unterschiedlichen Versionen eines →Konfigurationsobjekts (z. B. Version 1.0 und 1.1 einer Datei). Hierzu gehört auch das Mitführen von Kommentaren, aus denen der jeweilige Änderungsgrund hervorgeht.
- **Konfigurationsverwaltung**: Bestimmung und Verwaltung aller Dateien (Konfigurationsobjekte) in der jeweils passenden Version, die

zusammen ein Teilsystem bilden (→Konfiguration). Voraussetzung hierfür ist eine Versionenverwaltung.

- **Statusverfolgung von Fehlern und Änderungen**: Aufzeichnung von Problemberichten und Änderungsanforderungen und die Möglichkeit, deren Umsetzung an den Konfigurationsobjekten nachzuvollziehen.
- Um die Wirksamkeit des Konfigurationsmanagements zu prüfen, ist es sinnvoll, **Konfigurationsaudits** durchzuführen. In einem solchen →Audit kann z. B. geprüft werden, ob alle Softwarebausteine vom Konfigurationsmanagement erfasst werden, ob die Konfigurationen korrekt identifiziert werden können usw.

***Beispiel: KM im Projekt VSR***

Die im VSR-Projekt entwickelte Software liegt in unterschiedlichen Sprachversionen (z. B. in Deutsch, Englisch, Französisch) vor und muss auf verschiedenen Hardware- und Softwareplattformen ablauffähig sein. Einzelne Komponenten müssen zu bestimmten Versionen externer Software passen (z. B. zur aktuellen Kommunikationssoftware des Hostrechners). Des Weiteren müssen periodisch Datenbestände unterschiedlichster Quellen importiert werden (z. B. Produktkataloge, Preislisten, Vertragsdaten), deren Inhalte und Formate sich während der Einsatzzeit des Systems ändern. Das *VSR*-Konfigurationsmanagement muss sicherstellen, dass in der Entwicklung und beim Testen stets mit konsistenten, gültigen Produktkonfigurationen gearbeitet wird. Ähnliches gilt für den Produktivbetrieb beim Anwender.

Je nach Projekt sind andere Verfahren und Werkzeuge angemessen oder notwendig, um ein Konfigurationsmanagement zu realisieren, das oben genannte Anforderungen erfüllt. Deshalb ist in einem Konfigurationsmanagementplan das auf die Projektsituation zugeschnittene Vorgehen festzulegen. Auch für Konfigurationsmanagement und zugehörige Pläne gibt es Standards: [IEEE 1042] und [IEEE 828].

## 6.6 Relevante Normen und Standards

Auch in der Softwareentwicklung existieren mittlerweile eine Vielzahl von Normen und Standards, die Randbedingungen vorgeben und den »Stand der Technik« definieren. Wie die in diesem Buch zitierten Normen belegen, gilt dies in besonderem Maße für den Themenbereich Softwarequalitätsmanagement und Softwaretest.

Zu den Aufgaben eines Qualitäts- oder Testmanagers gehört es in diesem Zusammenhang festzustellen, welche Normen, Standards oder evtl. gesetzliche Richtlinien für das zu testende Produkt (Produktnormen) oder auch für das Projekt (Prozessnormen) relevant sind, und de-

ren Einhaltung sicherzustellen. Als mögliche Quellen kommen hierbei in Frage:

- **Firmenstandards**: Firmeninterne Richtlinien und Verfahrensanweisungen (des Herstellers und ggf. des Kunden), wie z. B. Qualitätsmanagementhandbuch, Rahmentestplan, Programmierrichtlinien.
- **Best Practises:** Nicht standardisierte, aber fachlich bewährte Vorgehensweisen und Verfahren, die den Stand der Technik in einem Anwendungsgebiet darstellen.
- **Qualitätsmanagementstandards:** Branchenübergreifende Standards, die Mindestanforderungen an Prozesse spezifizieren, ohne konkrete Anforderungen bezüglich der Umsetzung zu formulieren. Bekanntes Beispiel ist [ISO 9000], die z. B. fordert, dass im Produktionsprozess (also auch im Spezialfall Softwareentwicklungsprozess) geeignete (Zwischen-)Prüfungen vorzusehen sind, ohne anzugeben, wann und wie diese zu erledigen sind.
- **Branchenstandards:** Branchenspezifische Standards (beispielsweise [DIN 60601-1-4] für Medizinprodukte), die für eine bestimmte Produktkategorie oder ein Einsatzgebiet festlegen, in welchem Mindestumfang Tests durchzuführen oder nachzuweisen sind.
- **Softwareteststandards:** Prozessstandards, die produktunabhängig festlegen, wie Softwaretests fachgerecht durchzuführen sind. Beispiele sind die im Buch zitierten Normen [BS 7925-2], [IEEE 829], [IEEE 1028].

Die für den Softwaretest wichtigen und relevanten Normen und Standards werden in diesem Buch behandelt. Das Testkonzept nach IEEE 829 wird in Anhang A im Detail erläutert. Eine Orientierung an solchen Standards macht auch dort Sinn, wo deren Einhaltung nicht verpflichtend vorgeschrieben ist. Spätestens bei Rechtsstreitigkeiten ist die Entwicklung nach dem »Stand der Technik« nachzuweisen, was die Einhaltung von Normen einschließt.

## 6.7 Zusammenfassung

- Entwicklungs- und Testaktivitäten sollen organisatorisch getrennt sein. Je klarer diese Trennung, umso wirksamer kann getestet werden.
- Je nach Aufgabenstellung innerhalb des Testprozesses werden Mitarbeiter mit rollenspezifischen Testkenntnissen benötigt. Neben fachlichen Fähigkeiten ist auch soziale Kompetenz gefragt.

- Zu den Aufgaben des Testmanagers gehören Planung, Überwachung und Steuerung der einzelnen Testzyklen.
- Fehlermanagement und Konfigurationsmanagement bilden die Basis für einen effizienten Testprozess.
- Fehlermeldungen müssen nach einem projektweit einheitlichen Schema erfasst und über alle Stadien des Fehleranalyse- und Korrekturprozesses verfolgt werden.
- Normen und Standards enthalten Vorgaben und Empfehlungen zur fachgerechten Durchführung von Softwaretests. Eine Orientierung an solchen Standards macht auch dort Sinn, wo deren Einhaltung nicht verpflichtend vorgeschrieben ist.

# 7 Testwerkzeuge

*Dieses Kapitel gibt einen Überblick über die verschiedenen Testwerkzeuge, die den Tester bei seiner Arbeit unterstützen. Diskutiert werden die Voraussetzungen für den Einsatz solcher Werkzeuge und das Vorgehen bei Werkzeugauswahl und Einführung.*

## 7.1 Typen von Testwerkzeugen

Es gibt eine Vielzahl von Werkzeugen (Tools) zur Unterstützung oder Automatisierung von Testarbeiten. Angelehnt an den Begriff CASE-Tools (*Computer Aided Software Engineering*) wird hier auch von CAST-Tools (*Computer Aided Software Testing*) gesprochen. Je nachdem welche Phase im Testprozess (vgl. Kap. 2.2) unterstützt wird, werden verschiedene Werkzeugtypen unterschieden. Nur in seltensten Fällen wird die gesamte Palette an Werkzeugen in einem Testprojekt tatsächlich eingesetzt. Die verfügbaren Werkzeugtypen sollten aber bekannt sein, um entscheiden zu können, ob und wann ein Werkzeug im Projekt nutzbringend eingesetzt werden kann.

*Werkzeugliste auf WWW-Seite [URL: dPunkt]*

Eine Liste marktgängiger Werkzeuge mit Bezugsquellen und einigen Demoversionen zum Download bietet die zum Buch gehörende Website [URL: dPunkt]. Welche Funktionen Werkzeuge prinzipiell bieten, erläutern die folgenden Abschnitte.

### 7.1.1 Werkzeuge für Testplanung und Testmanagement

Werkzeuge zur Testplanung bieten Mechanismen zur bequemen Erfassung, Katalogisierung und Verwaltung der Testfälle und zu deren Priorisierung. Sie helfen dem Testmanager, den Überblick über hunderte oder tausende von Testfällen zu behalten.

Fortgeschrittene Tools unterstützen darüber hinaus anforderungsbasiertes Testen. Dazu bieten sie die Möglichkeit, Systemanforderungen (Requirements) zu erfassen (oder aus entsprechenden Requirement-Managementwerkzeugen zu importieren) und mit denjenigen Tests zu verknüpfen, die die entsprechende Anforderung prüfen. Es können verschiedene Konsistenzprüfungen durchgeführt werden, z. B. ein Check, ob zu jeder Anforderung mindestens ein Testfall eingeplant ist.

Meist bieten Testplanungswerkzeuge auch die Möglichkeit, den Status der Testfälle zu überwachen, also festzuhalten und auszuwerten wann, wie oft und mit welchem Resultat (OK oder NICHT-OK) ein Testfall ausgeführt wurde. Einige Werkzeuge unterstützen darüber hinaus den Aspekt des Projektmanagements innerhalb des Testens (z. B. Ressourcen- und Zeitplanung der Tests).

Nahezu unverzichtbar für den Testmanager ist ein Werkzeug zur Verwaltung der Problem- bzw. Fehlermeldungen. Wie in Kapitel 6 beschrieben, dienen Fehlermanagementwerkzeuge zur Erfassung, Verwaltung, Verteilung und statistischen Auswertung von Fehlermeldungen. Komfortable Vertreter dieser Klasse bieten individuell parametrisierbare Fehlerstatusmodelle. Von der Fehlerentdeckung über die Korrektur bis zum Regressionstest kann damit der gesamte Arbeitsablauf vorgegeben werden. Jeder Projektmitarbeiter wird entsprechend seiner Rolle im Team durch diesen Ablauf geführt.

*Generierung der Testdokumentation*

Sehr oft sind Fehlermanagement- und Testplanungswerkzeuge miteinander gekoppelt. Es kann beispielsweise ein Plan für den Fehlernachtest erzeugt werden, d. h. eine Liste aller Tests, die notwendig sind, um zu verifizieren, welche Defekte in einer bestimmten Testobjektversion erfolgreich korrigiert wurden. Sowohl Testmanagement- als auch Fehlermanagementwerkzeuge stellen ausführliche Analyse- und Berichtsfunktionen zur Verfügung, bis hin zur Möglichkeit, aus den verwalteten Daten die komplette Testdokumentation zu generieren (Testplan, Testspezifikation, Testbericht).

### 7.1.2 Werkzeuge zur Testspezifikation

Damit ein Testfall reproduzierbar durchgeführt werden kann, müssen neben Vor- und Nachbedingungen die Testdaten, mit denen das Testobjekt versorgt werden soll, und die zugehörigen erwartete(n) Sollreaktion(en) des Testobjekts festgelegt werden.

*Testgeneratoren*

Bei der Erzeugung von Testdaten können den Testdesigner so genannte Test(daten)generatoren unterstützen. Nach [Fewster 99] lassen sich verschiedene Ansätze unterscheiden, abhängig davon aus welcher Testbasis die Testdaten abgeleitet werden:

- **Datenbankbasierte Testdatengeneratoren** verarbeiten Datenbankschemata und sind in der Lage, daraus Testdatenbestände zu generieren oder aus Datenbankinhalten Testdaten geeignet herauszufiltern. Analoges gilt für die Generierung von Testdaten aus Dateien unterschiedlichster Datenformate.
- **Codebasierte Testdatengeneratoren** generieren Testdaten, indem sie den Code des Testobjekts analysieren. Nachteile und Grenzen dabei

sind, dass keine Sollwerte generiert werden können, denn dazu wird ein Testorakel benötigt (siehe Kap. 2.3). Auch wird (wie bei allen White-box-Methoden) nur Code berücksichtigt, der existiert. Fehlerzustände aufgrund vergessener Programmanweisungen bleiben unentdeckt. Grundsätzlich ist der Code als Testbasis zum Test desselben eine sehr schwache Grundlage.

- **Schnittstellenbasierte Testdatengeneratoren** analysieren die Testobjektschnittstelle, erkennen die Definitionsbereiche der Schnittstellenparameter und leiten z. B. mittels Äquivalenzklassen- und Grenzwertanalyse daraus Testdaten ab. Tools existieren für verschiedenste Schnittstellenarten, von Programmierschnittstellen (*Application Programming Interface*) bis zur Analyse grafischer Benutzungsoberflächen. Das Werkzeug erkennt, welche Datenfelder in einer Maske vorhanden sind (z. B. numerisches Feld, Datumsfeld) und generiert Testdaten, die die jeweiligen Wertebereiche abdecken (z. B. per Grenzwertanalyse). Auch hier besteht das Problem, dass keine Sollwerte generiert werden können. Die Werkzeuge sind aber gut zur automatischen Erzeugung von Negativtests (vgl. Robustheitstest) geeignet, da hier keine konkreten Sollwerte interessieren, sondern in der Regel nur die Tatsache, ob das Testobjekt mit einer Fehlermeldung reagiert.
- **Spezifikationsbasierte Testdatengeneratoren** leiten Testdaten und zugehörige Sollwerte aus einer Spezifikation ab. Voraussetzung ist natürlich, dass die Spezifikation in einer formalen Notation vorliegt. Einige Generatoren nutzen formale Modelle des Testobjekts, wie sie mit CASE-Tools erstellt werden können.

*Kreativität des Testdesigners nicht ersetzbar*

Von den Werkzeugen sind keine Wunder zu erwarten. Tests zu spezifizieren, ist eine sehr anspruchsvolle Tätigkeit, die ein umfassendes Verständnis des Testobjekts voraussetzt, aber auch Kreativität und Intuition. Ein Testdatengenerator kann gewisse Regeln anwenden (z. B. Grenzwertanalyse), um Tests systematisch zu erzeugen, beurteilen, welche der erzeugten Testfälle gut oder schlecht, wichtig oder irrelevant sind, kann er allerdings nicht. Diese kreativ-analytische Arbeit muss nach wie vor der Testdesigner leisten. Auch müssen meist die zugehörigen Sollreaktionen des Testobjekts manuell ergänzt werden.

### 7.1.3 Werkzeuge zur Testdurchführung

*Tools entlasten von mechanischen Testarbeiten*

Wird allgemein von Testwerkzeugen gesprochen, sind oft im engeren Sinne Werkzeuge zur Automatisierung der Testdurchführung gemeint. Sie entlasten den Testdurchführenden von den zum Testablauf nötigen mechanischen Arbeiten. Die Werkzeuge versorgen das Testobjekt mit

Testdaten, zeichnen die Reaktion des Testobjekts auf und protokollieren den Testlauf. Da solche Tools hierzu an die Testschnittstelle des jeweiligen Testobjekts angeschlossen werden müssen, unterscheiden sie sich sehr stark nach der Teststufe (Komponenten-, Integrations-, Systemtest), für die sie eingesetzt werden.

*Debugger*

Debugger ermöglichen es, ein Programm oder Programmstück zeilenweise abzuarbeiten, die Abarbeitung an jeder Programmanweisung anzuhalten, Variablen zu setzen und auszulesen. Debugger sind in erster Linie Analysewerkzeuge des Entwicklers, um Ausfälle zu reproduzieren und Fehlerursachen zu analysieren. Aber auch beim Test sind Debugger manchmal sinnvoll, um bestimmte Testsituationen zu erzwingen, die sonst nicht oder nur mit unverhältnismäßig hohem Aufwand herzustellen sind. Sie können auch als Testschnittstelle für Tests auf Komponentenebene dienen.

*Testtreiber*

Als Testtreiber werden kommerzielle Produkte oder auch individuell gefertigte Werkzeuge bezeichnet, die Mechanismen bieten, um Testobjekte über deren Programmierschnittstelle anzusprechen. Also Testobjekte, die keine Benutzerschnittstelle aufweisen und deshalb einem manuellen Test nicht direkt zugänglich sind. Benötigt werden Testtreiber hauptsächlich im Komponenten- und Integrationstest oder für spezielle Aufgaben im Systemtest. Erhältlich sind auch generische Testtreiber oder Testrahmengeneratoren. Diese analysieren die Programmierschnittstelle des Testobjekts und erzeugen einen Testrahmen. Testrahmengeneratoren sind daher auf spezielle Programmiersprachen oder Entwicklungsumgebungen zugeschnitten. Der generierte Testrahmen enthält die benötigten Initialisierungen und Aufrufsequenzen, um das Testobjekt ansteuern zu können. Eventuell erzeugt er auch Dummies oder Stubs. Hinzu kommen Funktionen zur Abfrage bzw. Erfassung der Sollreaktionen sowie zur Protokollierung des Testablaufs. Testrahmengeneratoren verringern damit ganz erheblich den Aufwand zur Programmierung einer Testumgebung.

