Mehr Testwirtschaftlichkeit durch Value-Driven-Testing

Harry M. Sneed · Stefan Jungmayr

Leistungsbemessung nach Wertbeitrag

Value-Driven Software-Engineering ist aus der Synthese von Softwareproduktmanagement und betriebswirtschaftlicher Wertanalyse entstanden. Barry Boehm hat bereits 1981 den Begriff earned value geprägt [8], mit dem der Entwicklungsfortschritt in einem Projekt gemessen werden kann. Ein Softwareentwicklungsprojekt besteht aus einzelnen Aufgaben. Jede Aufgabe produziert ein bestimmtes Ergebnis (ein Artefakt) als Teil des Gesamtprodukts. Dieses Ergebnis kann ein Dokument, ein Entwurfsdiagramm, ein Codemodul, eine Testprozedur oder ein sonstiges Softwareartefakt sein, das einen Wert relativ zu seinem Anteil am gesamten Softwareprodukt besitzt. Wer ein Ergebnis liefert, erwirbt dessen Wert in sogenannten Wertpunkten. Der Stand eines Projektes bzw. der Fertigstellungsgrad eines Produkts wird dann anhand der Anzahl erworbener Wertpunkte ermittelt, daher der englische Begriff earned value.

Den gleichen Ansatz benutzte Boehm später auch bei Anforderungen [12]: Statt alle Anforderungen gleich zu behandeln, gewichtete er die einzelnen Anforderungen mit value-points. Die Anforderungen mit den meisten Wertpunkten werden dabei vorgezogen, die mit den wenigsten Punkten zurückgestellt. Auf diese Weise werden die Anforderungen nach ihrer fachlichen Bedeutung priorisiert (die agile Softwareentwicklung verfolgt einen ähnlichen Ansatz, wenn Anforderungen in der Reihe ihres Wertes für den Kunden abgearbeitet werden [48]).

Der Earned-value-Ansatz erlaubt dem Projektcontrolling, den Fortschritt eines Projektes zu verfolgen. Nicht unterstützt wird aber die Entscheidung, ob ein Projekt oder Teilprojekt *überhaupt* angegangen werden soll und – falls ja – *welche* Anforderungen zu erfüllen sind und *welche nicht*. Diesen Fragen widmeten sich Boehm und Huang erst in jüngerer Vergangenheit unter dem Begriff Value-Driven Software-Engineering [10]: Die Softwareentwicklung soll wirtschaftlich gerechtfertigt werden, indem die Kosten mit dem Nutzen eines Produkts abgeglichen werden.

Value-Driven Software-Engineering

Wertgetriebenes Software-Engineering basiert u. a. auf den folgenden Softwaremanagementansätzen, die den ökonomischen Wert einer Aktivität, d. h. ihre Rentabilität, berücksichtigen [4]:

- *Requirements Management*: Gewichtung und Priorisierung von Anforderungen.
- *Reconcilitory Design*: Anpassung der fachlichen Systemziele an die erreichbaren technischen Möglichkeiten bzw. an die vorhandene Architektur.
- *Feasibility Analysis*: Bewertung der Implementierungsumgebung im Hinblick auf ihre Eignung für das Projekt.
- *Risk Management*: Einbeziehung der Projektrisiken in die Projektkalkulation.
- *Risk-Driven Testing*: Funktionen mit dem größten Risiko werden intensiver getestet.

DOI 10.1007/s00287-010-0498-3

Harry M. Sneed
ANECON GmbH, Wien, Österreich
E-Mail: Harry.Sneed@anecon.com

Stefan Jungmayr
71254 Ditzingen, Deutschland
E-Mail: artikel@testbarkeit.de

Zusammenfassung
Die Wirtschaftlichkeit von Tests wird von Projektverantwortlichen häufig infrage gestellt. Zu Recht, denn bisher fehlen messbare Argumente, warum und wie viel getestet werden sollte. Value-Driven Software-Engineering ist ein Ansatz von Barry Boehm, der die Entwicklung von Software an ihrem ökonomischen Wert ausrichtet. In diesem Beitrag weiten wir diesen Ansatz auf das Gebiet des Softwaretestens aus und stellen erprobte Metriken für Kosten und Nutzen von Tests vor, sodass deren Return on Investment (ROI) berechnet werden kann. Damit werden Testverantwortliche in die Lage versetzt, die hohen Kosten des Testens zu rechtfertigen.

- *Project Controlling*: Überwachung des Projektfortschritts anhand des Wertes der Zwischenergebnisse.
- *Quality Management*: Auswahl der Gewichtung der anzustrebenden Qualitätsziele.
- *Projektmanagement*: Aufstellung der Projekte im Hinblick auf die Ziele der Auftraggeber.
- *Software Reuse*: Anwendung vorgefertigter Softwarekomponente wie Frameworks, Fertigbauteile und Entwurfsmuster.

Jede Aufgabe in einem Projekt ist nur so viel wert wie ihr Ergebnis, und dieser Wert muss größer als die Kosten sein, d. h. der Return on Investment soll positiv sein. Boehm berechnet den Return on Investment wie folgt:

$$\text{ROI} = (\text{Nutzen} - \text{Kosten})/\text{Kosten}.$$

Der ROI eines einzelnen Produkts lässt sich mit dem eines anderen Produkts vergleichen und somit nach betriebswirtschaftlichen Gesichtspunkten einstufen. Am Ende läuft es darauf hinaus, dass jede Projektaufgabe, egal ob Anforderungsspezifikation, Entwurf, Programmierung oder Test, ökonomisch rechtfertigbar sein muss. Teilprodukte mit dem höchsten ROI werden vorgezogen. Die Softwareentwicklung soll von rein wirtschaftlichen Überlegungen getrieben werden [5].

Für iterative Entwicklung bedeutet dies, dass jene Anforderungen mit der höchsten Wertschöpfung in der ersten Iteration realisiert werden. Nach jeder Iteration werden die verbleibenden Anforderungen neu bewertet und entsprechend eingestuft. So werden neue Anforderungen mit einer hohen Wertschöpfung den alten Anforderungen mit weniger Wertschöpfung vorgezogen. Der betriebswirtschaftliche Wert der Anforderungen bestimmt die Reihenfolge ihrer Implementierung [11].