*Simulatoren*

Wenn für den Systemtest ein Test in der Produktivumgebung oder mit dem Zielsystem nicht möglich oder zu aufwändig ist (z. B. Robustheitstest einer Flugzeugsteuerung im Flugzeug selbst), können Simulatoren eingesetzt werden. Der Simulator bildet hierzu die eigentliche Einsatzumgebung möglichst umfassend und realitätsnah nach.

*Testroboter*

Dient als Testschnittstelle direkt die Bedienoberfläche eines Softwaresystems, können so genannte →Testroboter eingesetzt werden. Diese Werkzeuge werden auch *Capture-and-Replay-* bzw. *Capture-and-Playback-Tools* genannt, was ihre Funktionsweise beinahe schon erklärt. Testroboter arbeiten nach dem Prinzip des Videorekorders.

***Exkurs zur Funktion von Capture-and-Replay-Tools***

Das Capture-and-Replay-Tool[38] zeichnet während einer Testsitzung alle manuell durchgeführten Bedienschritte (Tastatureingaben, Mausklicks) auf, die der Tester am Testobjekt ausführt. Dabei werden nicht nur die x/y-Koordinaten der Mausklicks aufgezeichnet, sondern auch die an der grafischen Bedienoberfläche (GUI) ausgelösten Operationen (z. B. pressButton(»Start«)) sowie die zur Wiedererkennung des angeklickten Objekts benötigten Objekteigenschaften (Objektname, Objekttyp, Farbe, Beschriftung, x/y-Position u. a.). Diese Bedienschritte speichert das Werkzeug als →Testskript.

*Aufzeichnung (capture)*

Um feststellen zu können, ob sich das zu testende Programm korrekt verhält, kann der Tester (während der Testaufzeichnung oder bei der Skriptnachbearbeitung) Soll/Istvergleiche in das Testskript aufnehmen. Auf diese Weise lassen sich layoutrelevante Eigenschaften von Bedienobjekten (z. B. Farbe, Position, Größe eines Buttons) verifizieren, aber auch funktionelle Eigenschaften des Testobjekts (Wert eines Maskenfeldes, Inhalt einer Messagebox etc.).

*Soll/Istvergleiche*

Derart aufgezeichnete Testskripte sind wieder abspielbar und damit im Prinzip beliebig wiederholbar. Unterscheiden sich beim Erreichen eines Soll/Istvergleichs beide Werte, »schlägt der Test fehl«, und der Testroboter schreibt eine entsprechende Meldung ins Testprotokoll. Aufgrund dieser Fähigkeit, automatisch Soll- und Istwerte zu vergleichen, eignen sich Capture-and-Replay-Tools hervorragend zur Automatisierung von Regressionstests.

*Wiedergabe (Replay)*

Eine Schwierigkeit besteht allerdings: Ändert sich im Zuge einer Programmkorrektur oder Programmerweiterung die Bedienoberfläche des Testobjekts zwischen zwei Testläufen, so kann es passieren, dass das ursprünglich aufgezeichnete Skript nicht mehr zur neuen Bedienoberfläche »passt«. Das Skript bleibt dann unter Umständen stehen, der automatisierte Testlauf bricht ab. Capture-and-Replay-Tools bieten eine gewisse Robustheit gegenüber solchen Änderungen der GUI, da sie GUI-Objekte nicht an deren x/y-Position erkennen, sondern dazu andere Objekteigenschaften ausnutzen. Deshalb werden z. B. Buttons beim Abspielen des Testskripts auch dann wiedergefunden, wenn sie verschoben wurden.

*Problem: Änderung der GUI*

Die Testskripte werden üblicherweise in Skriptsprachen aufgezeichnet. Die Skriptsprachen sind an gängige Programmiersprachen angelehnt (Basic-, C- oder Java-ähnlich) und bieten die von den Programmiersprachen bekannten üblichen Sprachmittel (Fallunterscheidungen, Schleifen, Prozeduren usw.). Damit ist es möglich, auch komplexe Testabläufe zu implementieren oder aufgezeichnete Skripte nachzubearbeiten. Dies ist in der Praxis fast immer notwendig, denn ein Capture-Lauf liefert nur selten auf Anhieb ein regressionstestfähiges Testskript. Folgendes Beispiel illustriert dies:

*Testprogrammierung*

---

***Beispiel: Automatisierter Test der VSR-ContractBase***

Im Test des *VSR*-Teilsystems zur Vertragsverwaltung soll überprüft werden, ob Kaufverträge korrekt gespeichert und wiedergefunden werden. Zur Testautomatisierung zeichnet der Tester folgende Bediensequenz auf:

```
Maske "Vertragsdaten" aufrufen;
```

38. Siehe hierzu u. a. die ausführliche Beschreibung in [URL: imbus 99] sowie weiterführende Darstellungen in [Dustin 99] und [Fewster 99].

```
Daten für Kunde "Müller" erfassen;
Checkpunkt setzen;
Vertrag "Müller" in Vertragsdatenbank speichern;
Maske "Vertragsdaten" löschen;
Vertrag "Müller" aus Vertragsdatenbank lesen;
Maskeninhalt mit Checkpunkt vergleichen;
```

Wenn der Check klappt, stimmt der aus der Datenbank gelesene Vertrag mit dem gespeicherten überein, woraus geschlossen werden kann, dass das System Verträge korrekt speichert. Beim erneuten Abspielen des Skripts ist der Testautomatisierer überrascht, weil das Skript unerwartet stehen bleibt. Was ist passiert?

*Problem: Regressionstestfähigkeit*

Wird das Skript ein zweites Mal abgespielt, reagiert das Testobjekt beim Versuch, den Vertrag »Müller« zu speichern, anders als im ersten Lauf. Der Vertrag »Müller« ist in der Vertragsdatenbank jetzt schon vorhanden, und das Testobjekt quittiert den Versuch, den Vertrag doppelt zu speichern, mit der Meldung:

```
"Vertrag vorhanden.
Soll Vertrag überschrieben werden J/N?".
```

Das Testobjekt erwartet nun eine Tastatureingabe. Da diese im aufgezeichneten Testskript fehlt, bleibt der automatisierte Test stehen.

Beide Testläufe unterscheiden sich in ihren Vorbedingungen. Weil das aufgezeichnete Skript die Vorbedingung (Vertrag »Müller« nicht in der Datenbank) voraussetzt, ist der per »Capture« automatisierte Testfall nicht regressionstestfähig. Abhilfe schafft hier die Programmierung einer Fallunterscheidung oder das Löschen des Vertrags aus der Datenbank als letzte »Aufräumaktion« des Testfalls.

---

*Werkzeuge für Last- und Performanztests*

Capture-and-Replay-Tools werden in erster Linie zur Automatisierung funktionaler Tests eingesetzt. Auch für nicht funktionale Tests gibt es Werkzeugunterstützung, insbesondere für Last- und Performanztests.

Werkzeuge für den Lasttest generieren synthetische Last, z. B. Datenbankabfragen, Benutzertransaktionen oder Netzwerkverkehr. Sie werden eingesetzt zur Durchführung von Volumen-, Stress- und Performanztests. Werkzeuge für den Performanztest messen und protokollieren das Antwortzeitverhalten des untersuchten Systems, abhängig von der eingespeisten Last. Die jeweils benötigten »Messfühler« werden als Monitore bezeichnet.

***Exkurs***

Last-/Performanztests (vgl. Kap. 3.4.5) sind notwendig, wenn ein Softwaresystem eine große Zahl an parallelen Anfragen oder Transaktionen ausführen muss, wobei gewisse maximale Antwortzeiten nicht überschritten werden dürfen. Derartige Anforderungen müssen Echtzeitsysteme und in der Regel Client/Server-Systeme so-

wie webbasierte Anwendungen erfüllen. Durch Performanztests kann geprüft werden, wie die Antwortzeiten solcher Systeme mit steigender Last (z. B. steigender Anwenderzahl) anwachsen und ab welchem Punkt das System überlastet ist, so dass beim Einsatz der Softwaresysteme unakzeptable lange Wartezeiten auftreten. Werkzeuge für den Performanztest liefern dem Tester als Analysehilfsmittel üblicherweise umfangreiche Messprotokolle, Berichte und Diagramme, die die Antwortzeiten des Systems in Abhängigkeit zur eingespeisten Last wiedergeben und Hinweise auf Performanzengpässe geben. Stellt sich im Test heraus, dass eine Überlastung schon bei einer im Alltagsbetrieb auftretenden Last gegeben ist, müssen Maßnahmen zum Systemtuning (Ausbau der Hardware, Optimierung performanzkritischer Softwarekomponenten) erfolgen.

### 7.1.4 Werkzeuge zur Test- und Testobjektanalyse

*Komparatoren*

Um Unterschiede zwischen erwartetem und aktuellem Ergebnis festzustellen, werden →Komparatoren eingesetzt. Komparatoren können üblicherweise mit marktgängigen Datei- und Datenbankformaten arbeiten und Abweichungen zwischen Dateien mit Soll- und Istdaten herausfiltern. Testroboter besitzen normalerweise eingebaute Komparatorfunktionen, die mit Konsoleninhalten, GUI-Objekten oder Kopien des Bildschirminhalts arbeiten. Meist bieten diese Werkzeuge Filter- oder Maskiermechanismen an, um für den Soll/Istvergleich nicht relevante Daten oder Datenbereiche ausblenden oder überspringen zu können. Dies ist beispielsweise notwendig, wenn Datum-/Uhrzeit-Informationen in der Datei- oder Bildschirmausgabe des Testobjekts enthalten sind. Denn Datum und Uhrzeit wären von Testlauf zu Testlauf verschieden, und der Komparator würde diese Änderung fälschlicherweise als Soll-/Istabweichung interpretieren.

*Dynamische Analyse*

Werkzeuge für die dynamische Analyse ermitteln während der Programmausführung im dynamischen Test zusätzliche Informationen über den internen Zustand der getesteten Software, z. B. die Belegung, Verwendung und Freigabe von Speicher (Memory Leaks, Zeigerzuordnungen, Zeigerarithmetik usw.).

*Überdeckungsanalyse*

Überdeckungsanalysatoren liefern Maßzahlen der strukturellen Testabdeckung während der Testdurchführung (vgl. Kap. 5.2). Hierzu werden vor der Testdurchführung (von einer Instrumentierungskomponente des Analysetools) Messanweisungen in das Testobjekt eingefügt (Instrumentierung). Wird im Testlauf eine solche Messanweisung ausgelöst, wird die entsprechende Programmstelle als »überdeckt« protokolliert. Nach der Testdurchführung wird die Protokolldatei analysiert und eine Überdeckungsstatistik erzeugt. Die meisten Werkzeuge liefern einfache Überdeckungsmetriken, wie Anweisungs- und Zweigüberdeckung (vgl. Kap. 5.2.1 und 5.2.2).

*Statische Analyse*

Statische Analysatoren liefern Maßzahlen zu verschiedenen Charakteristika des Programmcodes, wie z. B. zyklomatische Zahl und andere Codemetriken. Sie können Datenfluss- und Kontrollflussanomalien aufdecken und sind somit ein geeignetes Werkzeug zur Erkennung von Unstimmigkeiten und Fehlern (vgl. Kap. 4.2).

## 7.2 Auswahl und Einführung von Testwerkzeugen

Von den oben vorgestellten Testwerkzeugen sind einige elementare Werkzeuge (z. B. Komparator, Abdeckungsanalysator) in verschiedenen Betriebssystemumgebungen (z. B. UNIX) bereits standardmäßig vorhanden. Der Tester kann sich in diesem Fall mit einfachen vorhandenen Mitteln die benötigte Werkzeugunterstützung zusammenstellen. Die Fähigkeiten solcher betriebssystemeigenen Werkzeuge sind naturgemäß limitiert, so dass es sinnvoll ist, am Markt erhältliche spezielle Testwerkzeuge anzuschaffen.

Wie oben beschrieben, werden hier für jede Phase im Testprozess spezielle Werkzeuge am Markt angeboten, die dem Tester bei den phasenspezifischen Arbeiten helfen bzw. die Arbeiten übernehmen – von Werkzeugen für Testplanung und Testspezifikation, die den Tester im kreativen Entwurfsprozess unterstützen, bis hin zu Testtreibern und Testrobotern, die mechanische Testdurchführungsarbeiten automatisieren können.

Wenn über die Anschaffung von Testwerkzeugen nachgedacht wird, sollten Tools zur Automatisierung der Testdurchführung nicht als erste oder einzige Möglichkeit in Betracht gezogen werden.

*Automating chaos gives faster chaos*

Wo Werkzeugunterstützung Vorteile bringen kann, ist stark abhängig vom jeweiligen Projektumfeld und von der Reife des dort anzutreffenden Entwicklungs- und Testprozesses. In einem chaotischen Projektumfeld, in dem »Programmierung auf Zuruf« üblich ist, Dokumentation ein Fremdwort ist und Testen unstrukturiert oder gar nicht erledigt wird, ist an eine Automatisierung der Testdurchführung nicht zu denken. Niemals kann ein Werkzeug einen nicht vorhandenen Prozess ersetzen oder eine schlampige Vorgehensweise kompensieren. »It is far better to improve the effectiveness of testing first than to improve the efficiency of poor testing. Automating chaos just gives faster chaos« ([Fewster 99, S. 11]).

In solchen Situationen muss zuerst der manuelle Testbetrieb geordnet werden. Das heißt, zunächst muss ein systematischer Testprozess definiert, eingeführt und gelebt werden. Dann kann überlegt werden, bei welchen Prozessschritten die Produktivität oder Arbeitsqualität

durch den Einsatz von Werkzeugen erhöht werden kann und sollte. Bei der Einführung von Tools aus den verschiedenen vorgestellten Kategorien ist es empfehlenswert, sich an folgender Einführungsreihenfolge zu orientieren:

*Einführungsreihenfolge beachten*

1. Fehlermanagement
2. Konfigurationsmanagement
3. Testplanung
4. Testdurchführung
5. Testspezifikation

*Lernkurve beachten*

Zu beachten sind auch die Zeiträume, die nötig sind, um den Umgang mit einem neuen Werkzeug zu etablieren. Aufgrund der Lernkurve kann statt einer erhofften Produktivitätssteigerung in einer Übergangsphase sogar eine Produktivitätseinbuße zu verzeichnen sein. Daher ist es gefährlich, in »heißen« Projektphasen ein neues Werkzeug einzuführen, in der Hoffnung, Personalengpässe kurzfristig durch »Automatisierung« zu lösen.

### 7.2.1 Wirtschaftlichkeit der Werkzeugeinführung

Die Einführung eines neuen Werkzeugs ist mit Kosten für Auswahl, Anschaffung und Wartung der Werkzeuge verbunden. Kosten für Hardware und Mitarbeiterschulung können hinzukommen. Das Investitionsvolumen kann je nach Komplexität des Werkzeugs und Anzahl der auszustattenden Arbeitsplätze schnell in den sechsstelligen Bereich gehen. Wie bei jeder Investition interessiert natürlich auch hier der Zeitrahmen der Amortisierung des neuen Testwerkzeugs.

*Kosten-Nutzen-Bilanz aufstellen*

Bei Werkzeugen zur Automatisierung der Testdurchführung kann gut abgeschätzt werden, wie viel Aufwandsersparnis ein automatisierter Testlauf gegenüber der manuellen Durchführung bringt. Der neue Aufwand für die Programmierung der Tests muss natürlich abgezogen werden, so dass nach nur einem automatisierten Testlauf die Kosten-Nutzen-Bilanz meist negativ ausfällt. Erst mit weiteren automatisierten Regressionstestläufen (s. Abb. 7-1) summiert sich die pro Testlauf erzielte Ersparnis.

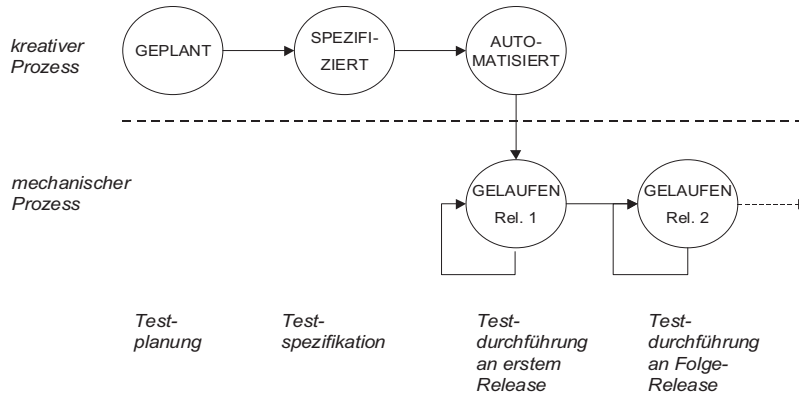



***Abb. 7–1*** *Lebenszyklus eines Testfalls*

Die Bilanz fällt nach einer gewissen Anzahl von Regressionstests positiv aus. Eine genaue Abschätzung der Amortisierungszeit ist schwierig. Eine Kostendeckung wird nur erreicht, wenn die Tests regressionstestfähig entworfen und programmiert sind. Ist diese Regressionstestfähigkeit gegeben, kann bei Capture-Replay-Tools eine positive Bilanz durchaus schon ab dem dritten Testzyklus erreicht werden (vgl. [URL: imbus 99]). Eine solche Rechnung macht natürlich nur Sinn, wenn die Alternative einer manuellen Durchführung besteht. Es gibt jedoch viele Tests, die rein manuell gar nicht machbar sind (z. B. Performanztests).

*Einfluss auf die Testqualität bewerten*

Nur über den Testaufwand zu argumentieren, greift zu kurz. Es muss auch betrachtet werden, inwieweit die Testqualität durch den Einsatz des neuen Werkzeugs ansteigt, wodurch mehr Fehler gefunden und behoben werden. Entwicklungs-, Support- und Wartungskosten werden dadurch gesenkt, was sich allerdings erst mittelfristig auswirkt. Das Sparpotenzial ist hier aber wesentlich höher und damit interessanter.

Zusammenfassend ist festzustellen:

- Die kreativen Testarbeiten können durch Werkzeuge unterstützt werden. Sie helfen dem Tester, die Testqualität zu steigern.
- Die mechanische Testdurchführung kann automatisiert werden, was den Testaufwand reduziert oder bei gleichem Aufwand mehr Tests zulässt. Mehr Tests bedeutet aber nicht zwangsläufig bessere Tests.
- In beiden Fällen gilt, dass ohne guten Testprozess bzw. eingespielte Testmethoden Werkzeuge gar nicht den gewünschten Einsparungseffekt erzielen können.

### 7.2.2 Werkzeugauswahl

Wenn geklärt ist, welche Testaufgabe durch ein Werkzeug unterstützt werden soll, beginnt die eigentliche Auswahl (Evaluation) des Werkzeugs. Da die Investitionen, wie oben erläutert, sehr hoch sein können, muss hier sorgfältig und geplant vorgegangen werden. Der Auswahlprozess weist fünf Schritte auf:

1. Anforderungsspezifikation für den Werkzeugeinsatz
2. Marktstudie (Aufstellen einer Übersichtsliste der Kandidaten)
3. Vorführung von Werkzeugdemos
4. Evaluierung der Werkzeuge der engeren Wahl
5. Review der Ergebnisse und Werkzeugauswahl

Im ersten Schritt, der Anforderungsspezifikation, können folgende Kriterien eine Rolle spielen:

*Auswahlkriterien*

- Güte des Zusammenspiels mit den potenziellen Testobjekten
- Know-how der Tester über Werkzeug oder Methode
- Möglichkeit zur Integration in die vorhandene Entwicklungsumgebung
- Möglichkeit zur Integration mit anderen eingesetzten Testtools
- Plattform, auf der das Werkzeug eingesetzt werden soll
- Integrationsmöglichkeiten mit Werkzeugen desselben Herstellers
- Service, Verlässlichkeit und Marktstellung des Herstellers
- Lizenzbedingungen, Preis, Wartungskosten

Diese und mögliche weitere individuelle Kriterien werden in einer Liste zusammengestellt und dann gemäß ihrer relativen Bedeutung gewichtet. Absolut unverzichtbare Eigenschaften werden als K.-o.-Kriterien gekennzeichnet. Ein Muster eines solchen Kriterienkatalogs ist unter [URL: imbus-downloads] verfügbar.

*Markrecherche und Vorauswahl*

Parallel dazu wird der Markt gesichtet, eine Liste der in der jeweiligen Werkzeugkategorie überhaupt verfügbaren Produkte erstellt und Produktbeschreibungen angefordert bzw. im Internet recherchiert. Die aufgrund ihrer Beschreibung besten Kandidaten werden dann vom jeweiligen Hersteller bzw. dessen Vertrieb vorgeführt. Durch diese Demonstrationen wird meist ein recht zuverlässiger Eindruck über die dahinterstehende Firma und deren Servicephilosophie gewonnen. In die endgültige Evaluierungsrunde werden dann die besten Anbieter aufgenommen. Hier gilt es vor allem folgende Punkte zu verifizieren:

- Harmoniert das Werkzeug mit den Testobjekten und der Entwicklungsumgebung?

- Werden die sonstigen Leistungsmerkmale, wegen derer das jeweilige Werkzeug die Evaluationsrunde erreicht hat, in der Praxis eingehalten bzw. überhaupt erreicht? Die Werbung verspricht viel.
- Kann der Herstellersupport (auch nach dem Kauf[39]) qualifizierte Auskunft und Hilfe auch bei Nichtstandardfragen liefern?

### 7.2.3 Werkzeugeinführung

Ist die Wahl getroffen, gilt es das Werkzeug im eigenen Unternehmen einzuführen. Üblicherweise wird hierzu zunächst ein Pilotprojekt durchgeführt, um nachzuweisen, ob der erwartete Nutzen in einem realen Projektumfeld auch tatsächlich erreicht wird. Das Pilotprojekt sollte nicht von denselben Personen durchgeführt werden, die schon bei der Evaluation mitgearbeitet haben. Die Evaluationsergebnisse werden sonst unter Umständen zu unkritisch übernommen.

*Pilotbetrieb*

Die im Pilotbetrieb gewonnenen Erfahrungen helfen bei den weiteren Schritten. Es wird deutlich, wo und in welchem Umfang Schulungsbedarf besteht und wo ggf. Änderungen am Testprozess notwendig sind. Des Weiteren sollten im Pilotprojekt Daten gesammelt werden, die eine realistische Kosten-Nutzen-Bewertung ermöglichen. Es wäre falsch, mit überzogenen Versprechungen in die Breiteneinführung zu gehen. Die Akzeptanz der späteren Anwender des Werkzeugs muss auf realistischen Erwartungen beruhen. Denn die Einführung des Werkzeugs erfordert starke und anhaltende Zustimmung von den neuen Anwendern, da die Benutzung jedes neuen Werkzeugs immer einen anfänglichen Mehraufwand bedeutet. Deshalb ist es wichtig, begleitende Benutzertrainings- und Coachingmaßnahmen vorzusehen. Eine erfolgreiche Werkzeugeinführung gliedert sich damit in folgende sechs Schritte:

*Einführungsschritte*

1. Pilotprojekt
2. Review der Pilotprojekterfahrungen
3. Prozessanpassung
4. Anwenderschulung
5. Breiteneinführung
6. Begleitendes Coaching

In diesem Kapitel wurde auf viele Schwierigkeiten und den zusätzlichen Aufwand bei der Auswahl und dem Einsatz von Werkzeugen zur Unterstützung der unterschiedlichen Tätigkeiten im Testprozess hingewie-

39. Bei manchen Herstellern verweisen die Techniker aus der Vertriebsniederlassung nach dem Kauf an den telefonischen Support der Zentrale.

sen. Es soll aber nicht der Eindruck entstehen, dass sich ein Werkzeugeinsatz nicht lohnt. Ganz im Gegenteil, Testen ist ohne die Unterstützung entsprechender Werkzeuge bei größeren Projekten nicht durchführbar. Eine sorgfältige Werkzeugeinführung ist jedoch notwendig. Andernfalls wird das teure Werkzeug schnell zur »Schrankware« (liegt im Schrank und wird nicht verwendet).

## 7.3 Zusammenfassung

- Für jede Phase im Testprozess sind Werkzeuge verfügbar, die dem Tester helfen, seine Testarbeiten qualitativ zu verbessern und/oder zu automatisieren.
- Die Nutzung eines Testwerkzeugs bringt nur Vorteile, wenn der Testprozess als solcher beherrscht wird und definiert abläuft.
- Die Einführung eines Testwerkzeugs kann mit hohen Investitionen verbunden sein, weshalb die Werkzeugauswahl sorgfältig und nachvollziehbar erfolgen muss.
- Die Einführung des ausgewählten Werkzeugs muss begleitet werden. Information und Training der späteren Anwender helfen, deren Akzeptanz und damit den regelmäßigen Einsatz des Werkzeugs sicherzustellen.

# A Testkonzept nach IEEE 829

*Dieser Anhang erläutert die Inhalte eines Testkonzepts nach IEEE 829 und kann als Hilfestellung bei der Erstellung eines Testkonzepts dienen.*

### Testkonzeptbezeichnung

Kennung/Bezeichnung des Testkonzepts, die sicherstellen muss, dass das Dokument in anderen Projektdokumenten klar und eindeutig referenziert werden kann. Ein entsprechendes Benennungsschema wird oft durch das Projektarchiv oder ein zentrales Dokumentenmanagement vorgegeben. Je nach Größe der Organisation, in der das Testprojekt abgewickelt wird, kann so eine Dokumentenkennung beliebig kompliziert aufgebaut sein. Mindestens berücksichtigt sein sollten: Dokumentname, Dokumentversion, Freigabestatus.

### Einführung

Die Einführung sollte eine kurze Darstellung des Projekthintergrundes bieten, die allen am Projekt beteiligten (Kunde, Management, Entwickler, Tester) hilft, den Inhalt des Testkonzepts besser verstehen und einordnen zu können.

Des Weiteren sollte eine Auflistung aller Ausgangsdokumente, die das Testkonzept referenziert oder umsetzen muss, enthalten sein. Dies sind üblicherweise: Normen, Unternehmensstandards, Projektstandards, Kundenstandards, Spezifikationen u. Ä.

### Testobjekte

Kurze Darstellung, aus welchen Teilen/Komponenten die zu testende Software besteht. Auflistung derjenigen Komponenten daraus, die Gegenstand des Tests sind (inkl. Angabe von Versionsnummer, Übergabemedium, zugehörige Spezifikationen usw.).

Um Missverständnisse zu vermeiden, sollen die Systemteile, die nicht getestet werden, aufgelistet werden.

### Zu testende Leistungsmerkmale

Identifiziert, welche Funktionen oder Eigenschaften des Systems zu testen sind. Verweist auf zugehörige Testspezifikationen und ordnet ggf. Testobjekte und Teststufen zu.

### Leistungsmerkmale, die nicht getestet werden

Um Missverständnisse zu vermeiden bzw. unberechtigten Erwartungen vorzubeugen, muss auch klargestellt werden, welche Aspekte nicht getestet werden sollen oder nicht getestet werden können (z. B. aus Ressourcenmangel oder technischen Gründen).

*Tipp*

- Da das Testkonzept ziemlich früh im Projektverlauf erstellt wird, wird diese Liste unvollständig sein. Oft wird sich später herausstellen, dass weitere Komponenten oder Leistungsmerkmale (obwohl geplant) doch nicht testbar sind. Der verantwortliche Testmanager muss dann entsprechende Warnungen in seinen Statusbericht aufnehmen.

### Teststrategie

Definition der Testziele, möglichst basierend auf einer Risikoanalyse. Diese klärt, welche Risiken drohen, wenn (mangels Test) Fehler nicht gefunden werden. Daraus folgt, welche Testziele unverzichtbar, wichtig und weniger wichtig sind. Hierdurch wird sichergestellt, dass wichtige Themen zuerst angegangen werden.

Darauf aufbauend erfolgt die Auswahl und Darstellung der Testmethoden, die angewendet werden sollen. Es muss klar erkennbar sein, ob und warum die ausgewählten Methoden geeignet sind, die Testziele zu erreichen, und zwar unter Berücksichtigung der vorhandenen Ressourcen und der identifizierten Risiken.

### Abnahmekriterien

Nachdem alle für ein Testobjekt vorgesehenen Tests durchgeführt worden sind, muss anhand der Testresultate entschieden werden, ob das jeweilige Testobjekt freigegeben[40] und geliefert[41] werden kann. Hierzu werden Abnahme- bzw. Testendekriterien festgelegt.

Das Kriterium »fehlerfrei« wäre hier ein ziemlich ungeeignetes Kriterium, weil »fehlerfrei« durch Testen nicht nachgewiesen werden kann. Üblicherweise wird deshalb ersatzweise eine Kombination von Kriterien aus »Anzahl der gelaufenen Tests«, »Anzahl der gefundenen Fehler« und »Schweregrad der gefundenen Fehler« gewählt.

Zum Beispiel: Mindestens 90 % der geplanten Tests sind gelaufen, und es wurden keine Abstürze festgestellt.

---

40. Freigabe: Managemententscheidung, dass das betreffende Testobjekt als »fertig« betrachtet wird.
41. Liefern kann auch bedeuten: Übergabe an die nächste Teststufe.

Solche Abnahmekriterien können von Testobjekt zu Testobjekt unterschiedlich gewählt werden. Die Schärfe der Kriterien ist an der Risikobetrachtung zu orientieren, d. h., für unkritische Testobjekte werden weichere Abnahmebedingungen formuliert als etwa für sicherheitskritische Testobjekte. Der Testaufwand wird so auf die wichtigen Systemteile konzentriert.

#### Kriterien für Testabbruch und Testfortsetzung

Neben Abnahme- oder Testendekriterien werden auch Kriterien benötigt, die eine Unterbrechung bzw. den Abbruch von Tests anzeigen.

Denn es kann sein, dass ein Testobjekt so schlecht ist, dass auch nach unzähligen Tests nie die Abnahmefähigkeit erreicht wird. Hier werden zusätzliche Kriterien benötigt, die dieses unnütze Testen unterbinden, indem sie rechtzeitig einen Testabbruch auslösen. Das Testobjekt wird dann, ohne dass alle geplanten Tests durchexerziert werden müssen, zur Nachbesserung an die Entwickler zurückgegeben.

Analog werden Kriterien für die Fortsetzung bzw. Wiederaufnahme der Tests benötigt. Hierzu führt die zuständige Testgruppe z. B. einen Eingangstest durch. Nur wenn diese entsprechend ausgewählten Tests fehlerlos laufen, steigt man in die eigentliche Testdurchführung ein.

---

***Tipp***

- Es sollten nur solche Metriken definiert werden, die regelmäßig, zuverlässig und einfach zu messen sind, z. B. weil sie von den eingesetzten Testwerkzeugen automatisch geliefert werden. Der Testmanager sollte die aktuellen Messergebnisse in jedem seiner Berichte auflisten und interpretieren.

---

#### Testdokumentation

Hier wird vereinbart, welche Berichte und Ergebnisse die Testaktivitäten liefern und wann, in welcher Form und an wen diese Ergebnisse kommuniziert werden. Gemeint sind hier nicht nur die Testergebnisse im engeren Sinne (z. B. Fehlermeldungen, Testprotokolle), sondern auch alle Planungsdokumente (z. B. Testkonzept, Testspezifikationen, Zeitplan), Begleitdokumente zu Testobjektübergaben (Testobjektlieferschein) und zusammenfassende Teststatusberichte.

---

***Tipp***

- Obwohl an dieser Stelle nur das formale Berichtswesen behandelt wird, ist die informelle Kommunikation in Testprojekten nicht zu vernachlässigen. Insbesondere in Projekten, die schon in Schieflage geraten sind, oder in »heißen« Projektphasen (Abnahmewoche etc.) sucht ein erfahrener

Testmanager noch stärker als sonst auch das informelle, direkte Gespräch mit den »Testbetroffenen«. Dabei geht es nicht darum, schlechte Nachrichten zu beschönigen, sondern die Bereitschaft zu wecken, aus diesen vielleicht schlechten Nachrichten die richtigen Konsequenzen zu ziehen.

#### Testaufgaben

Auflistung aller Aufgaben, die zur Planung und Durchführung der Tests nötig sind, mit Zuordnung der Verantwortlichkeiten. Der Aufgabenstatus (offen, laufend, zurückgestellt, erledigt) muss verfolgt werden. Dieser Punkt gehört deshalb besser in den regelmäßigen Teststatusbericht, auf den hier verwiesen werden sollte.

#### Testinfrastruktur

Auflistung der Bestandteile der Testinfrastruktur, die notwendig sind, um die geplanten Tests abarbeiten zu können. Hierzu gehören üblicherweise: Testplattform(en), Testerarbeitsplätze und deren Ausstattung, Testtools, Entwicklungsumgebung bzw. die Teile davon, die auch die Tester benötigen, sowie sonstige Werkzeuge (E-Mail, WWW, Officepaket usw.)

***Tipp***

- Gedanken machen sollte sich der Testmanager in diesem Zusammenhang zu folgenden Aspekten: Beschaffung nicht vorhandener Teile aus o. g. Wunschliste, Budgetfragen, Administration und Betrieb der Testinfrastruktur (meist werden zur Administration der Testinfrastruktur, der Testobjekte und der Testtools – zumindest zeitweise – verschiedene Spezialisten benötigt, die evtl. aus anderen Abteilungen oder von externen Dienstleistern angefordert werden müssen).