Der ROI einer Komponente steigt mit ihrer *Wiederverwendung*. Eine empirische Studie [6] bezifferte die Einsparung der Entwicklungskosten durch die Nutzung eines gemeinsamen Produkt-Frameworks sowie einer gemeinsamen Klassenbibliothek im Rahmen einer Softwareproduktlinie mit 55 %.

Neben der Wiederverwendung spielt auch die *Zeit* bei der Ermittlung des Werts eines Softwareprodukts eine Rolle [47]. Der Wert eines Softwareprodukts ist nicht statisch, sondern er kann sich über die Zeit stark verändern. Der aktuelle Wert eines Produkts entspricht dem Betrag, den ein Kunde zu zahlen bereit wäre, wenn das Produkt zum aktuellen Zeitpunkt vollständig verfügbar *wäre* (auch wenn die Entwicklung aktuell noch nicht abgeschlossen ist). Mit jedem Monat, den der Kunde wartet, sinkt der Nutzen des Produkts für ihn. Es kommt daher darauf an, den Wertverlust eines Produkts über die Zeit möglichst klein zu halten, auch wenn das mit Abstrichen in der Qualität verbunden ist. Eine Verkürzung der Zeit bis zur Auslieferung kann über die Parallelisierung der Arbeit und die Erhöhung der Anzahl der involvierten Mitarbeiter erreicht werden [21].

Der Nutzen zum Zeitpunkt der Auslieferung muss vorkalkuliert werden und ergibt sich aus dem gegenwärtigen Nutzen minus des monatlichen Wertverlusts für jeden Monat, den der Kunde auf das Produkt warten muss. Je länger ein Projekt nicht fertig wird, desto geringer wird der ROI (dies ist ein gutes Argument für agile Entwicklung).

Wertverlust durch Softwarefehler

Testen erhöht nicht nur die Kosten, es verlängert auch die Projektdauer und senkt damit den ROI. Daher muss entschieden werden, wie viel in den Test eines Softwareprodukts zu investieren ist. Gleichzeitig senkt der Test die Wartungskosten, da die Behebung von Fehlern zwischen 25 und 40 % der gesamten Wartungskosten ausmacht [39, 45], die wiederum für mindestens 67 % der Lebenszykluskosten verantwortlich sind [9]. Demnach macht die Fehlerkorrektur 17–26 % der gesamten

Abstract
The following contribution is an extension of the latest research on value-driven software engineering to the field of software testing. The goals and methods of value-driven software engineering are reviewed and analyzed in regard to their application to testing. The particular difficulties of cost justifying testing are pointed out. The point is made that despite these difficulties it is possible to quantify the benefits of testing in terms of the costs of errors occurring in production and the costs of correcting those errors. The paper proposes using the methods of risk analysis to predict those costs. On the other hand, a cost estimation method for calculating the costs of testing based on a modified COCOMO-II equation is proposed. In the end the costs and benefits are compared to compute a return of investment (ROI) on testing. This is illustrated in constructed examples. The goal of the work is to put test managers in a position to justify the increasingly high costs of system testing.

Lebenszykluskosten aus (Abb. 1). Bei Produktkosten von beispielsweise 5 Mio. Euro entspricht dies 0,85–1,35 Mio. Euro Kosten für die Fehlerkorrektur! Fehlerbehebung ist also ein signifikanter Kostenfaktor! Alles was dazu dient, die Fehlerbehebungskosten zu senken, ist von nachweisbarem Nutzen [68].

Neben den Kosten für die Fehlerbehebung ist auch die Wirkung von unentdeckten Fehlern auf die Auftragslage zu betrachten: Bei der Untersuchung eines Finanzdienstleistungsunternehmens in Kalifornien wurde beispielsweise festgestellt, dass 10 % der möglichen Geschäfte (über einen Zeitraum von drei Jahren hinweg) wegen Fehlern verloren gegangen sind [13].

Zusätzlich sind die durch Fehler verursachten Folgekosten *beim Kunden* zu betrachten. So führten beispielsweise Softwarefehler beim Arbeitslosenunterstützungssystem ALU-II in Deutschland dazu, dass 25 Mio. Euro zu viel an die Kassen überwiesen wurden. Laut einer parlamentarischen Untersuchung des Landes Schleswig-Holstein gingen täglich zwei Arbeitsstunden pro Sachbearbeiter wegen Fehlern in errechneten Beträgen verloren. Hinzu kommt, dass ein Jahr nach Auslieferung immer noch 146 Fehlerumgehungen erforderlich waren. Die parlamentarische Untersuchung kam zu dem Schluss, dass es billiger wäre, die Gelder manuell auszuzahlen. Die manuelle Auszahlung der Arbeitslosengelder hätte in einem Bundesland 300 Mio. Euro gekostet – die tatsächlichen Kosten mit dem ALU-II-System beliefen sich auf 380 Mio. Euro. So verursachte das fehlerhafte System einen jährlichen Verlust von 80 Mio. Euro [54].

Diese Beispiele zeigen, dass Softwarefehler erhebliche Folgekosten haben können und daher zu einem Wertverlust führen. Anderseits stellt jeder zusätzlich benötigte Entwicklungsmonat ebenfalls einen Wertverlust dar. Der Projektlei-