#### Verantwortlichkeiten/Zuständigkeiten

Organisatorische Einbindung des Testpersonals in das Projekt, Weisungsbefugnis, Verfügbarkeit u. Ä. Gegebenenfalls Aufteilung/Organisation des Personals in verschiedene Testgruppen und/oder Teststufen.

Zuordnung zwischen Personal/Personen und Aufgabenfeldern/Aufgaben.

***Tipp***

- Die Verantwortlichkeiten/Zuständigkeiten werden sich häufiger im Projektverlauf ändern. Daher wird empfohlen, diese Zuordnung tabellarisch darzustellen und als Anhang des zu erstellenden Testkonzepts zu führen.

#### Personal, Einarbeitung, Ausbildung

Darstellung der zur Planumsetzung notwendigen Personalausstattung (Rollen, Qualifikationen, Personalkapazität, Einsatz- und Urlaubsplanung), nicht nur für das Testpersonal im engeren Sinne, sondern auch für das Personal zur Administration der Testinfrastruktur (s. o.), einzubeziehende Entwickler und Kunden.

Maßnahmenplan zur Ausbildung bzw. Einweisung des Personals in die jeweiligen Aufgaben.

#### Zeitplan/Arbeitsplan

Grober Zeitplan der Testaktivitäten auf Meilensteinebene. Diese Planung muss in den Projektplan des Projektmanagers Eingang finden und dort gepflegt werden. Eine regelmäßige Abstimmung zwischen Projektmanager und Testmanager muss gegeben sein: Der Testmanager muss über Lieferverzüge der Entwicklung informiert werden und auf diese in seiner Detailtestplanung reagieren. Der Projektmanager muss auf Testergebnisse reagieren und ggf. Meilensteine verschieben, weil zusätzliche Korrekturzyklen vorgesehen werden müssen.

***Tipp***

- Der Testmanager muss sicherstellen, dass sein Zeit- und Aufgabenplan im Projektplan des Projektmanagers Eingang findet. Die QS- und Testaktivitäten dürfen kein »Paralleluniversum« bilden.

#### Planungsrisiken und Unvorhergesehenes

Im Abschnitt »Teststrategie« geht es um Risiken, die im Testobjekt bzw. dessen Betrieb liegen. Hier geht es nun um die Risiken, die sich ergeben, wenn Probleme bei der Umsetzung des Testkonzepts auftauchen, oder um Risiken, die daraus resultieren, dass wünschenswerte Maßnahmen nicht eingeplant wurden, da schon zum Zeitpunkt der Planerstellung klar ist, dass sie im konkreten Projekt nicht umsetzbar sind.

Als Minimum sollte eine Liste solcher Risikopunkte geführt werden. Wobei periodisch (z. B. im regelmäßigen Teststatusbericht) die erkannten Risiken bewertet und ggf. Maßnahmen zur Risikominderung eingeleitet werden.

#### Genehmigung/Freigabe

Auflistung der betroffenen Personen/Organisationseinheiten, die das Testkonzept freigeben, ihm zustimmen oder zumindest informiert wer-

den müssen. Dokumentation der jeweiligen Zustimmung durch Unterschrift. Dokumentation der Kenntnisnahme nach (wesentlichen) Änderungen, insbesondere Strategieänderungen oder personellen Änderungen.

*Tipp*

- Betroffene Personengruppen/Organisationseinheiten sind üblicherweise Entwicklungsgruppe(n), Projektmanagement, Projektlenkungsausschuss, Betreiber des Softwaresystems, Anwender des Softwaresystems, Auftraggeber/Projektkunde und natürlich die Testgruppe(n).

Je nach Projektsituation kann die Intention der hier dokumentierten Zustimmung eine andere sein.

Wunschsituation: »Ihr stimmt zu, dass die genannten umfangreichen Ressourcen eingesetzt und finanziert werden, um dieses System wie hier dargestellt angemessen testen zu können.«

Die leider oft realistische Situation: »Wegen des dargestellten Ressourcenmangels können Tests nur in einem unzureichenden/allernotwendigsten Umfang durchgeführt werden. Dennoch stimmt ihr diesem Ansatz zu und akzeptiert, dass darauf aufbauend Freigabe- und Lieferentscheidungen getroffen werden, die mit hohen Risiken verbunden sind.«

### Glossar

Da im Testbereich ein ziemlicher Begriffswildwuchs herrscht, sollte (ergänzend zu o. g. Gliederung nach IEEE 829) auf ein Glossar der im Projekt verwendeten Test-Fachbegriffe nicht verzichtet werden. Die Gefahr, dass die verschiedenen Personengruppen, die betroffenen sind, ganz unterschiedliche Interpretationen ein und desselben Testbegriffs zugrunde legen, ist sehr sehr hoch. Als Beispiel können ja einmal die verschiedenen am Projekt beteiligten Parteien nach ihrer Definition des Begriffs »Lasttest« gefragt werden.

# B Wichtige Hinweise zum Lehrstoff und zur Prüfung zum Certified Tester

Dem vorliegenden Lehrbuch liegt der Lehrstoff zum »Certified Tester – Foundation Level« gemäß den Lehrplänen [URL: ISEB CTF 99] und [URL: GTB CTF 01] zugrunde.

Der deutsche Lehrplan des ASQF enthält den ISEB-Lehrstoff und zusätzlich für den deutschen Markt spezifische Ergänzungen (z. B. der Bezug zu DIN-Normen). Der ISEB-Lehrplan wird durch das »British Testing Board« in größeren Zeitabständen überarbeitet. Den ASQF-Lehrplan pflegt das »German Testing Board« [URL: GTB]. Weitere nationale Ausgaben des Lehrplans erstellen und pflegen die zuständigen Boards. Die nationalen Boards stimmen sich gegenseitig ab und garantieren Kompatibilität ihrer Lehrpläne und Prüfungen. Zuständiges Gremium ist das »International Software Testing Qualification Board« [URL: ISTQB].

Grundlage für Prüfungen ist der jeweils zum Prüfungszeitpunkt aktuelle Lehrplan in der jeweiligen Prüfungssprache. Die Prüfungen werden angeboten und durchgeführt vom jeweiligen nationalen Board oder dessen Beauftragten. Weitere Informationen zu den Lehrplänen und Prüfungen sind unter [URL: ISTQB] für den internationalen Bereich und unter [URL: GTB] für den deutschen Bereich einsehbar.

Der in diesem Buch enthaltene Stoff wird aus didaktischen Gründen ggf. in anderer Reihenfolge als im Lehrplan behandelt. Aus dem Umfang einzelner Kapitel kann nicht auf die Prüfungsrelevanz der dargestellten Inhalte geschlossen werden. Im Buch sind einzelne Themen vertieft dargestellt. Solche Passagen, die über den Lehrplan hinausgehen, sind als Exkurs gekennzeichnet. Prüfungsgrundlage sind auf jeden Fall die offiziellen Lehrpläne.

Im Buch oder auf der zugehörigen WWW-Seite enthaltene Übungsaufgaben und Übungsfragen sind ausschließlich als Übungsmaterial und Beispiele zu verstehen. Sie entsprechen nicht den offiziellen Prüfungsfragen.

# C Übungsaufgaben

### Übungsaufgaben zu Kapitel 2

2.1 Definieren Sie die Begriffe Fehlerwirkung, Fehlerzustand, Fehlhandlung.

2.2 Was ist Fehlermaskierung?

2.3 Erläutern Sie den Unterschied zwischen Testen und Debugging.

2.4 Erläutern Sie, warum jeder Test eine stichprobenartige Prüfung ist.

2.5 Nennen Sie die Hauptmerkmale der Softwarequalität nach ISO 9126.

2.6 Definieren Sie den Begriff Zuverlässigkeit eines Systems.

2.7 Erläutern Sie die Phasen des fundamentalen Testprozesses.

2.8 Was ist ein Testorakel?

2.9 Warum sollte ein Entwickler nicht seine eigenen Programme testen?

### Übungsaufgaben zu Kapitel 3

3.1 Erläutern Sie die einzelnen Phasen des allgemeinen V-Modells.

3.2 Definieren Sie die Begriffe Verifikation und Validierung.

3.3 Begründen Sie, warum Verifikation sinnvoll ist, auch wenn eine sorgfältige Validierung stattfindet (und umgekehrt).

3.4 Charakterisieren Sie die typischen Testobjekte im Komponententest.

3.5 Nennen Sie die Testziele des Integrationstests.

3.6 Welche Integrationsstrategien lassen sich unterscheiden?

3.7 Welche Gründe sprechen dafür, Tests in einer separaten Testinfrastruktur durchzuführen?

3.8 Erläutern Sie anforderungsbasiertes Testen.

3.9 Definieren Sie Lasttest, Performanztest, Stresstest. Was sind die Unterscheidungsmerkmale?

3.10 Worin unterscheiden sich Fehlernachtest und Regressionstest?

3.11 In welcher Projektphase nach allgemeinem V-Modell sollte das Testkonzept erstellt werden?

### Übungsaufgaben zu Kapitel 4

4.1 Beschreiben Sie die grundlegenden Schritte, die bei einem Review durchzuführen sind.

4.2 Welche Reviewarten werden unterschieden?

4.3 Welche Rollen wirken an einem technischen Review mit?

4.4 Warum sind Reviews ein effizientes Mittel zur Qualitätssicherung?

4.5 Erklären Sie den Begriff statische Analyse.

4.6 Wie stehen statische Analyse und Reviews in Zusammenhang?

4.7 Warum kann die statische Analyse nicht alle in einem Programm enthaltenen Fehlerzustände aufdecken?

4.8 Welche Arten von Datenflussanomalien werden unterschieden?

### Übungsaufgaben zu Kapitel 5

5.1 Was ist ein dynamischer Test?

5.2 Wozu dient ein Testrahmen?

5.3 Wodurch unterscheiden sich Black-box- und White-box-Testverfahren?

5.4 Erklären Sie das Verfahren der Äquivalenzklassenbildung.

5.5 Was ist ein Repräsentant?

5.6 Wie ist das Testendekriterium Äquivalenzklassenüberdeckung definiert?

5.7 Warum ist die Grenzwertanalyse eine gute Ergänzung zur Äquivalenzklassenbildung?

5.8 Nennen Sie weitere Black-box-Verfahren.

5.9 Erläutern Sie den Begriff Anweisungsüberdeckung.

5.10 Worin unterscheiden sich Anweisungs- und Zweigüberdeckung?

5.11 Wozu dient eine Instrumentierung?

### Übungsaufgaben zu Kapitel 6

6.1 Welche grundsätzlichen Modelle einer Aufgabenteilung zwischen Entwicklung und Test lassen sich unterscheiden?

6.2 Welche Rollen sind beim Testen auszufüllen und mit speziell qualifizierten Mitarbeitern zu besetzen?

6.3 Welche Aufgaben hat der Testmanager?

6.4 Welche Arten von Metriken zur Überwachung des Testfortschritts lassen sich unterscheiden?

6.5 Welche Informationen sollte ein Teststatusbericht enthalten?

6.6 Welche Daten sollte eine Fehlermeldung enthalten?

6.7 Was ist der Unterschied zwischen Fehlerpriorität und Fehlerklasse?

6.8 Wozu wird ein Fehlerstatusmodell benötigt?

6.9 Welche Aufgabe hat ein Änderungskontrollausschuss?

6.10 Welche Anforderungen aus Sicht des Tests stellen sich an das Konfigurationsmanagement?

6.11 Welche grundsätzlichen Arten von Normen und Standards lassen sich unterscheiden?

### Übungsaufgaben zu Kapitel 7

7.1 Welche grundlegenden Funktionen bieten Testplanungswerkzeuge?

7.2 Welche Typen von Testdatengeneratoren werden unterschieden?

7.3 Welcher Typ von Testdatengenerator kann auch Sollwerte generieren? Warum können die anderen Typen das nicht?

7.4 Erläutern Sie die prinzipielle Funktionsweise eines Capture-and-Replay-Tools.

7.5 Aus welchen Schritten besteht der Prozess zur Auswahl eines Testwerkzeugs?

7.6 Welche Schritte sind bei der Werkzeugeinführung zu beachten?

# Glossar

*Dieses Glossar enthält Fachbegriffe aus dem Bereich des Softwaretests sowie ergänzende Begriffe aus dem Bereich der Softwaretechnik*[42]*, die einen Bezug zum Testen haben. Die Fachbegriffe sind im Text des Buches beim ersten Auftreten mit einem Pfeil gekennzeichnet; im Glossar wurden die ergänzenden Begriffe aus der Softwaretechnik unterstrichen.*

**Abnahmetest**
Test gegen die expliziten Anforderungen des Auftraggebers/Anwenders, wie sie in einem Anforderungsdokument (z.B. Pflichtenheft oder Fachkonzept) für beide Seiten verbindlich festgelegt sind, sowie gegen die impliziten Erwartungen des Auftraggebers, die dem allgemeinen Stand der Technik entsprechen. (Auch Systemtest aus Sicht des Auftraggebers.)

**Akzeptanztest**
1. Abnahmetest aus Sicht des Benutzers.
2. Teilmenge aller vorhandenen Testfälle, die das Testobjekt als Eintrittskriterium in eine bestimmte Teststufe bestehen muss.

**Alpha-Test**
Test/Erprobung einer Vorabversion eines Softwareprodukts durch repräsentative Kunden/Anwender in einer Testumgebung des Herstellers.

**Analysator**
Werkzeug, das formalisierte Prüfobjekte (z.B. Quellcode) auf bestimmte Eigenschaften oder Qualitätsmerkmale hin überprüft.

**analytische Qualitätssicherung**
Diagnostische Maßnahmen (z.B. Testen) zur Bestimmung der Qualität eines Produkts.

**Änderung**
Um- oder Neuformulierung eines freigegebenen Entwicklungs(zwischen)produkts (Dokument, Quellcode).

42. Grundlage für das Glossar waren folgende Quellen:
[Balzert 98], [Balzert 00], [BS7925-1], [Duden 99], [IEEE 610.12], [ISO 8402], [ISO 9126], [Liggesmeyer 90], [Link 02], [Myers 82], [Pol 00], [URL: GI TAV Glossar] [URL: SWT Glossar]

**Änderungsanforderung**
Schriftlich dokumentierter Wunsch bzw. Antrag auf Genehmigung, eine bestimmte Änderung an einem Entwicklungs(zwischen)produkt vornehmen zu dürfen.

**Änderungsauftrag**
Auftrag bzw. Genehmigung, eine bestimmte Änderung an einem Entwicklungs(zwischen)produkt vorzunehmen.

**Anforderung *(requirement)***
1. Bedingung oder Fähigkeit, die von einer Person zur Lösung eines Problems oder zur Erreichung eines Ziels benötigt wird.
2. Bedingung oder Fähigkeit, die eine Software erfüllen oder besitzen muss, um einen Vertrag, eine Norm oder ein anderes, formell bestimmtes Dokument zu erfüllen.
3. Aussage über eine zu erfüllende qualitative und/oder quantitative Eigenschaft eines Produkts.

**anforderungsbasierter Test**
Methode zur Herleitung oder Auswahl der Testfälle auf Grundlage der Anforderungen.

**Anforderungsdefinition**
1. Schriftliche Dokumentation der Anforderungen an ein Entwicklungs(zwischen)produkt.
2. Entwicklungsphase (des allgemeinen V-Modells), in der die Anforderungen an das zu erstellende Softwaresystem gesammelt, spezifiziert und verabschiedet werden.

**Anomalie**
Unstimmigkeit, die zu einer Fehlerwirkung führen kann, aber nicht zwingend dazu führt (anomal: unregelmäßig, regelwidrig).

**Anweisung**
Syntaktisch definierte Einheit einer Programmiersprache (z.B. Zuweisung an eine Variable).

**Anweisungsüberdeckung**
Kontrollflussbasiertes Testverfahren, das die mindestens einmalige Ausführung aller Anweisungen des Testobjekts fordert (auch als CO-Maß bezeichnet).

**Anwendungsfall *(use case)***
Eine durch genau einen Akteur angestoßene Folge von Systemereignissen, die für den Akteur ein Ergebnis produziert und an welchem weitere Akteure teilnehmen können.

**anwendungsfallbasierter Test**
Methode zur Herleitung oder Auswahl der Testfälle auf der Grundlage von Anwendungsfällen.

**Äquivalenzklasse**
Eine Menge von Werten, bei der jeder Wert
- als Eingabe für ein Testobjekt gleichartiges Sollverhalten zeigt (Äquivalenzklasse von Eingabewerten),
- als Ergebnis eines Testlaufs gleichartiges Sollverhalten aufzeigt (Äquivalenzklasse von Ausgabewerten).

**Äquivalenzklassenbildung**
Zerlegung des Wertebereichs der Eingabe- oder Ausgabedaten jeweils eines Parameters in Äquivalenzklassen.

**atomare Teilbedingung**
Eine Bedingung, die keine logischen Operatoren wie AND, OR oder NOT, sondern höchstens Relationssymbole wie »>« oder »=« enthält.