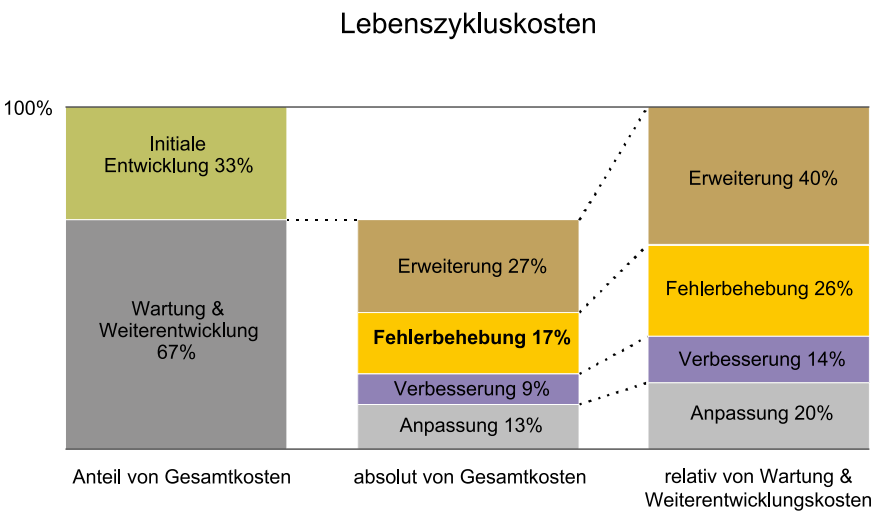

Abb. 1 Anteil der Fehlerkorrektur an den Lebenszykluskosten

ter muss daher bewerten, welcher Wertverlust größer ist: der durch die zusätzlich benötigte Entwicklungszeit oder der durch die Fehlerhaftigkeit bedingte.

Ansätze zur Fehlerreduzierung

Einerseits sollen die Folgekosten von Fehlern möglichst gering gehalten werden. Andererseits ist das Budget für die Fehlervermeidung in der Praxis begrenzt. In den vergangenen 40 Jahren Software-Engineering sind etliche Ansätze entstanden, Fehler zu vermeiden bzw. zu finden: psychologische, konstruktive, analytische und empirische Ansätze [57].

Zu den *psychologischen Ansätzen* gehören alle Maßnahmen, die Projektmitarbeiter zu einer möglichst fehlerfreien Arbeit motivieren sollen, wie z. B. Gewinnbeteiligung, Quality Grades, Earned Values, Zero Defects und Geldstrafen. Bisher hat keiner dieser Ansätze wirklich viel genutzt. Untersuchungen haben gezeigt, dass von 100 Zeilen, die ein Mensch schreibt, im Schnitt 1,5 fehlerhaft sind [25]. Von diesen Fehlern wird er die Hälfte selber erkennen und ausbessern, 0,7 Fehler in den 100 Zeilen bleiben aber unentdeckt. Schreibt jemand beispielsweise 1000 Codezeilen, so summiert sich dies immerhin auf 7 Fehler.

Konstruktive Qualitätsansätze versuchen, die Fehlerursachen zu reduzieren. Dazu gehört die Nutzung von Modellen (Model-Driven Development), die Nutzung automatisierter Werkzeuge, bessere Entwicklungsmethoden und Wiederverwendung. Insbesondere die Wiederverwendung bewährter Komponenten hat in der Tat dazu beigetragen, die Fehleranzahl zu reduzieren. Nur dort, wo der Mensch nichts Neues schafft, sondern bestehendes wiederverwendet, ist die Fehlerrate signifikant zurückgegangen. Der zuverlässigste Code ist demnach der Code, den man nicht neu schreiben muss [1]. Andere konstruktive Ansätze, wie das Model-Driven Development, haben die Fehlerursachen lediglich auf eine höhere semantische Ebene verschoben: Statt im Code entstehen die Fehler nun im Modell.

Im Rahmen der *analytischen Qualitätssicherung* werden Softwareartefakte wie Dokumente, Diagramme, Codebausteine usw. von einem Menschen oder einem Automaten geprüft. Zu den analytischen Ansätzen gehören u. a. Design Reviews, Codeinspektionen, Walkthroughs und statische Analysen [24]. In empirischen Untersuchungen wurde belegt, dass *Codeinspektionen* mehr Fehler aufdecken können als Tests [27]. *Analysewerkzeuge* werden bezüglich ihrer Fehlererkennungsfähigkeit immer mächtiger. Dennoch gibt es bestimmte Klassen von Fehlern, die durch einen analytischen Ansatz nicht aufgedeckt werden können – vor allem solche, die *lokal* in einem Dokument oder einem Codebaustein nicht erkennbar sind. Fehler, die durch das Zusammenwirken der Bausteine oder die Interaktion mit der Umgebung entstehen, bleiben unerkannt. Da die heutige Software immer mehr Standardbausteine verwendet und immer mehr von ihrer Umgebung abhängig ist, steigt der Anteil solcher Fehler [51].

Der *Test* ist ein *empirischer Ansatz zur Softwarequalitätssicherung*. Dabei wird die Anwendungssoftware auf der Hardware und im Zusammenhang mit der Basissoftware, in die sie eingebettet ist, ausgeführt. Da diese Probe unausweichlich ist, glauben viele, auf die anderen Qualitätsansätze verzichten zu können: „Warum prüfen, wenn man sowieso testen muss? Die Fehler, die durch die Prüfung entdeckt werden, werden auch durch den Test auffallen." Dies ist eine falsche Schlussfolgerung. Man braucht sowohl das Prüfen als auch das Testen, da beide weitgehend unterschiedliche Kategorien von Fehlern aufdecken. Aber leider ist die genannte Denkweise sehr verbreitet und führt zu einem erhöhten Testaufwand bei lokalen Fehlern, da diese wesentlich günstiger durch analytische Prüfungen aufgedeckt werden könnten [18].

Ein weiterer Weg zur Steigerung der Softwarequalität ist die *Wiederverwendung bestehender Softwarekomponenten*, z. B. Frameworks, Klassenbibliotheken und neuerdings Webservices. Dieser Ansatz setzt jedoch voraus, dass diese Fertigsoftware einen hohen Qualitätsstand hat, sonst werden nur noch mehr Fehler in die eigene Software hineinimportiert. Als Folge muss der Anwender der Fertigsoftware nicht nur die Kosten der eigenen, sondern auch noch die Kosten der fremden Fehler tragen. Diese Kosten werden durch die Mehrfachnutzung der Fertigsoftware vervielfältigt. Daher die Notwendigkeit einer Zertifizierung aller öffentlich angebotenen Softwarebausteine. Natürlich wird dieser Zertifizierungsprozess Kosten verursachen, aber diese Kosten sind gering im Verhältnis zu den Kosten, die entstehen, wenn fehlerhafte Fertigsoftware mehrfach benutzt wird. Es muss verhindert

werden, dass fertige aber fehlerhafte Bausteine neue Anwendungssysteme verseuchen [63].

Entwicklung der Testkosten

Der Anteil der Testkosten an den Gesamtkosten nimmt beständig zu: Ende der 1980er-Jahre rechnete ein Projektleiter mit maximal 25 % Aufwand für den Test (was schon für die damalige Komplexität zu gering war, aber mehr als das war man nicht bereit zu investieren). Heute ist die Komplexität der Software um ein Vielfaches gestiegen und die Testkosten liegen bei mehr als 50 % der Gesamtprojektkosten [67].

Komplexitätstreiber ist der Einsatz immer verschiedenartiger Komponenten mit immer mehr Beziehungen untereinander und zur Umwelt. Die Anzahl der potenziellen Kombinationen von Funktionen und Daten nimmt exponentiell zu. Insbesondere die Objekttechnologie hat zu einer Steigerung der Komplexität geführt, da die Anzahl der Beziehungen relativ zur Anzahl der Bauelemente überproportional gestiegen ist (siehe Exkurs).

Die Webtechnologie mit ihren zahlreichen Verbindungs- und Verwendungsmöglichkeiten führt zu einer weiteren Komplexitätssteigerung. Das liegt vor allem an den vielen potenziellen Links. Die Anzahl möglicher Pfade durch das System, die Anzahl möglicher Zustände der Daten und die Anzahl der Abhängigkeiten nehmen mit jedem Technologieschub zu. Gleichzeitig wächst die absolute Codemenge, obwohl der von den Anwendungsentwicklern beigetragene Codeanteil stetig abnimmt. Dies erklärt, warum der Testaufwand relativ zum Entwicklungsaufwand immer höher wird. Man testet nicht nur die selbst erstellte Funktionalität, sondern auch die übernommene Funktionalität plus die Beziehungen zwischen den beiden [6].

Wenn eine Tätigkeit 50 % der Gesamtprojektkosten und mehr für sich beansprucht, drängen sich natürlich Fragen zu ihrer Wirtschaftlichkeit auf. Welchen Gegenwert hat man für die hohen Kosten? Warum muss man so viel für den Test ausgeben? Wie viel darf der Test kosten? Diese und andere Fragen lassen sich nur dann beantworten, wenn man eine solide Kalkulationsbasis für den Testaufwand hat [3].

Exkurs zur Objektorientierung und Komplexität:

Das eigentliche Ziel der Objekttechnologie war die Reduzierung der Codemenge. Jede Funktion (d. h. Methode) und jedes Datenattribut soll nur einmal vorkommen. Wer sie verwenden will, muss sie entweder erben oder per Assoziation auf sie zugreifen. Da gleichzeitig angestrebt wird, den Code auf möglichst viele kleine Module (d. h. Klassen) zu verteilen, entsteht dadurch eine Vielzahl gegenseitiger Beziehungen bzw. Abhängigkeiten. Es ist zwar leichter geworden, einzelne Klassen zu testen, weil innerhalb der Klassen weniger Pfade existieren, stattdessen ist aber der Integrationstestaufwand mit der Zahl der potenziellen Interaktionen um ein Vielfaches gestiegen [65].

Wirtschaftlichkeit des Softwaretests

Nach Spillner [69] verfolgt der Softwaretest zwei Ziele: Er soll a) nachweisen, dass sich die Software konform zu den Anforderungen verhält und b) Fehler finden. Beide Ziele sind wichtig und sollten nicht vernachlässigt werden, aber in diesem Beitrag liegt der Schwerpunkt auf der Fehlerfindung. Wir vertreten die These, dass allein das Finden von Fehlern den Aufwand für den Test rechtfertigen kann und zwar im Bezug zu den eingesparten Folgekosten. Die Betonung liegt dabei allerdings auf dem Wort „kann“. Sollten die Fehlerkosten doch noch niedriger als die Testkosten ausfallen, ist der Testaufwand nicht gerechtfertigt.

Die Wirtschaftlichkeit des Tests wird durch die Berechnungen des ROI bewertet. Ein ROI-Wert kleiner gleich Null deutet auf einen ungerechtfertigten Testaufwand hin. Je höher der ROI-Wert über Null liegt, desto mehr lohnt es sich, zu testen. Der ROI-Wert für den Test wird wie folgt errechnet:

$$\text{Test\_ROI} = \frac{(\text{Testnutzen} - \text{Testkosten})}{\text{Testkosten}}.$$

Daraus ergeben sich zwei Fragen: Wie wird der Testnutzen ermittelt und wie sind die Testkosten im Voraus zu berechnen? Dieser Beitrag ist ein Versuch, diese beiden Fragen zu beantworten, denn nur so kann ein zunehmender Testaufwand gerechtfertigt werden.

Fehlerkosten

Softwarefehler lösen unterschiedliche Arten von Kosten aus, wie z. B. Kosten durch Produktions-

störungen, Kosten durch die Fehlerbehebung und Kosten durch den Vertrauensverlust der Kunden.

Produktionsstörungskosten

Produktionsstörungskosten sind Kosten, die durch einen Systemausfall oder durch falsche Ergebnisse verursacht werden: Fehler können zu einer Unterbrechung bzw. Störung der Anwendung führen. Anwender müssen dann den Fehler umgehen, ihn manuell ausbessern oder auf seine Behebung warten. Neben dem bereits erwähnten ALU-II-System gibt es viele weitere Beispiele für Produktivitätsverluste durch Softwarefehler: Für die Automobil- und Flugzeugbranche in den USA wurde allein für das Jahr 2000 ein Schaden von 1,8 Mrd. US-Dollar aufgrund fehlerhafter Software errechnet [50]. Nach einer Studie der Meta Group aus dem Jahre 2003 haben 74 % der europäischen IT-Anwender Unkosten von mehr als 500.000 Euro pro Jahr als Folge von Softwarefehlern in der Produktion [53]. Dies bezeugt, dass Produktionsverluste durch Softwarefehler zwar nicht exakt beziffert aber doch erheblich sind.