**Audit**
Eine systematische und unabhängige Untersuchung, um festzustellen, ob die (qualitätsbezogenen) Tätigkeiten und die damit zusammenhängenden Ergebnisse geplanten Vorgaben entsprechen und diese Vorgaben geeignet sind, die Ziele zu erreichen.

**Ausfall *(failure)***
1. Aussetzen der Ausführung einer festgelegten Aufgabe eines Systems aufgrund einer in ihm selbst liegenden Ursache und im Rahmen der zulässigen Beanspruchung.
2. Unvermögen eines Systems, eine verlangte Funktion auszuführen (s. a. Fehlerwirkung).

**Ausnahmebehandlung *(exception handling)***
Reaktion des Programms auf eine vorhergesehene und berücksichtigte Fehleingabe des Benutzers, falsche Parameterwerte oder auf ein Fehlverhalten des Programms an sich.

**äußerer Fehler *(failure)***
s. Fehlerwirkung

**Auswertung (der Tests)**
Anhand der Testprotokolle wird ermittelt, ob Fehlerwirkungen vorliegen; ggfs. wird eine Einteilung in Fehlerklassen vorgenommen.

**Back-to-back-Test**
Diversifizierender Test, bei dem die Programmversionen aus einer einzigen Spezifikation durch mehrfache Realisierung durch verschiedene Programmiergruppen entstehen.

**Beta-Test**
Test oder testweiser Betrieb einer Vorabversion eines Softwareprodukts durch repräsentative Kunden/Anwender in der Einsatzumgebung des Kunden/Anwenders.

**Black-box-Verfahren**
Alle Verfahren, die zur Herleitung oder Auswahl der Testfälle keine Information über die innere Struktur des Testobjekts benötigen.

**blockierter Test(fall)**
Zur Durchführung eingeplanter Testfall, der nicht ausgeführt werden kann, weil seine Voraussetzungen zur Ausführung nicht hergestellt werden können.

**codebasierter Test**
s. struktureller Test

**Datenflussanalyse**
Statisches Analyseverfahren zur Aufdeckung von fehlerhaften Zugriffssequenzen auf die Variablen des Testobjekts.

**Datenflussanomalie**
Unerwünschte oder nicht erwartete Folge von Operationen auf einer Variablen.

**datenflussbasierter Test**
Dynamischer Test, bei dem die Testfälle unter Berücksichtigung der Datenverwendungen im Testobjekt hergeleitet werden und die Vollständigkeit der Prüfung (Überdeckungsgrad) anhand der Datenverwendung bewertet wird.

**Debugger**
Analysewerkzeug des Entwicklers zur kontrollierten, zeilenweisen Abarbeitung eines Testobjekts.

**Debugging**
Tätigkeit des Lokalisierens und Behebens eines Fehlerzustands.

**Defekt *(fault)***
s. Fehlerzustand

**diversifizierender Test**
Prüft verschiedene Versionen eines Programms oder einer Komponente gegeneinander.

**Dummy**
Spezieller Platzhalter, ersetzt die echte Implementierung für Testzwecke.

**dynamischer Test**
Prüfung des Testobjekts durch Ausführung auf einem Rechner.

**Effizienz**
Messung der Verarbeitungsgeschwindigkeit bzw. Antwortzeit und des Ressourcenverbrauchs (Speicher u. a.) des Testobjekts für bestimmte Anwendungsfälle, in der Regel in Abhängigkeit steigender Last.

**einfache Bedingungsüberdeckung**
Kontrollflussbasiertes, dynamisches Testverfahren, das die Überdeckung der atomaren Teilbedingungen einer Entscheidung mit `wahr` und `falsch` fordert.

**endlicher Automat (auch Zustandsautomat, *finite state machine*)**
Besteht aus einer endlichen Anzahl interner Zustände. Zwischen den Zuständen gibt es Zustandsübergänge, die in Abhängigkeit von Eingaben oder Ereignissen durchlaufen werden. Eine Ausgabe oder Aktion kann bei jedem Zustandsübergang erfolgen oder in einem Zustand. Hat einen Anfangszustand und kann einen Endzustand besitzen.

**Entscheidungstabelle**
Erlaubt die Beschreibung auszuführender Aktionen in Abhängigkeit von durch »und« verknüpften Bedingungen.

**Entwicklertest**
Test, der in der (ausschließlichen) Verantwortung des Entwicklers bzw. der Entwicklungsgruppe des Testobjekts durchgeführt wird (oft mit Komponententest gleichgesetzt).

**Entwicklungsprozess**
s. Softwareentwicklungsmodell

**Extreme Programming**
Leichtgewichtiger, agiler Entwicklungsprozess, in dem u. a. Test-First-Entwicklung propagiert wird.

**Fehler**
1. Oberbegriff für Fehlhandlung, Fehlerzustand, Fehlerwirkung
2. Nichterfüllung einer festgelegten Anforderung

**Fehler aufdeckender Testfall**
Testfall, der bei Ausführung ein Istverhalten des Testobjekts erzeugt, das von dem geforderten Sollverhalten abweicht.

**Fehlerdatenbank**
Enthält aktuelle, vollständige Information über alle aufgedeckten Fehlerwirkungen.

**Fehlerfindungsrate**
Verhältnis zwischen Anzahl der aufgedeckten Fehlerwirkungen und definiertem Testaufwand (z.B. Zeitintervall, Zahl durchgeführter Testfälle, erreichte Abdeckung).

**Fehlerklasse**
Einteilung der aufgedeckten Fehlerwirkungen nach Schwere der Fehlerwirkung aus Sicht des Anwenders (z.B. Grad der Behinderung des Produkteinsatzes) .

**Fehlermanagement**
Systematische Verfolgung und Steuerung der Behebung von aufgedeckten Fehlern.

**Fehlermaskierung (*error*)**
Ein vorhandener Fehlerzustand wird durch einen oder mehrere andere Fehlerzustände in anderen Teilen des Testobjekts kompensiert, so dass dieser Fehlerzustand keine Fehlerwirkung hervorruft.

**Fehlernachtest**
Wiederholung aller Testfälle oder Testsequenzen, die eine Fehlerwirkung erzeugt haben; dient zum Nachweis, ob die Korrektur des ursächlichen Fehlerzustands erfolgreich war.

**Fehlerpriorität**
Festlegung der Dringlichkeit von Korrekturmaßnahmen unter Berücksichtigung der Fehlerklasse, des erforderlichen Korrekturaufwands und der Auswirkungen auf den gesamten Entwicklungs- und Testprozess.

**Fehlertoleranz**
Eine Fehlertoleranz liegt vor, wenn das System sein spezifiziertes Leistungsniveau bewahrt oder wiedererlangt, nachdem ein Fehlerzustand, ein falscher Schnittstellenaufruf oder eine falsche Bedienung durch den Anwender zu einer Fehlerwirkung geführt hat.

**Fehlerwirkung *(failure)***
1. Wirkung eines Fehlerzustands, die bei der Ausführung des Testobjekts nach »außen« in Erscheinung tritt.

2. Abweichung zwischen (spezifiziertem) Sollwert und (beobachtetem) Istwert (bzw. Soll- und Istverhalten).

**Fehlerzustand *(fault)***

1. Inkorrektes Teilprogramm, inkorrekte Anweisung oder Datendefinition, die Ursache für einen äußeren Fehler ist.
2. Zustand eines (Software)produkts oder einer seiner Komponenten, der unter spezifischen Bedingungen (z.B. bei einer hohen Belastung) eine geforderte Funktion des Produkts beeinträchtigen kann bzw. zu einer Fehlerwirkung führt.

**Fehlhandlung *(error)***

1. Die menschliche Handlung (des Entwicklers), die zu einem Fehlerzustand in der Software führt.
2. Eine menschliche Handlung (des Anwenders), die ein unerwünschtes Ergebnis (im Sinne von Fehlerwirkung) zur Folge hat (Fehlbedienung).
3. Unwissentlich, versehentlich oder absichtlich ausgeführte Handlung oder Unterlassung, die unter gegebenen Umständen (Aufgabenstellung, Umfeld) dazu führt, dass eine geforderte Funktion eines Produkts beeinträchtigt ist.

**Feldtest**

Erprobung einer Vorabversion eines Softwareprodukts durch einen (repräsentativ) ausgewählten Kundenkreis mit dem Ziel, Einflüsse aus nicht vollständig bekannten oder nicht spezifizierten Einsatzumgebungen zu erkennen und/oder die Marktakzeptanz zu prüfen.

**funktionale Anforderung**

s. Funktionalität

**funktionaler Test**

Dynamischer Test, bei dem die Testfälle unter Verwendung der funktionalen Spezifikation des Testobjekts hergeleitet werden und die Vollständigkeit der Prüfung (Überdeckungsgrad) anhand der funktionalen Spezifikation bewertet wird.

**Funktionalität**

Spezifiziert das Verhalten, das das System erbringen muss; beschreibt »was« das System leisten soll. Umsetzung ist Voraussetzung dafür, damit das System überhaupt einsetzbar ist. Merkmale der Funktionalität sind nach [ISO 9126] Angemessenheit, Richtigkeit, Interoperabilität, Ordnungsmäßigkeit, Sicherheit.

**Funktionstest**

s. funktionaler Test

**geschäftsprozessbasierter Test**
Methode zur Herleitung oder Auswahl der Testfälle auf Grundlage von Geschäftsprozessen.

**Grenzwertanalyse**
Fehlerorientiertes Testverfahren, verlangt die Wahl von Testfällen, die auf bzw. knapp inner- und außerhalb der Randbereiche der Äquivalenzklassen liegen.

**GUI (*graphical user interface*)**
Grafische Bedienoberfläche

**Individualsoftware**
Softwareprodukt, das für einen einzelnen Kunden entwickelt wird (s. a. Standardsoftware).

**informelles Review**
Review ohne festgelegten formalen Ablauf.

**innerer Fehler**
s. Fehlerzustand

**Inspektion**
Manuelle Prüfmethode mit definiertem Ablauf, die nach der individuellen Vorbereitung der Gutachter in einer Teamsitzung schwerwiegende Defekte in einem schriftlich vorliegenden Prüfobjekt identifiziert sowie Verbesserungen für den Entwicklungs- und Inspektionsprozess vorschlägt.

**Instrumentierung**
(Werkzeuggestütztes) Einfügen von Protokoll- oder Zählanweisungen in den Quell- oder Objektcode eines Testobjekts.

**Integration**
Verbinden von Software- und/oder Hardware-Komponenten zu größeren Baugruppen oder zum Gesamtsystem.

**Integrationstest**
Test mit dem Ziel, Fehler in Schnittstellen und im Zusammenspiel zwischen integrierten Komponenten zu finden.

**intuitive Testfallermittlung**
Methode zur Herleitung oder Auswahl der Testfälle, gestützt auf Erfahrung und Wissen des Testers.

**Istverhalten**
Im Test beobachtetes Verhalten des Testobjekts.

**Klassentest**
Test einer oder mehrerer Klassen eines objektorientierten Softwaresystems.

**Komparator**
Werkzeug zum automatischen Vergleich der Soll- und Istergebnisse.

**Komponente**
1. Kleinste Softwareeinheit, für die eine separate Spezifikation verfügbar ist (Hinweis: Komponente wird unterschiedlich definiert, diese Definition entspricht [BS7925-1]).
2. Softwareeinheit, die die Implementierungsstandards eines Komponentenmodells (EJB, CORBA, .NET) erfüllt.

**Komponententest**
Test einer Softwareeinheit (s. a. Komponente).

**Konfiguration**
1. Menge von Softwareobjekten einer bestimmten Version, die zusammen ein (Teil)System bilden.
2. Zustand der Umgebung eines Testobjekts, der als Vorbedingung für die Durchführung von Testfällen erreicht sein muss.

**Konfigurationsmanagement**
Aktivitäten zur Verwaltung der Konfigurationen.

**Konfigurationsobjekt**
Softwareobjekt oder Testumgebung, die der Konfigurationsverwaltung unterliegt.

**konkreter Testfall**
Testfall mit konkreten Werten für die Testdaten (s. a. logischer Testfall).

**konstruktive Qualitätssicherung**
Einsatz von Methoden, Werkzeugen, Richtlinien usw., die dazu beitragen, dass
- der Erstellungsprozess und/oder das zu erstellende Produkt a priori bestimmte Eigenschaften besitzt,
- Fehlhandlungen vermieden oder verringert werden.

**Kontrollfluss**
Abstrakte Repräsentation von allen möglichen Reihenfolgen von Ereignissen während einer Programmausführung, Repräsentation der Kontrollstruktur (meist grafisch dargestellt in einem Kontrollflussgraphen).

**Kontrollflussanomalie**
Statisch feststellbare Unstimmigkeit beim Ablauf des Testobjekts (z.B. nicht erreichbare Anweisungen).

**kontrollflussbasierter Test**
Dynamischer Test, bei dem die Testfälle unter Berücksichtigung des Kontrollflusses des Testobjekts hergeleitet werden und die Vollständigkeit der Prüfung (Überdeckungsgrad) anhand des Kontrollflussgraphen bewertet wird.

**Kontrollflussgraph**
Ein gerichteter Graph G = (N, E, nstart, nfinal). N ist die endliche Menge der Knoten. E ist die Menge der gerichteten Kanten. »nstart« ist der Startknoten. »nfinal« ist der Endknoten. Kontrollflussgraphen dienen zur Darstellung der Kontrollstruktur von Komponenten.

**Lasttest**
Messung des Systemverhaltens in Abhängigkeit steigender Systemlast (z.B. Anzahl parallele Benutzer, Anzahl Transaktionen).

**Linear Code Sequence and Jump (LCASJ)-Verfahren**
Dynamischer Test, bei dem die Testfälle unter Berücksichtigung der LCASJ des Testobjekts hergeleitet werden und die Vollständigkeit der Prüfung (Überdeckungsgrad) anhand der LCASJ bewertet wird. LCASJ ist eine Folge von Anweisungen, die sequenziell ausgeführt wird und durch einen Sprung auf eine andere Anweisung als die folgende beendet wird.

**logischer Testfall**
Testfall ohne Angabe von konkreten Werten für Eingaben und Ausgaben; es werden meist Wertebereiche (Äquivalenzklassen) für die Eingaben (und Ausgaben) angegeben.

**Managementreview**
Review, bei dem Projektpläne und Entwicklungsprozesse als solches analysiert werden.

**Mangel**
Ein Mangel liegt vor, wenn eine gestellte Anforderung oder berechtigte Erwartung nicht angemessen erfüllt wird.

**Massentest**
s. Volumentest

**Mehrfachbedingungsüberdeckung**
Kontrollflussbasiertes, dynamisches Testverfahren, das die Überdeckung der atomaren Teilbedingungen einer Entscheidung mit `wahr` und `falsch` in allen Kombinationen fordert.

**Meilenstein**
Markiert einen Zeitpunkt im Softwareentwicklungsprozess, zu dem ein bestimmtes Arbeitsergebnis fertig gestellt sein soll.

**Metrik**
Größe zur Messung einer bestimmten Eigenschaft eines Programms oder einer Komponente. Die Ermittlung von Metriken ist Aufgabe der statischen Analyse.

**minimale Mehrfachbedingungsüberdeckung**
Kontrollflussbasiertes, dynamisches Testverfahren, das die Überdeckung der atomaren Teilbedingungen einer Entscheidung mit `wahr` und `falsch` fordert, wobei nur die Kombinationen zu berücksichtigen sind, die eine Änderung des Wahrheitswertes der Gesamtbedingung bewirken.

**Mock**
Enthält mehr Funktionalität als ein Dummy, das Einstellen von gewünschten Reaktionen ist möglich.

**Moderator**
Leiter und Hauptverantwortlicher einer Reviewsitzung.

**Modultest**
Test eines einzelnen Moduls eines modularisierten Softwaresystems.

**Monitor**
Werkzeug zur testbegleitenden Aufzeichnung von Daten und Messwerten wie z.B. Netzwerkverkehr, Datenbankbelastung.

**Nachbedingung**
Zustand des Testobjekts (und/oder der Umgebung), in dem sich das Testobjekt (oder die Umgebung) nach Ausführung eines Testfalls oder einer Testsequenz befindet.

**Negativtest**
(Funktionaler) Testfall mit Eingabewerten, die laut Spezifikation des Testobjekts unzulässig sind. Das Testobjekt soll robust reagieren, z.B. indem es die Werte abweist und eine entsprechende Ausnahmebehandlung durchführt.

**nicht funktionale Anforderung**
Beschreibt Attribute des Systemverhaltens, also wie gut bzw. mit welcher Qualität das System seine Funktion erbringen soll. Umsetzung beeinflusst stark, wie zufrieden der Kunde bzw. Anwender mit dem Produkt ist. Merkmale sind nach [ISO 9126]: Zuverlässigkeit, Benutzbarkeit, Effizienz.