Die Höhe der Produktionsstörungskosten hängt von der Dauer der Unterbrechung bzw. der Dauer der manuellen Korrektur des falschen Ergebnisses ab. Wenn beispielsweise 20 Mitarbeiter mit einem System arbeiten und eine Unterbrechung der Arbeit minimal 1 h und maximal 2 h dauert, so ergibt dies 20–40 Sachbearbeiterstunden. Natürlich verursacht die Arbeitsverzögerung auch andere Unkosten, z. B. einen Auftragsstau, der mit Überstunden abgearbeitet werden muss oder sogar einen Auftragsverlust in unbestimmter Höhe. Da jedoch solche Unkosten schwer zu entziffern sind, bleiben sie zunächst unbeachtet. Im Weiteren werden nur die verlorenen Arbeitsstunden gezählt.

Zur Ermittlung der verlorenen Arbeitszeit beziehen wir uns auf die Methode der Risikoanalyse und folgen dem Minimum-Maximum-Prinzip, indem wir den geometrischen Mittelwert zwischen den maximalen und minimalen Verlustwerten als wahrscheinlichen Schaden annehmen [31].

Für oben genanntes Beispiel wird folgender Verlust geschätzt:

- maximaler Verlust: $20 \times 2\,\text{h} = 40$ Arbeitsstunden,
- minimaler Verlust: $20 \times 1\,\text{h} = 20$ Arbeitsstunden,
- wahrscheinlicher Verlust:

$$(40 + 20)/2 = 30 \text{ Arbeitsstunden.}$$

Wenn eine Arbeitsstunde 50 € kostet, folgt daraus ein wahrscheinlicher Verlust von 1500 € für diesen Fehler. Gäbe es, wie in dem ALU-II-System, 146 solcher Fehler, wäre das ein Gesamtverlust von 219.000 € pro Installation, bis alle Fehler behoben bzw. umgangen sind.

Die Störungskosten ergeben sich also aus den wahrscheinlich verlorenen Arbeitsstunden, die sich aus der Dauer des Ausfalls mal der dadurch betroffenen Benutzeranzahl berechnen.

Fehlerbehebungskosten

Jemand, der für die Instandhaltung zuständig ist, muss den Fehler untersuchen, die Ursache identifizieren, sie korrigieren und die Korrektur testen. Anschließend muss die korrigierte Version wieder in die Produktion eingespielt werden. Dies kann sich als sehr teures Unterfangen erweisen, vor allem, wenn die Software mehrfach installiert ist.

Die Fehlerbehebungskosten ergeben sich aus den folgenden Anteilen:

- Arbeitsstunden, die gebraucht werden, um die Fehler manuell zu lokalisieren,
- Arbeitsstunden, die gebraucht werden, um die Fehler manuell zu beheben,
- Arbeitsstunden, die gebraucht werden, um die korrigierte Komponente erneut zu testen,
- Werkzeugnutzungskosten zur Lokalisierung der Fehler,
- Kosten der Rechenzeit zum Test der korrigierten Komponente und
- Kosten des neuen Rollouts.

Fehlerbehebungskosten in Abhängigkeit vom Zeitpunkt der Entdeckung

In einer viel zitierten Studie zu den Fehlerbehebungskosten von Boehm aus dem Jahre 1975 [7] wurden die Fehlerbehebungskosten von drei verschiedenen amerikanischen Softwareproduzenten in der Rüstungsindustrie verglichen. Dabei wurde gezeigt, dass es rund 20-mal so viel kostet, Fehler in der Produktion zu korrigieren, als wenn sie beim Modultest bereits identifiziert und behoben werden. Einen Fehler beim Modultest zu beheben, kostet im Durchschnitt 1 h und beim Systemtest ca. 4 h. Den gleichen Fehler in der Produktion zu beheben, kostet mindestens 20 h, also das 20-fache der Behebung im Modultest (Abb. 2).

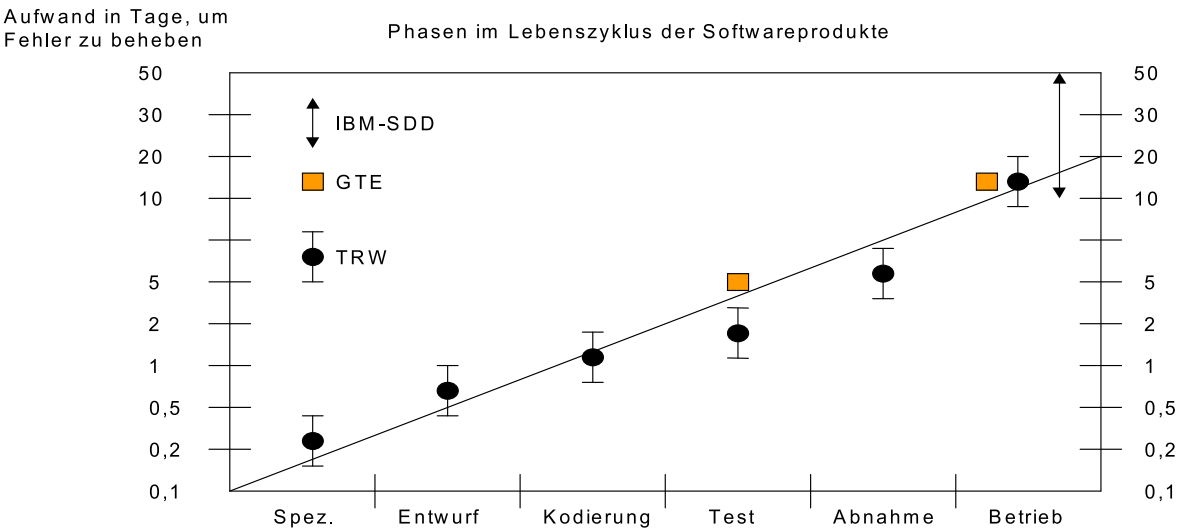

Abb. 2 Fehlerbehebungskosten nach Boehm im Jahre 1976

Tabelle 1

Schätzungen von durchschnittlichen Fehlerbehebungskosten

Studie	Jahr	Fehlerbehebungskosten pro Phase [h]				Referenz
		Inspektion	Test		Produktion	
			Modultest	Systemtest		
Fagan, IBM	1977	0,5	3		28	[23]
Remus, IBM Santa Theresa Labor	1983	–	1,5	15	82	[55]
Kan, IBM AS400 Labor	1995	–	–	13	92	[41]
Endres, IBM Labor Böblingen	2003	–	–	4	13	[19]
Teichmann, GEOS Projekt	2004	–	–	6	18	[66]

Ähnliche Untersuchungen kommen zu den in Tab. 1 dargestellten Schätzungen der durchschnittlichen Fehlerbehebungskosten.