**nicht funktionaler Test**
Prüfung der nicht funktionalen Anforderungen.

**Patch**
Ungeplante Softwarelieferung mit korrigierten Dateien zur ggf. provisorischen Behebung einzelner (meist die Testdurchführung blockierender) Fehlerzustände.

**Performanz**
Effizienz bzgl. Verarbeitungsgeschwindigkeit.

**Performanztest**
Prüfung der Verarbeitungsgeschwindigkeit bzw. Antwortzeit für bestimmte Anwendungsfälle, in der Regel in Abhängigkeit steigender Last.

**Pfad**
Alternierende Sequenz von Knoten und gerichteten Kanten eines Kontrollflussgraphen; vollständiger Pfad beginnt mit dem Knoten »nstart« und endet mit dem Knoten »nfinal« des Kontrollflussgraphen.

**Pfadüberdeckung**
Dynamisches, kontrollflussbasiertes Testverfahren, fordert die Ausführung aller unterschiedlichen vollständigen Pfade eines Testobjekts. In der Praxis durch die hohe Anzahl an Pfaden in der Regel nicht anwendbar.

**Pflege/Pflegeprozess**
Durchführung von Änderungen und Erweiterungen in Softwaresystemen, die bereits im Einsatz sind (s. a. Wartung/Wartungsprozess).

**Platzhalter *(stub)***
Platzhalter werden beim Komponenten- und Integrationstest benötigt, um noch nicht implementierte Komponenten für die Testdurchführung zu ersetzen bzw. zu simulieren.

**Point of Control (PoC)**
Schnittstelle, über die das Testobjekt mit Testdaten versorgt wird.

**Point of Observation (PoO)**
Schnittstelle, an der die Reaktionen und Ausgaben des Testobjekts beobachtet und aufgezeichnet werden.

**Produktivumgebung**
Beim Anwender oder Betreiber im Einsatz befindliche Hardware, auf der das System betrieben wird, und sonstige dort eingesetzte Software mit zugehörigen Datenbeständen.

**Prozessmodell**
s. Softwareentwicklungsmodell

**Prüfobjekt**
Dokument, das auf Fehler und Unstimmigkeiten untersucht wird.

**Prüfung**
1. Tätigkeit wie Messen, Untersuchen, Ausmessen von einem oder mehreren Merkmalen einer Einheit (Prüfobjekt) sowie Vergleichen mit festgelegten Forderungen, um festzustellen, ob Konformität für jedes Merkmal erzielt ist.
2. Oberbegriff für alle analytischen Qualitätssicherungsmaßnahmen unabhängig von Methode und Prüfobjekt.

**Qualität**
Gesamtheit der Merkmale und Merkmalswerte eines Produkts oder einer Dienstleistung, die sich auf deren Eignung beziehen, festgelegte oder vorausgesetzte Erfordernisse zu erfüllen.

**Qualitätsmerkmal**
Satz von Eigenschaften eines Softwareprodukts, anhand dessen seine Qualität beschrieben und beurteilt wird. Ein Softwarequalitätsmerkmal kann über mehrere Stufen in Teilmerkmale verfeinert werden.

**Qualitätssicherung**
Alle Tätigkeiten innerhalb des Qualitätsmanagements, die dazu dienen, den Nachweis zu erbringen, dass die Qualitätsanforderungen erfüllt sind.

**Randbedingung**
Teil der Vor- und Nachbedingung, der gegenüber der Durchführung des Testfalls invariant ist, also durch einen Testlauf nicht verändert wird.

**Regressionstest**
Erneuter Test eines bereits getesteten Programms bzw. einer Teilfunktionalität nach deren Modifikation, mit dem Ziel nachzuweisen, dass durch die vorgenommenen Änderungen keine Fehler eingebaut oder (bisher maskierte Fehler) freigelegt wurden.

**Release**
s. Konfiguration

**Review**
Manuelle Prüfmethode mit mehr oder weniger festgelegtem Ablauf, die nach einer individuellen Vorbereitung der Gutachter in einer Teamsitzung Stärken und Schwächen eines schriftlich vorliegenden Prüfobjekts identifiziert.

**reviewfähig**
Arbeiten an einem Dokument haben einen gewissen Abschluss gefunden und eine ausreichende Vollständigkeit erreicht, so dass dieses Dokument einem Review unterzogen werden kann.

**Robustheit**
Verhalten gegenüber Fehlbedienung, Fehlprogrammierung, Hardwareausfall usw. sowie Fehlerbehandlung und das Wiederanlaufverhalten (*recovery*).

**Robustheitstest**
s. Negativtest

**Rolle**
Beschreibung bestimmter Fähigkeiten, Qualifikationen und Tätigkeitsprofile in der Softwareentwicklung, die von am Projekt beteiligten Personen (Rollenträger) auszufüllen sind.

**sicherheitsrelevantes/-kritisches System**
Softwaresystem, bei dessen Versagen Menschenleben gefährdet sind oder andere hohe Verluste entstehen können.

**Simulator**
Werkzeug, mit dem die Einsatz- bzw. Produktivumgebung nachgebildet wird.

**Smoke-Test**
Automatisierter Test, der möglichst alle Hauptfunktionen des Testobjekts (Basisfunktionalität) auslöst und vorwiegend die Robustheit prüft.

**Softwareentwicklungsmodell/Softwareentwicklungsprozess**
Beschreibt einen festgelegten organisatorischen Rahmen der Softwareentwicklung. Festgelegt wird, welche Aktivitäten in welcher Reihenfolge von welchen Rollen zu erledigen sind und welche Ergebnisse dabei entstehen und wie diese in der Qualitätssicherung überprüft werden.

**Softwareobjekt**
Identifizierbares Teilergebnis des Softwareentwicklungsprozesses (Quellcodedatei, Dokument u.a.).

**Softwarequalität**
Gesamtheit der Merkmale und Merkmalswerte eines Softwaresystemsprodukts, die sich auf dessen Eignung beziehen, festgelegte oder vorausgesetzte Erfordernisse zu erfüllen.

**Sollverhalten**
Erwartete Ausgabewerte und erwartetes Verhalten des Testobjekts (für jeden Testfall festzulegen).

**Spezifikation**
Beschreibt möglichst (mathematisch) genau und vollständig das geforderte Verhalten (und/oder den geforderten Aufbau) eines Softwareobjekts. Dient Entwicklern als Grundlage für die Programmierung und Testern als Grundlage für die Herleitung von Testfällen mittels Blackbox-Verfahren.

**Standardsoftware**
Softwareprodukt, das für einen größeren Markt bzw. Massenmarkt entwickelt wird und in identischer Form an eine Vielzahl von Kunden geliefert wird (s. a. Individualsoftware).

**statische Analyse**
Überprüfung, Vermessung und Darstellung eines Programms oder einer Komponente (bzw. eines Dokuments, das eine formale Struktur hat).

**statischer Test**
s. statische Analyse

**Stellvertreter**
s. Platzhalter

**Stresstest**
Prüfung des Systemverhaltens bei Überlastung (z.B. durch Betrieb mit zu hohem Datenvolumen oder auch gezielte Fehlbedienung, s.a. Robustheit).

**struktureller Test**
Dynamischer Test, bei dem die Testfälle unter Berücksichtigung der Struktur des Testobjekts hergeleitet werden und die Vollständigkeit der Prüfung (Überdeckungsgrad) anhand von Strukturelementen (z.B. Zweige, Pfade, Daten) bewertet wird. Oberbegriff für kontroll- und datenflussbasierte Tests.

**stub**
s. Platzhalter

**Syntaxtest**
Verfahren zur Ermittlung der Testfälle, das bei Vorliegen einer formalen Spezifikation der Syntax der Eingaben angewendet werden kann.

**Systemtest**
Test eines integrierten Systems, um sicherzustellen, dass es spezifizierte Anforderungen erfüllt.

**technisches Review**
Formales Review, bei dem hauptsächlich Fachexperten als Gutachter herangezogen werden und die Zweckhaftigkeit des Prüfobjekts im Mittelpunkt steht.

**Test(en)**
1. Gesamter Prozess, ein Programm auf systematische Weise auszuführen, um die korrekte Umsetzung der Anforderungen nachzuweisen und um Fehlerwirkungen aufzudecken.
2. Jede einzelne (i. A. stichprobenartige) Ausführung des Testobjekts unter spezifizierten Bedingungen zum Zwecke des Überprüfens der (beobachteten) Ergebnisse im Hinblick auf gewisse gewünschte Eigenschaften.
3. Oberbegriff für alle Tätigkeiten und Phasen im Testprozess.

**Testabdeckung**
s. Überdeckungsgrad

**Testaufwand**
Zu veranschlagender bzw. ermittelter Bedarf an Ressourcen für den Testprozess.

**Testautomatisierung**
1. Einsatz von Softwarewerkzeugen zur Erstellung bzw. Programmierung von Testfällen mit dem Ziel, die Testfälle rechnergestützt wiederholt ausführen zu können.
2. Unterstützung aller Arbeiten im Testprozess durch entsprechende Softwarewerkzeuge.

**Testbarkeit**
1. Mühelosigkeit und Geschwindigkeit, mit der die Funktionalität und das Leistungsniveau des Systems (auch nach jeder Anpassung) getestet werden können.
2. Zugänglichkeit des zu testenden Systems für den Test (Aspekte sind: Offenheit der Schnittstellen, Dokumentationsqualität, Zerlegbarkeit in kleinere Einheiten, Nachbildbarkeit der Produktivumgebung in der Testumgebung).

**Testbasis**
Alle Dokumente, aus denen die Anforderungen ersichtlich werden, die an das Testobjekt gestellt werden, bzw. die Dokumentation, auf der die Herleitung oder Auswahl der Testfälle beruht.

**Testdaten/Testdatum**
Eingabe- und Zustandswerte für ein Testobjekt und die Sollwerte nach Ausführung des betreffenden Testfalls.

**Testdurchführung**
Ausführung der Testfälle bzw. Testszenarien (Aktivität im Testprozess).

**Tester**
1. Experte für Testdurchführung und Fehlerberichte, mit Wissen im Anwendungsbereich des jeweiligen Testobjekts.
2. Oberbegriff für die im Testprozess direkt tätigen Personen.

**Testergebnis**
1. Alle Dokumente, die im Rahmen eines Testlaufs erstellt werden (hauptsächlich das Testprotokoll und dessen Auswertung).
2. Freigabe oder Zurückweisung des Testobjekts (je nach Anzahl und Schwere der ermittelten Fehlerwirkungen).

**Testfall**
Umfasst folgende Angaben: die für die Ausführung notwendigen Vorbedingungen, die Menge der Eingabewerte (ein Eingabewert je Parameter des Testobjekts) und die Menge der erwarteten Sollwerte, die Prüfanweisung (wie Eingaben an das Testobjekt übergeben und Sollwerte abzulesen sind) sowie die erwarteten Nachbedingungen.

**Testfallexplosion**
Bezeichnung für den (mit der Anzahl der Parameter) exponentiell steigenden Testaufwand bei vollständigem Test.

**Testinfrastruktur**
Bestandteile, die notwendig sind, um die geplanten Testaktivitäten erledigen zu können. Hierzu gehören üblicherweise: Arbeitsplätze der Tester und deren Ausstattung, Testumgebungen sowie sonstige Werkzeuge (E-Mail, WWW, Texteditoren usw.).

**Testkonzept *(test plan)***
Dokument, das den Umfang, die Vorgehensweise, die Ressourcen und die Zeitplanung der intendierten Tests (inklusive aller Aktivitäten) beschreibt (Inhalt z. B. gem. [IEEE 829]).

**Testlauf**
Die Ausführung eines oder mehrerer Testfälle.

**Testmanagement**
1. Aktivitäten zur Planung, Durchführung, Auswertung und Steuerung des Testens.
2. Verantwortliche Person(engruppe) für den Test.

**Testmethode**
Planmäßiges, auf einem Regelwerk aufbauendes Vorgehen zur Herleitung oder Auswahl von Testfällen.

**Testmetrik**
Messbare Eigenschaft eines Testfalls, Testlaufs oder Testzyklus mit Angabe der zugehörigen Messvorschrift.

**Testobjekt**
Komponente, integriertes Teilsystem oder System (in einer bestimmten Version), das einem Test unterzogen wird.

**Testorakel**
Informationsquelle zur Ermittlung der jeweiligen Sollergebnisse eines Testfalls (kann beispielsweise die Anforderungsdefinition sein).

**Testphase**
Menge inhaltlich/thematisch zusammengehöriger Aktivitäten (innerhalb des Testprozesses), die der Erarbeitung eines Teilergebnisses dienen (z. B. Erstellung der Testspezifikation).

**Testplan *(test schedule)***
1. Zeitliche Planung der Testdurchführung (Zuordnung der Testfälle zu Testern und Festlegung des Durchführungszeitpunktes).
2. Verzeichnis aller Testfälle, in der Regel thematisch bzw. nach Testzielen gruppiert.

**Testplanung**
Aktivität im Testprozess zur Erstellung und Fortschreibung von Testkonzept und Testplan.

**Testprotokoll**
Schriftlich festgehaltenes Ergebnis eines Testlaufs oder einer Testsequenz (im Fall von automatisierten Tests meist vom Werkzeug erstellt). Aus dem Protokoll muss hervorgehen, welche Teile wann, von wem, wie intensiv und mit welchem Ergebnis getestet wurden.

**Testprotokollierung**
Aktivität im Testprozess zur Erstellung des Testprotokolls.

**Testprozess**
Umfasst alle Aktivitäten, die zur Planung, Spezifikation, Durchführung, Protokollierung und zur Auswertung der Tests in einem Projekt benötigt werden.

**Testrahmen (*test bed*)**
Sammlung aller Programme (u. a. Testtreiber und Platzhalter), die notwendig sind, um Testfälle auszuführen, auszuwerten und Testprotokolle aufzuzeichnen.

**Testroboter**
Werkzeug zur Durchführung von Tests, das offen gelegte bzw. zugängliche äußere Schnittstellen des Testobjekts, beispielsweise die GUI, mit Eingabedaten ansteuert und deren Reaktionen dort abliest.

**Testsequenz**
Aneinanderreihung mehrerer Testfälle, wobei Nachbedingungen des einen Tests als Vorbedingungen des folgenden Tests genutzt werden.

**Testskript**
Anweisungen zur automatischen Ausführung eines Testfalls oder einer Testsequenz (oder übergeordnete Ablaufsteuerung weiterer Testwerkzeuge) in einer geeigneten Programmiersprache.

**Testspezifikation**
1. Begründung der Auswahl der (logischen) Testfälle sowie die Beschreibung der (logischen) Testfälle in einem Dokument.
2. Aktivität im Testprozess.

**Teststrategie**
Aufteilung von Testaufwand über die zu testenden Teile und/oder zu erfüllende Qualitätsmerkmale des Testobjekts. Auswahl und Bestimmung der Reihenfolge (bzw. des Zusammenspiels) von einzelnen Testmethoden und der Reihenfolge ihrer Anwendung auf die verschiedenen Testobjekte. Festlegung der zu erreichenden Überdeckungsgrade bei jeder Testmethode.

**Teststufe**
Phase im (allgemeinen) V-Modell, in der bestimmte Testaufgaben zusammengefasst werden.

**Testszenario**
Zusammenstellung von Testsequenzen.

**Testtreiber (*driver*)**
Programm bzw. Werkzeug, das es ermöglicht, ein Testobjekt ablaufen zu lassen, mit Testdaten zu versorgen und Ausgaben/Reaktionen des Testobjekts entgegenzunehmen.

**Testumgebung**
Gesamtheit aller Hardware- und Softwarekomponenten (auch der Testrahmen), die notwendig sind, um Testfälle durchzuführen.

**Testverfahren**
s. Testmethode

**Testware**
Alle Dokumente und ggf. Werkzeuge, die während des Testprozesses erstellt bzw. benutzt werden. Alles soll während der Wartung und Pflege wieder benutzbar sein und muss daher übertragbar und aktualisierbar sein.

**Testziel**
1. Aufdeckung von Fehlern als allgemeines Ziel des Testens.
2. Aufdeckung spezifischer Fehlerwirkungen durch passende Testfälle.
3. Nachweis der Erfüllung bestimmter Anforderungen im/durch das Testobjekt als spezifisches Ziel eines oder mehrerer Testfälle.

**Testzyklus**
Durchlauf durch den fundamentalen Testprozess auf der Basis genau einer Version des Testobjekts, an dessen Ende Fehlerkorrektur- oder Änderungsaufträge für die Entwickler vorliegen.

**tote Programmanweisung *(dead code)***
kann nicht zur Ausführung gebracht werden.

**Tuning**
Änderung von Programmen oder Programmparametern und/oder Ausbau der Hardware zur Optimierung des Zeitverhaltens eines Hardware/Softwaresystems.