Die Studie von Endres [19] ist für die Wirtschaftlichkeit des Testens von besonderer Bedeutung, da sie auf die wirtschaftliche Bewertung der Softwarequalität aus Nutzersicht sowie auf das große Einsparungspotenzial durch die rechtzeitige Erkennung von Softwarefehlern eingeht: Ein System mit 10 Mio. Programmzeilen wird mindestens 30.000 Fehler beinhalten, wenn es ausgeliefert wird. Wenn es gelingen würde, nur zwei Drittel dieser Fehler vor der Auslieferung abzufangen, entspräche dies einer Ersparnis von 180.000 Arbeitsstunden (bei Fehlerbehebungskosten im Systemtest von 4 h und in der Wartung von 13 h). Daher plädiert Endres für einen strengen und automatisierten Testprozess [20].

Weitere Untersuchungen zum durchschnittlich Aufwand für eine Fehlerbehebung kommen zu folgenden Ergebnissen:

- Jorgensen, norwegische Telekom, 1995 [34]: 18 h,
- Sneed, Wartungsprojekte, 1997 [59]: 25 h,
- Kajko-Mattsson, ABB, 2003 [38]: 32 h und
- Harrison und Walten, Wartungsprojekt, 2002 [29]: 28 h.

Lientz und Swanson berichten, dass mindestens 26 % der Softwareerhaltungskosten durch die Fehlerbehebung verursacht werden [45].

Aus diesen Wartungsstudien ergibt sich eine Bandbreite von 18–32 h pro Fehlerkorrektur nach der Freigabe, d. h. im besten Fall zwei Personentage und im schlimmsten Fall vier Personentage. Dies entspricht in etwa den Schlussfolgerungen der in Tab. 1 genannten Studien von Endres, Teichmann und Kajko-Mattsson [37].

Fehlerbehebungskosten in Abhängigkeit vom Zeitpunkt ihrer Entstehung

Softwarefehler entstehen in verschiedenen Phasen der Softwareentwicklung. Je früher ein Fehler zustande kommt, desto größer ist i. A. seine Auswirkung. Die Höhe des Fehlerbehebungsaufwands hängt weniger von der Schwere als vielmehr von der Quelle des Fehlers ab. Fehler in den Anforderungen verursachen einen viel größeren Aufwand bei der Fehlerbehebung als Fehler im Code, unabhängig von ihrer Auswirkung. Das liegt vor allem daran, dass man gleich *drei* Artefakte auszubessern hat: a) die Anforderungsspezifikation, b) das Entwurfsdokument und c) den Source-Code. Zusätzlich sind durch einen Fehler in der Anforderungsspezifikation in der Regel *mehrere* Entwurfsdokumente und *mehrere* Source-Module betroffen. Ebenso sind Fehler im Entwurf aufwendiger zu beheben als Fehler im Code, da man zwei Artefakte ausbessern sollte – das Entwurfsdokument und das betroffene Source-Modul.

Anforderungsfehler sind demnach die größten Kostenverursacher, gefolgt von Entwurfsfehlern und Codefehlern. Leider sind Anforderungsfehler die häufigste Fehlerart. Studien über GEOS [15] und andere Projekte haben gezeigt, dass mindestens 40 % der Fehler Anforderungsfehler sind und der Rest Codier-, Entwurfs- oder sonstige Fehler sind (Abb. 3) [28]. Daher schätzen wir den Fehlerbehebungsaufwand in Abhängigkeit von der Fehlerart wie folgt ab:

- Anforderungsfehler: 2,5 Personentage,
- Entwurfsfehler: 1,0 Personentag und
- Codierungsfehler: 0,5 Personentage.

Die Fehlerbehebungsaufwände bei grundlegenden Architekturfehlern (Konzeptmängeln) sind schwer in Personentagen auszudrücken, da sie relativ zum Aufwand der Gesamtentwicklung stehen. Eine Architekturfehlerbehebung kann leicht bis zur Hälfte des gesamten Entwicklungsaufwands verursachen. Deshalb werden solche hier nicht weiter behandelt.

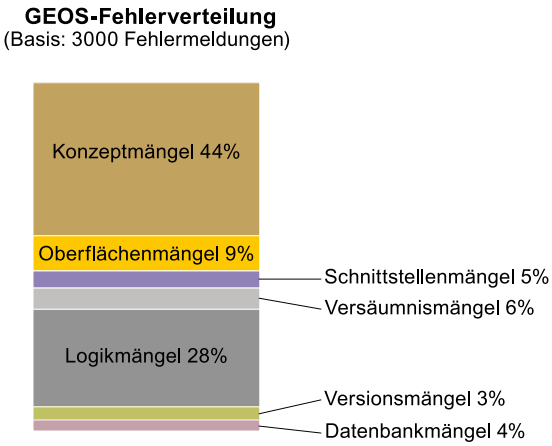

Abb. 3 Fehlerverteilung

Weitere Fehlerkosten

Zu den durch Fehler verursachten Kosten zählen auch jene, die durch den Vertrauensverlust der Benutzer entstehen. Dieser qualitative Aspekt ist zwar schwerwiegend, aber nicht ohne weiteres in Geld auszudrücken. Deshalb wird er hier nicht weiter berücksichtigt. Ein weiterer Kostenfaktor ist das Rollout der korrigierten Version. Wenn eine Software viele Anwender hat, ist dieser Posten nicht unerheblich. Bei eingebetteten Echtzeitsystemen kann er sogar in eine Rückrufaktion ausarten. Eines der Hauptargumente für Cloud Computing ist, dass der Softwarelieferant diese Unkosten größtenteils vermeiden kann. Er braucht nur die eine zentrale Version zu ersetzen und nicht sämtliche verteilte Versionen. Experten schätzen, dass die Verteilung von Hot-Fixes nochmals 15 % der Release-Kosten ausmacht [66].