**Überdeckungsgrad**
Kriterium zur Beendigung des Tests, unterschiedlich je nach Testmethode, meist durch Werkzeuge zu ermitteln.

**Unit Test**
s. Komponententest

**unnützer Testfall/unnötige Tests**
Testfall, der zu einem bereits vorhandenen Testfall redundant ist und somit zu keinen neuen Erkenntnissen führt.

**Ursache-Wirkungs-Graph-Analyse**
Funktionsorientiertes, dynamisches Testverfahren, das die Spezifikation in eine grafische Form – den Ursache-Wirkungs-Graphen – umsetzt, aus dem über Zwischenschritte Testfälle abgeleitet werden.

**Validierung**
Prüfung, ob ein Entwicklungsergebnis die individuellen Anforderungen bezüglich einer speziellen beabsichtigten Nutzung erfüllt.

**Verifikation**

1. Prüfung, ob die Ergebnisse einer Entwicklungsphase die Vorgaben der Phaseneingangsdokumente erfüllen.
2. Mathematisch formaler Beweis der Korrektheit eines Programm(teil)s.

**Versagen**

Verhalten eines Testobjekts, das nicht den aufgrund des Verwendungszwecks spezifizierten Funktionen entspricht.

**Version**

Entwicklungsstand eines Softwareobjekts zu einem bestimmten Zeitpunkt. In der Regel durch eine Nummer angegeben (vgl. Konfiguration).

**V-Modell (allgemeines)**

Vorgehensmodell für die Softwareentwicklung mit Verifikations- und Validationsaktivitäten. Inzwischen existieren viele unterschiedliche Varianten des V-Modells (nicht zu verwechseln mit dem Vorgehensmodell des Bundes und der Länder, das auch mit dem Begriff V-Modell verbunden wird; s. [URL: VM-IESE], [URL: VM-scope]).

**vollständiger Test**

Berücksichtigt alle möglichen Eingabewerte und deren Kombinationen mit allen unterschiedlichen Vorbedingungen; in der Praxis nur äußerst selten wegen des erheblichen Aufwands durchführbar.

**Volumentest**

Test, bei dem große Datenmengen manipuliert werden (s. a. Stresstest und Lasttest).

**Vorbedingung**

Zustand des Testobjekts (und/oder der Umgebung), der gegeben sein muss, um einen Testfall oder eine Testsequenz ablaufen zu lassen.

**Vorgehensmodelle**

s. Softwareentwicklungsmodelle

**Walkthrough**

Manuelle, informale Prüfmethode, um Fehler, Defekte, Unklarheiten und Probleme in schriftlichen Dokumenten zu identifizieren. Der Autor präsentiert das Dokument in einer Sitzung den Gutachtern.

**Wartung / Wartungsprozess**

Beseitigung von bekannten Fehlerwirkungen, d. h. Lokalisierung und Behebung von Fehlerursachen, bei bereits in Betrieb befindlichen Softwaresystemen.

**White-box-Verfahren**
Alle Verfahren, die zur Herleitung oder Auswahl der Testfälle Information über die innere Struktur des Testobjekts benötigen (s. a. struktureller Test).

**Zufallstest**
Testverfahren, bei dem Testfälle zufallsgesteuert generiert werden.

**Zustandsautomat**
s. endlicher Automat

**zustandsbezogener Test**
Dynamischer Test, bei dem die Testfälle unter Berücksichtigung der Zustände und Zustandsübergänge des Testobjekts hergeleitet werden und die Vollständigkeit der Prüfung (Überdeckungsgrad) anhand der Zustände und Zustandsübergänge bewertet wird.

**Zustandsmodell (oder Zustandsdiagramm)**
Grafische Darstellungsform eines Zustandsautomaten. Die Zustände werden entweder als Kreise oder als Rechtecke dargestellt. Die Zustandsübergänge werden durch beschriftete Pfeile angegeben (Eingabe/Ausgabe).

**Zweigüberdeckung**
Dynamisches, kontrollflussbasiertes Testverfahren, fordert die Überdeckung aller Zweige des Kontrollflussgraphen einer Komponente; auch als C1-Maß bezeichnet.

**Zuverlässigkeit**
Die Gesamtheit der Eigenschaften eines Systems, die sich auf die Eignung zur Erfüllung gegebener Erfordernisse unter gegebenen Bedingungen für ein gegebenes Zeitintervall beziehen.

**zyklomatische Zahl**
Metrik zur Komplexität von Kontrollflussgraphen, gibt die Anzahl der linear unabhängigen Pfade eines Kontrollflussgraphen bzw. einer durch einen Kontrollflussgraphen dargestellten Komponente an.

# Quellenverzeichnis

## Literatur

[Balzert 98] Balzert, H.: Lehrbuch der Software-Technik, Band 2, Spektrum Akademischer Verlag, 1998.

[Balzert 00] Balzert, H.: Lehrbuch der Software-Technik, Band 1, 2. Auflage, Spektrum Akademischer Verlag, 2000.

[Beck 00] Beck, K.: Extreme Programming, Addison-Wesley, München, 2000.

[Boehm 73] Boehm, B.W.: »Software and its impact: a quantitative assessment«. In: Datmation, Vol. 19, No. 5, 1973, S. 48–59.

[Boehm 79] Boehm, B.W.: »Guidelines for Verifying and Validating Software Requirements and Design Specification«. In: Proceedings of Euro IFIP 1979, S. 711–719.

[Boehm 81] Boehm, B.W.: Software Engineering Economics, Englewood Cliffs, Prentice Hall, 1981.

[Bourne 97] Bourne, K.C.: Testing Client/Server Systems, McGraw-Hill, 1997.

[Bush 90] Bush, M.: »Software Quality: The use of formal inspections at the Jet Propulsion Laboratory«. In: Proceedings of the 12th ICSE, IEEE 1990, S. 196–199.

[Duden 99] Duden-Oxford-Großwörterbuch Englisch, 2. Auflage, Mannheim, 1999 (CD-ROM).

[Dustin 99] Dustin, E.; Rashaka, J.; Paul, J.: Automated Software Testing, Introduction, Management and Performance, Addison-Wesley, 1999.

[Fagan 76] Fagan, M.E.: »Design and Code Inspections to Reduce Errors in Program Development«. In: IBM Systems Journal, Vol. 15, No. 3, 1976, S. 182–211.

[Fenton 91] Fenton, N.E.: Software Metrics, Chapman & Hall, 1991.

[Fewster 99] Fewster, M.; Graham, D.: Software Automation, Effective use of test execution tools, Addison-Wesley, 1999.

[Frühauf 91] Frühauf, K.; Ludewig, J,; Sandmayr, H.: Software-Prüfung: eine Fibel, Verlag der Fachvereine, Zürich und Teubner Verlag, Stuttgart, 1991.

[Gilb 96] Gilb, T.; Graham, D.: Software Inspections, Addison-Wesley, 1996 (Nachdruck von 1993).

[Jacobson 99] Jacobson, I.; Booch, G.; Rumbaugh, J.: The Unified Software Developement Process, Addison-Wesley Longman, Amsterdam, 1999.

[Liggesmeyer 90] Liggesmeyer, P.: Modultest und Modulverifikation, State of the Art, BI Wissenschaftsverlag, Mannheim, Wien, Zürich, 1990.

[Link 02] Link, J.; Fröhlich, P.: Unit Tests mit Java: Der Test-first-Ansatz, dpunkt.verlag, 2002.

[McCabe 76] McCabe, T.J.: »A Complexity Measure«. In: IEEE Transactions on Software Engineering, Vol. SE-2, No. 4, 1976, S. 308–320.

[Mellis 96] Mellis, W.; Herzwurm, G.; Stelzer, D.: TQM der Softwareentwicklung, Vieweg, 1996.

[Möller 96] Möller, K.-H.: »Ausgangsdaten für Qualitätsmetriken – Eine Fundgrube für Analysen«. In: Ebert, C.; Dumke, R. (Hrsg.): Software-Metriken in der Praxis, Springer-Verlag, Berlin, 1996.

[Myers 82] Myers, G. J.: Methodisches Testen von Programmen, Oldenbourg, München, Wien, 1982 (7. Auflage 2001).

[Pol 00] Pol, M.; Koomen, T.; Spillner, A.: Management und Optimierung des Testprozesses, dpunkt.verlag, 2000 (2. Auflage 2002).

[Riedemann 97] Riedemann, E.H.: Testmethoden für sequentielle und nebenläufige Software-Systeme, Teubner-Verlag, Stuttgart, 1997.

[Royce 70] Royce, W.W.: »Managing the development of large software systems«. In: IEEE WESCON, Aug. 1970, S. 1–9 (Nachdruck in Proceedings of the 9th International Conference on Software Engineering, 1987, Monterey, CA., S. 328–338).

[Sneed 02] Sneed, H.M.; Winter, M.: Testen objektorientierter Software. Das Praxisbuch für den Test objektorientierter Client/Server-Systeme, Hanser, München, Wien, 2002.

[Spillner 00] Spillner, A.: »From V-model to W-model – Establishing the Whole Test Process«. In: Proceedings Conquest 2000 – Workshop on »Testing Non-Functional Software Requirements«, Sept. 2000, Nürnberg, S. 221–231.

[Thaller 00] Thaller, G.E.: Software-Test, Verifikation und Validation, Heise, 2000.

[Versteegen 99] Versteegen, G. (Hrsg.): Das V-Modell in der Praxis, dpunkt.verlag, Heidelberg, 1999.

[Wallmüller 90] Wallmüller, E.: Software-Qualitätssicherung in der Praxis, Hanser, 1990.

[Wiener 94] Wiener, L.R.: Digitales Verhängnis, Gefahr der Abhängigkeit von Computern und Programmen, Addison-Wesley,1994.

### Weitere empfohlene Literatur

Beizer, B.: Software Testing Techniques, Van Nostrand Reinhold, 1990.

Hetzel, W.C.: The Complete Guide to Software Testing, Collins, 1985.

Kaner, C.: Testing Computer Software, Van Nostrand Reinhold, 1993.

Kit, E.: Software Testing in the Real World, Addison-Wesley, 1995.

Ould, M. A.; Unwin, C. (eds.): Testing in Software Development, Cambridge University Press, 1986.

Perry, W.E.: Effective Methods for Software Testing, Wiley, 2000.

Roper, M.: Software Testing, McGraw-Hill, 1994.

Royer, T. C.: Software Testing Management, Prentice Hall, 1993.

### Normen und Standards

[BS 7925-1] British Standard 7925-1, Software testing, Part 1: Vocabulary, 1998.

[BS 7925-2] British Standard 7925-2, Software testing, Part 2: Software component testing, 1998.

[DIN 60601-1-4] DIN EN 60601-1-4, Ausgabe: 2001-04, Medizinische elektrische Geräte – Teil 1-4: Allgemeine Festlegungen für die Sicherheit; Ergänzungsnorm: Programmierbare elektrische medizinische Systeme.
[DIN 66271] DIN 66271:1995, Informationstechnik – Software-Fehler und ihre Beurteilung durch Lieferanten und Kunden.
[IEEE 610.12] IEEE Std 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology (Rev. and redisignation of IEEE Std 729–1983).
[IEEE 730] IEEE Std 730-1998, Standard for Software Quality Assurance Plans.
[IEEE 828] IEEE Std 828-1990, Standard for Software Configuration Management Plans.
[IEEE 829] IEEE Std 829-1998, IEEE Standard for Software Test Documentation (Rev. of IEEE Std 829-1983).
[IEEE 1028] IEEE Std 1028-1997, IEEE Standard for Software Reviews.
[IEEE 1042] IEEE Std 1042-1987, IEEE Guide to Software Configuration Management.
[IEEE 1044] IEEE Std 1044-1993, IEEE Standard Classification for Software Anomalies.
[ISO 8402] ISO 8402:1994, Qualitätsmanagement und Qualitätssicherung – Begriffe (ersetzt durch [ISO 9000]).
[ISO 9000] ISO 9000:2000, Qualitätsmanagementsysteme – Grundlagen und Begriffe.
[ISO 9126] ISO/IEC 9126:1991, Bewerten von Softwareprodukten, Qualitätsmerkmale und Leitfaden zu ihrer Verwendung (identisch mit [DIN 66272, 94]).
[VDE 3694] VDI/VDE-Richtlinie 3694, 1991-04, Lastenheft/Pflichtenheft für den Einsatz von Automatisierungssystemen.

### Weitere Normen und Standards

IEEE Std 729-1983 IEEE Standard Glossary of Software Engineering Terminology (ersetzt durch IEEE Std 610.12-1990).
IEEE Std 982.2-1988 IEEE Guide for the Use of IEEE Standard Dictionary of Measures to Produce Reliable Software.
IEEE Std 1008-1987 IEEE Standard for Software Unit Testing.
ISO/IEC 9126-1:2001-06 Software-Engineering – Qualität von Software-Produkten – Teil 1: Qualitätsmodell.

### WWW-Seiten[43]

[URL: ASQF] *http://www.asqf.de* Arbeitskreis Software-Qualität Franken e. V.
[URL: BCS] *http://www.bcs.org* British Computer Society (BCS), vergleichbar mit der Gesellschaft für Informatik e. V. (GI) in Deutschland.
[URL: dPunkt] *http://www.dpunkt.de/certified-tester* WWW-Seite zum Buch
[URL: GI] *http://www.gi-ev.de* Gesellschaft für Informatik e. V.

43. Die Gültigkeit der angegebenen URLs wurde mit Drucklegung überprüft. Eine Garantie über deren Gültigkeit über dieses Datum (August 2002) hinausgehend, kann nicht übernommen werden.

[URL: Graham] *http://www.grove.co.uk/About_Grove/Dorothy.html* Homepage von Dorothy Graham, A.B., M.Sc.

[URL: GTB] *http://www.certified-tester.de/board* German Testing Board

[URL: GTB CTF 01] *http://www.certified-tester.de/lehrplan.php* Lehrplan »Grundlagen des Testens«, ASQF-Certified-Tester, Foundation Level, Stand: 1. 11. 2001 (bzw. neueste Version)

[URL: imbus 99] *http://www.imbus.de/forschung/pie24306/gui/aquis-full_paper-1.3.html* »How to Automate Testing of Graphical User Interfaces, Results of the ESSI PIE 24306«, imbus AG, 1999

[URL: imbus-downloads] *http://www.imbus.de/produkte/download/dl_whitepapers.html* Nützliche Artikel und Unterlagen der imbus AG zum Herunterladen

[URL: ISEB] *http://www.iseb.org.uk* Information Systems Examination Board (ISEB)

[URL: ISEB CTF 99] *http://www.iseb.org.uk* -> Qualifikation Areas -> Software Testing ISEB Foundation Syllabus V2.0 – 25 February 1999 oder direkt *http://www1.bcs.org.uk/DocsRepository/00900/913/Syll.htm* oder als PDF-Datei *http://www1.bcs.org.uk/DocsRepository/00900/913/docs/syllabus.pdf*

[URL: ISEB CTP 01] *http://www.iseb.org.uk* -> Qualifikation Areas -> Software Testing ISEB Practitioner Syllabus V1.1 – 04 September 2001 oder direkt *http://www1.bcs.org.uk/DocsRepository/00900/913/Syll.htm* oder als PDF-Datei *http://www1.bcs.org.uk/DocsRepository/00900/913/docs/practsyll.pdf*

[URL: ISTQB] *http://www.istqb.org* International Software Testing Qualification Board

[URL: NIST Report] *http://www.nist.gov/director/prog-ofc/report02-3.pdf* »The Economic Impacts of Inadequate Infrastructure for Software Testing«, National Institute of Standards & Technology, USA, May 2002.

[URL: Parnas] *http://www.crl.mcmaster.ca/SERG/parnas.homepg* Homepage von Dr. David Lorge Parnas, P.Eng., Ph.D., Dr. h.c., FRSC, FACM, Dr. h.c.

[URL: Pol] *http://www.polteq.com/pol.htm* Homepage von Martin Pol

[URL: VM-IESE] *http://www.iese.fhg.de/VModell/* Vorgehensmodell des Bundes und der Länder, Fraunhofer IESE – Institut für Experimentelle Softwareentwicklung, Kaiserslautern

[URL: VM-scope] *http://www.scope.gmd.de/vmodell/* Vorgehensmodell des Bundes und der Länder, (ehemals) GMD – Forschungszentrum Informationstechnik GmbH, Projektgruppe »Software Quality and Productivity«.

**Weitere nützlichen WWW-Seiten:**

[URL: BCS SIGIST] *http://www.testingstandards.co.uk/* The BCS SIGIST Standards Working Party

[URL: DIN] *http://www2.din.de/* Deutsches Institut für Normung e. V.

[URL: GI TAV] *http://www.fbe.hs-bremen.de/spillner/gi.htm* Gesellschaft für Informatik e. V., Fachgruppe TAV (2.1.7): Test, Analyse und Verifikation von Software

[URL: GI TAV Glossar] *http://www.fbe.hs-bremen.de/spillner/begriffe/start.html*

[URL: ISO] *http://www.iso.ch/* The International Organization for Standardization (ISO), Normenkatalog

[URL: SEPT] *http://www.12207.com/index.html* SEPT, Supplying Software Engineering Standards Information to the World

[URL: SWT Glossar] *http://www.software-technik.de/microsoft/glossar/* Glossar zu Begriffen der Softwaretechnik von Prof. Dr. H. Balzert

# Index


## A

Abnahmetest 41, 64, 65, 78
Abnahmeumgebung 65
Ad-hoc-Integration 56
Akkreditierung 3
Allgemeines V-Modell 39, 75, 77
Alpha-Test 67
Analysatoren 91, 93
Analyse 79, 93
    statische 14, 91, 93
Analysewerkzeug 93
Analysierbarkeit 14
Änderbarkeit 12, 14
Änderungsanforderung 165
Änderungsauftrag 153
Änderungskontrollausschuss 163
Anforderung 8, 10, 17, 24, 33, 40, 59, 60, 62
    funktionale 59
    nicht funktionale 59
anforderungsbasiertes Testen 169
Anforderungsdefinition 40
Angemessenheit 12
Anomalie 94, 95
Anpassbarkeit 14
Antwortzeit 174
Anweisung 129
Anweisungsüberdeckung 129
Anwender 59
Äquivalenzklassen 104, 105
Äquivalenzklassenbildung 104, 108, 114, 119
ASQF (Arbeitskreis Software Qualität Franken e.V.) 3
atomare Teilbedingung 134
Audit 165
Ausfall 9, 33
Auslieferung 57, 67
Ausnahmebehandlung 24, 46
äußerer Fehler 9
Austauschbarkeit 14
Auswahl 176, 179
Auswahlkriterien 179
Auswahlprozess 179
Auswertung der Tests 26
Autor 84, 86, 99

## B

Back-to-back-Test 26, 30
BCS (British Computer Society) 2
Bedienbarkeit 13
Bedienoberfläche 172
    grafische 173
Bedienungsüberdeckung
    einfache 134
    Mehrfach- 135
    minimale Mehrfach- 136
Benutzbarkeit 12, 13
Beta-Test 67
Black-box-Verfahren 23, 104
blockierter Test 154
Bottom-up-Integration 56

## C

Capture-and-Replay-Tool 172
CASE-Tool 169
CAST-Tool 169
Certified Tester 4, 36, 152
Checklisten 89
Checkpunkt 174
Client/Server-System 174
codebasierter Test 128

## D

Datenflussanalyse 94
Datenflussanomalie 94
datenflussbasierter Test 17
Debugger 172
Debugging 10, 12

Defekt 9, 10, 12, 49, 69, 70
direkte Fehlerkosten 72
Dummy 101
dynamischer Test 101

**E**

Effizienz 12, 13, 47
einfache Bedingungsüberdeckung 134
endlicher Automat 120
Entscheidungstabelle 127
Entwicklertest 34
Entwicklungsprozess 18
Erlernbarkeit 13
Evaluation 179
Exception Handling 46

**F**

Fehler 9, 16, 17, 26
  äußerer 9
  innerer 9
Fehler aufdeckender Testfall 154
Fehleranalyse 72, 161, 164
fehlerbasierte Metriken 154
Fehlerbegriff 9
Fehlerbehandlung 62
Fehlerdatenbank 159
Fehlerfindungsrate 28
Fehlerklasse 26, 154
Fehlerklassifikation 159
Fehlerkorrektur 161
Fehlerkorrekturkosten 72
Fehlerkosten 72
  direkte 72
  indirekte 72
Fehlermanagement 156, 163, 167
Fehlermanagementwerkzeug 170
Fehlermaskierung 10
Fehlermeldung 67, 157, 170
Fehlerpriorität 160
Fehlerschwere 32
Fehlerstatus 157, 161, 165
Fehlerstatusmodelle 170
Fehlertoleranz 13
Fehlerursache 17, 172
Fehlerverfolgung 4
Fehlerwirkung 9, 26, 28, 32, 35, 42, 46, 53, 54, 70
Fehlerzustand 9, 10, 52
Fehlhandlung 10
Feldtest 66
funktionale Anforderung 59
Funktionalität 9, 12, 13

**G**

German Testing Board 3
Geschäftsprozessanalyse 61
geschäftsprozessbasierter Test 11, 61
GI (Gesellschaft für Informatik e.V.) 3
grafische Bedienoberfläche 173
Grenzwertanalyse 115, 119, 171
GUI (graphical user interface) 173
Gutachter 83, 84, 86, 87, 89, 90

**I**

Implementierung 40, 45
indirekte Fehlerkosten 72
informelles Review 90
innerer Fehler 9
Inspektion 80, 89, 99
Inspektor 86, 89
Installierbarkeit 14
Instrumentierung 144
Integrationsstrategie 54
Integrationstest 41, 49, 51, 52, 57, 77, 78, 150
ISTQB (International Software Testing Qualification Board) 2, 3
Interoperabilität 12, 13
intuitive Testfallermittlung 144
ISEB (Information Systems Examination Board) 2, 3
Istverhalten 10, 37

**K**

Klasse 40, 42
Klassentest 11, 42
Komparator 175, 176
Komponenten 33
Komponentenspezifikation 40
Komponententest 16, 41, 42, 43, 44, 45, 48, 49, 52, 54, 78, 149
Konfiguration 165
Konfigurationsaudit 165

Konfigurationsmanagement 163, 164
Konfigurationsobjekt 164
Konfigurationsverwaltung 164
Konformität 14
konkreter Testfall 23
Kontrollfluss 14
Kontrollflussanalyse 96
Kontrollflussanomalie 96
kontrollflussbasierter Test 14, 17
Kontrollflussgraph 14
Konventionen 93
Kostenaspekte 72
Kostendeckung 178
Kriterienkatalog 179
Kunde 59, 61, 68

**L**

Lasttest 11, 62
Linear Code Sequence and Jump (LCASJ)-Verfahren 143
logischer Testfall 23

**M**

Management der Testarbeiten 153
Managementreview 87
Mangel 9
Maße 27
Massentest 62
Mehrfachbedingungsüberdeckung 135
Meilenstein 18
Metriken 27, 92, 96, 97, 154
  fehlerbasierte 154
  testfallbasierte 154
  testobjektbasierte 154
minimale Mehrfachbedingungs-überdeckung 136
Mitarbeiterqualifikation 151
Mock 101
Moderator 83, 84, 86, 99
Modifizierbarkeit 14
Modul 40, 42
Modultest 42
Monitor 174

**N**

Negativtest 171
nicht funktionale Anforderungen 59
nicht funktionaler Test 13
Normen 36, 165

**O**

objektorientiertes Softwaresystem 18
Ordnungsmäßigkeit 12

**P**

Patch 160
Performanztest 62
Pfade 94
Pfadüberdeckung 129, 138
Pflegeprozess 20
Phase 42, 75
Phasenergebnis 41, 75
Point of Control 103
Point of Observation 102
Priorisierung 32
Priorisierung der Tests 22, 32, 33, 34, 37
Produktivumgebung 58, 66
Produktmanagement 163
Produktnorm 165
Programmanweisungen
  tote 27
Programmierung 40, 45
Projekt 16, 34
Projektmanagement 163
Projektmanager 55
Projektplan 55
Projektplanung 19
Protokollant 86, 99
Prozessnorm 165
Prüfung 3, 8, 23
Prüfung auf Änderbarkeit/Wartbarkeit 63
Prüfung der Dokumentation 62
Prüfung zum Certified Tester 3, 4
Psychologie des Testens 34

**Q**

Qualifikation 152
Qualifikationsprofile 151

Qualitätsmerkmal 12, 33
Qualitätssicherungsplan 76

**R**

Randbedingung 10, 14
Regressionstest 70, 164, 170, 178
Regressionstestfähigkeit 178
Reife 13
Requirements 169
Ressourcen 153
Review 18, 80, 95, 99
    Einführung 82
    informelles 90
    Management- 87
    Nachbereitung 85
    technisches 89
    Vorbereitung 83, 88, 99
Reviewarten 87
Reviewer 86
reviewfähiger Status 82
Reviewplanung 82
Reviewsitzung 83, 85
Reviewteam 82, 85
Richtigkeit 12
Risiko 16, 17
Robustheitstest 171
Rollen 19, 85

**S**

Sicherheit 12
Simulator 172
Skript 173
Smoke-Test 127
Software
    Versagen der 13
Softwareentwicklungsmodelle 18
Softwarequalität 12
Softwarewartung 67
Sollverhalten 10, 30, 37
Sollwerte 30
Spezifikation 17, 23, 37, 41, 48
    formal 171
Stabilität 14
Standards 93, 165
statische Analyse 14, 91, 93
statischer Test 52, 79
Stichprobe 36
Stresstest 62
strukturierte Gruppenprüfungen 79
stub 101
Syntaxtest 127
Systemanforderungen 169
Systemarchitektur 55
Systeme
    sicherheitsrelevante 16
Systementwurf
    funktionaler 40
    technischer 40
Systemtest 11, 16, 41, 57, 66, 67, 77, 150

**T**

technisches Review 89
Teilbedingung
    atomare 134
Test 10
    Alpha- 67
    anforderungsbasierter 169
    Auswertung 26
    Back-to-back- 26, 30
    Beta- 67
    blockierter 154
    codebasierter 128
    datenflussbasierter 17
    dynamischer 101
    Entwickler- 34
    Feld- 66
    geschäftsprozessbasierter 11, 61
    Komponenten- 16, 41, 42, 43, 44, 45, 48, 49, 52, 54, 78, 149
    kontrollflussbasierter 14, 17
    Last- 11, 62
    Massen- 62
    Modul- 42
    Negativ- 171
    nicht funktionaler 13
    Performanz- 62
    Priorisierung des 22, 32, 33, 34, 37
    Regressions- 70, 164, 170, 178
    Robustheits- 171
    Smoke- 127
    statischer 52, 79
    Stress- 62
    Syntax- 127
    System- 11, 16, 41, 57, 66, 67, 77, 150

Unit- 42
unnötiger 17
vollständiger 14, 21
Volumen- 62
Zufalls- 127
zustandsbezogener 119
Test auf Benutzerakzeptanz 65
Test auf Benutzungsfreundlichkeit 62
Test auf Kompatibilität/ Datenkonversion 62
Test auf Robustheit 62
Test auf vertragliche Akzeptanz 65
Test der Stabilität/Zuverlässigkeit 62
Test der (Daten)Sicherheit 62
Test neuer Produktversionen 67
Test unterschiedlicher Konfigurationen 62
Testabdeckung 175
Testablauf 171
Testadministrator 151
Testarbeiten
Management 153
Testaufwand 14, 15, 16, 178
Testaufzeichnung 173
Testauswertung 20, 164
Testautomatisierer 151
Testbasis 23, 60, 61, 170
Testbegriff 10
Testbericht 165
Testdaten 10
Testdatengenerator 170, 171
Testdesigner 151
Testdienstleister 149
Testdokumentation 176
Testdurchführung 20, 25, 26, 175, 176
Werkzeuge zur 171
Testen 8
Testende 26, 27
Testendekriterium 14, 119, 155
Tester 9, 25, 30, 32, 34, 35, 149, 151
Testergebniss 153
Testfall 10, 23, 24, 25, 31, 32, 35, 36, 37, 109, 170, 174
Fehler aufdeckender 154
konkreter 23
logischer 23
unnützer 114
testfallbasierte Metriken 154
Testfallermittlung
intuitive 144
Testfallexplosion 18
Testfortschritt 154, 163
Testgenerator 170
Testinfrastruktur 21, 23
Testkonzept 21, 55, 77, 153, 166
Testkosten 73
Testlauf 10, 175
Testmanagement 10, 21, 149, 161, 169
Testmanager 17, 18, 36, 54, 55, 74, 78, 151, 169, 170
Testmethode 11, 22
Testmetriken 154
Testobjekt 10, 11, 23, 35, 42, 51, 52, 57, 170, 173, 174, 175
testobjektbasierte Metriken 154
Testorakel 30, 37
Testorganisation 149
Testplan 23, 155, 170
Testplanung 20, 21, 76, 151, 169, 170, 177
Testprotokoll 26, 156
Testprotokollierung 20, 26, 151
Testprozess 10, 18, 30, 37
Testrahmen 23, 26, 172
Testressourcen 32
Testroboter 172, 175
Testskript 173
regressionstestfähiges 173
Testspezifikation 20, 23
Teststatusbericht 154
Teststeuerung 151
Teststrategie 21, 22, 48, 75
Teststufe 11, 19, 57, 77
Testszenario 11, 61
Testteam 35, 149
Testtechniken 8
Testtreiber 172, 176
Testumgebung 43, 52, 57
Testverfahren 17
Testware 21, 29
Testwerkzeug 169
Einführung 176
Typen 169
Testziel 11, 45, 52

Testzyklus 29, 178
-planung 153
-steuerung 155
-überwachung 154
Top-down-Integration 55
tote Programmanweisungen 27
Transaktion 174

## U

Überdeckung 175
Überdeckungsanalysator 175
Überdeckungsanalyse 175
Überdeckungsgrad 22
Übertragbarkeit 12, 14
Überwachung 154
Umgebung 56, 66
Unified Modeling Language 31
Unit 40, 42
Unit Test 42
unnötiger Test 17
unnützer Testfall 114
Update 70
Ursachen 9
Ursache-Wirkungs-Graph-Analyse 127

## V

Validierung 41
Verantwortlichkeiten 85
Verifikation 41
Versagen der Software 13
Version 164, 165
Versionenverwaltung 164
Verständlichkeit 13
V-Modell 18, 19, 77
Allgemeines 39, 75, 77
vollständiger Test 14, 21
Volumentest 62
Vorgehensmodell 18, 19, 39, 76, 78
Vorgehensmodell des Bundes und der Länder 18, 20

## W

Walkthrough 88, 89, 99
Wartungsprozesses 20
Wasserfall-Modell 18, 19
Werkzeug 175, 176
Wirtschaftlichkeit 177
Werkzeugauswahl 179
Werkzeuge zur Testdurchführung 171
Werkzeugeinführung 180
Werkzeugtyp 169, 174
Werkzeugunterstützung 144
White-box-Verfahren 23, 128
Wiederherstellbarkeit 13
Wirtschaftlichkeitsaspekte 72

## Z

Zertifikat 2
Zufallstest 127
Zustandsautomat 120
zustandsbezogener Test 119
Zustandsmodell 119
Zuverlässigkeit 12, 13
Zweigüberdeckung 129, 131

**dpunkt.**verlag im Internet: www.dpunkt.de

**Andreas Spillner • Tilo Linz**

# Basiswissen Softwaretest

Ob Geldautomat, Navigationssystem oder Mobiltelefon: Software bestimmt die Funktion von Geschäftsabläufen, Geräten und technischen Anlagen aus den unterschiedlichsten Lebensbereichen. Als Nutzer dieser Systeme sind wir immer stärker davon abhängig, dass Software trotz zunehmender Komplexität unter allen Bedingungen einwandfrei und zuverlässig funktioniert.

Professionelles Prüfen und Testen von Software wird daher zu einer immer wichtigeren Aufgabe, die eine fundierte Ausbildung erfordert. Mit dem Certified-Tester-Programm entsteht ein international standardisiertes Aus- und Weiterbildungsschema für Softwaretester. In diesem Buch wird das Grundlagenwissen im Bereich Softwaretest beschrieben und anhand eines durchgängigen Beispiels erklärt. Es beschreibt die Grundsätze des Testens, alle Aktivitäten des Testens innerhalb des Entwicklungsprozesses, Techniken des Testens, Testmanagement und Testwerkzeuge. Der Inhalt entspricht dem Stoff der Prüfung zum Certified Tester (Foundation Level) nach ASQF- und ISEB-Standard. Das Buch ist so aufbereitet, dass es für das Selbststudium geeignet ist.

Auf der Website zum Buch www.dpunkt.de/certified-tester/ befinden sich Übungsaufgaben mit Lösungen, aktuelle Informationen zum Certifed-Tester-Programm und eine Auswahl an Testwerkzeugen zum Herunterladen.

*Mit Geleitworten von Dorothy Graham, David Parnas und Martin Pol.*

ISBN 3-89864-178-3

Thema
- **Softwaretest**
- **Qualitätssicherung**
- **Softwareentwicklung**
- **Programmierung**

Leser
- **Softwaretester**
- **Programmierer / Softwareentwickler**
- **Projektleiter**
- **Lehrende und Studierende**

€ 39,00 (D)
€ 40,10 (A)

9 783898 641784