Fehlerhäufigkeitsberechnung

Für die Betrachtung der Testwirtschaftlichkeit ist neben den durchschnittlichen Kosten eines einzelnen Fehlers die Fehlerhäufigkeit ein entscheidender Faktor. Die aus der Literatur bekannten Methoden zur Abschätzung der Fehlerhäufigkeit beziehen sich in der Regel auf ein Größenmaß des Zielsystems wie z. B. Function-Points [33], Object-Points [58], Use-Case-Points [40], Codezeilen oder Anweisungen.

Der Einfluss der Programmiersprache auf die Fehlerhäufigkeit wurde von Gaffney bereits 1984 [26]

als untergeordnet bewertet: „The number of bugs in a program does not appear to be a function of the language employed to write the code […].“. Die Fehleranzahl hängt viel mehr von der Codemenge ab. Nach seiner Untersuchung mehrerer Projekte mit verschiedenen Programmiersprachen stellte Gaffney fest, dass *vor* der ersten Inspektion oder dem ersten Modultest die Fehleranzahl zwischen 2,11 und 2,27 pro 100 Zeilen liegt. Das sind zwar 0,7 mehr als Fetzer behauptet [25], aber immerhin in dem Bereich [26]. Lipow hatte vorher mehrere Programme mit 340–12.541 Codezeilen untersucht [46] und kam zu dem Schluss:

$$\text{Fehleranzahl} = 4{,}04 + 0{,}0014 \times (\text{LOCs})^{4/3} .$$

Für ein Programm mit 10.000 Codezeilen sind das 30,5 Fehler pro 1000 Zeilen. Gaffney [26] kommt in seinem Experiment nur auf 22,7 Fehler pro 1000 Zeilen. Beide Autoren zählten sämtliche Mängel, auch die sogenannten Schönheitsfehler (die unterschiedlichen Fehlerraten liegen u. a. auch in uneinheitlichen Definitionen begründet, was als Fehler zu zählen ist – dies erschwert auch den Vergleich solcher Studien).

In einer Studie von C++ Code durch Vokac im Jahre 2004 wurden bei einem System mit 227 Klassen und 32.000 Codezeilen 54 Fehler innerhalb von sechs Monaten nach der Freigabe gemeldet. Das sind 1,69 Fehler pro 1000 Zeilen. Da Anweisungen in der Regel circa die Hälfte der Codezeilen sind, wären das 0,85 Fehler pro 1000 Anweisungen. Diese Fehlerrate reflektiert den Stand des Codes nach den Teststufen und ist repräsentativ für sicherheitskritische Software dieser Art in der Produktion. Bemerkenswert ist der große Unterschied zwischen der Fehlerrate von Gaffney vor dem Test und der Fehlerrate von Vokac nach dem Test [71].

Dies entspricht der Erfahrung der Firma ANECON beim Test des Webportals des Freistaats Sachsen. Dort wurden 33.000 Java-Anweisungen vom Entwicklerteam an das Testteam abgeliefert. In dem Systemtest, der vier Monate gedauert hat, wurden durch die 1,495 Testfälle 452 Fehler aufgedeckt. Das ergibt eine Fehlerrate von 0,013 bzw. 13 Fehler pro 1000 Java-Anweisungen. In den ersten sechs Monaten nach der Installation wurden 56 Fehler gemeldet. Daraus folgt eine Fehlerrate von 0,0017 oder 1,7 Fehlern pro 1000 Anweisungen. Der Systemtest hat die Fehlerrate um 88 % reduziert [62].

In neueren Studien, bezogen auf die Sprachen Java und C#, zeigt sich ein ähnliches Verhältnis zwischen Codegröße und Fehleranzahl, was aber in der Tat einer Fehlerverminderung durch die Nutzung dieser Sprachen entspricht, weil zur gleichen Aufgabe weniger Anweisungen benötigt werden [52]. Diese Sprachen fördern die Nutzung vorgefertigter und bereits getesteter Bausteine, die zwar noch Fehler aufweisen können, aber viel weniger als in den Anweisungen, die für die spezifische Anwendung neu geschrieben werden. Der Schlüssel zur Fehlerreduktion liegt also weniger in der Sprache selbst als in der Verwendung fehlerbereinigter Bausteine. Laut Britcher [14] müsste die Zahl der wiederverwendeten Anweisungen von der Gesamtzahl der Anweisungen abgezogen werden, um die Fehlerrate des neuen Codes festzustellen.

In einer Studie von Fehlerdaten der NASA [16] spielt die Anzahl der erforderlichen Testfälle eine Rolle bei der Entscheidung, wann mit dem Systemtest aufzuhören ist. Brown, Maghsoodloo und Deason [16] haben ein Kostenmodell zur Bestimmung der Anzahl Testfälle entwickelt. Sie greifen auf die Forschung von Laemmel [43] zurück, wonach sich die Wahrscheinlichkeit der Entdeckung eines Fehlers in der Produktion wie folgt berechnet:

$$\mathrm{P} = \mathrm{Pm} \times \mathrm{Pu} .$$

Pm ist die Wahrscheinlichkeit, dass der Tester einen Fehler nicht findet.

Pu ist die Wahrscheinlichkeit, dass der Benutzer auf einen Fehler stößt:

$$\mathrm{Pu} = E/N .$$

N ist die Anzahl der Möglichkeiten zur Nutzung eines Systems, wobei jede Möglichkeit mit einem bestimmten Eingabezustand verbunden ist.

E ist die Anzahl der Eingabezustände, die zu einem Fehler führen können.

Sofern der Tester alle Eingabezustände testet, die der Benutzer benutzt, kann die Wahrscheinlichkeit, dass der Tester einen Fehler verpasst, wie folgt berechnet werden:

$$\mathrm{Pm} = (1 - E/N)^t$$

t ist die Anzahl der getesteten Eingabezustände.

Demnach ist die Wahrscheinlichkeit eines Fehlers in der Produktion:

$$\mathrm{P} = (1 - E/N)^t \times (E/N) .$$

Die Bestimmung der Größe von N, d. h. der Anzahl der möglichen Nutzungen durch den Benutzer, ist schwierig. Wenn wir davon ausgehen, dass der Benutzer nur das nutzt, was er in den Anforderungen verlangt hat, dann entspricht N der Anzahl der einzelnen Nutzungsarten, die der Benutzer vorgibt, ausführen zu wollen. Dazu gehören auch jene Schritte, die für die Ausführung einer bestimmten Funktion Vorbedingungen erfüllen. Sollten die Anforderungen in Anwendungsfällen verkörpert sein, enthält der Anwendungsfall so viele Nutzungsmöglichkeiten, wie er Ausgänge hat. Jedenfalls sollte es möglich sein, zumindest die Anzahl der vorgesehenen Nutzungsmöglichkeiten N aus den Anforderungen bzw. Anwendungsfallbeschreibungen abzuleiten.

Die Bestimmung der Größe von E, d. h. der Wahrscheinlichkeit, dass der Benutzer mit einer bestimmten Nutzungsart auf einen Fehler stößt, erfordert die Kenntnis von Erfahrungswerten: Wir müssen von der Fehlerrate in bisherigen Systemen ausgehen – eine Alternative dazu gibt es nicht. Daher sind wir auf empirische Studien angewiesen (siehe Exkurse Nasa Studie und Webportal).

Im Abschnitt „Ansätze zur Fehlerreduzierung" haben wir bereits festgestellt, dass durchschnittlich mit einer Fehlerrate von 7 Fehlern pro 1000 Codezeilen zu rechnen ist. Dies ist der Qualitätsstand des Codes, wenn er aus dem Modultest kommt. Durch die Kombination von Integrations- und Systemtest wird von den Fehlern etwas mehr als die Hälfte gefunden, sodass die Fehlerrate der ausgelieferten Software circa 2 bis 3 Fehler pro 1000 Zeilen beträgt.

Durch rigide Qualitätssicherungsmaßnahmen wie Reviews, Inspektionen, Modultests, systematische Systemtests usw. ist es möglich, die Fehlerrate auf unter 1 pro 1000 Zeilen zu drücken. Die Softwarefehlerrate der Luft- und Raumfahrt in den USA liegt bei unter 0,5 Fehlern pro 1000 Zeilen zum Einsatzzeitpunkt (Tab. 2) [44]. Im Vergleich dazu hat die Software der Finanzwirtschaft der USA ca. 5 Fehler pro 1000 Zeilen. Dieser Faktor 10 in den unterschiedlichen Fehlerhäufigkeiten spiegelt den zusätzlichen Aufwand wider, den die Luft- und Raumfahrtindustrie in die Qualitätssicherung investiert. Eine mittlere Fehlerrate für die US-Industrie liegt bei 5 Fehlern pro 1000 Zeilen, wenn die Software zum ersten Mal ausgeliefert wird.

Die Fehlerhäufigkeiten sind sicherlich nicht nur von der Branche abhängig, sondern unterscheiden sich auch von Land zu Land. Deshalb können Statistiken wie die obige nur einen ersten Anhaltspunkt für eine Schätzung bieten. Ein Anwenderbetrieb kommt nicht umhin, selbst Daten zu seiner eigenen Fehlerhäufigkeit zu ermitteln und zu sammeln.

Die Fehlerhäufigkeit kann nicht nur auf die Anzahl der Programmzeilen bezogen werden, sondern auch auf die Personentage. Beispielsweise wurde im

Exkurs Nasa-Studie:

Ein Beispiel für eine solche Studie bezieht sich auf das NASA-Space-Shuttle-Projekt aus dem Jahre 1994 [42]. Im Space-Shuttle-Projekt wird zwischen drei Fehlerkategorien unterschieden. Zur ersten Kategorie gehören die früh entdeckten Fehler, die bei Inspektionen und Modultests aufgedeckt werden. Sie werden als Entwicklungsmängel bezeichnet. Zur zweiten Kategorie gehören die Fehler, die beim System- und Abnahmetest in Erscheinung treten. Sie gelten als Prozessfehler. Zur dritten Kategorie gehören die Fehler, die erst nach der Freigabe vom Anwender berichtet werden. Sie werden als Produktfehler bezeichnet. Die Trennung zwischen Prozess- und Produktfehler ergibt sich während des NASA Software-Readiness-Reviews, bei dem das System offiziell freigegeben wird.

In einem Evolutionsprojekt der NASA, in dem 24.000 Codezeilen geändert, hinzugefügt oder gelöscht wurden, wurden insgesamt 121 Fehler vor der Freigabe aufgedeckt – 92 (76 %) durch Inspektionen, 22 (18 %) durch den Modultest und 7 (6 %) durch den Systemtest. Dies bedeutet, dass 94 % aller vor der Freigabe entdeckten Fehler im Entwicklungsprozess beseitigt wurden. Die ursprüngliche Fehlerrate betrug vor der Inspektion 0,005 (bzw. 5 Fehler pro 1000 Zeilen), nach der Inspektion 0,0012 und nach dem Modultest nur noch 0,0003. Nach der Freigabe wurden noch 5 Produktfehler gemeldet. Das heißt es sind 95 % der erkannten Fehler vor der Freigabe ausgefiltert worden, davon 72 % durch Inspektionen.

Aufgrund dieser Erfahrungen entwickelte die NASA eine Metrik für die Hochrechnung der Restfehlerwahrscheinlichkeit bei der Übergabe der Software an die Verification & Validation-Gruppe. Diese Metrik diente als Stoppkriterium für den Systemtest.

Exkurs Test Webportal:
Im Rahmen der Entwicklung eines Webportals für den Freistaat Sachsen im Jahre 2005 wurde ein Systemtest von drei Java-Applikationen mit zusammen 33.000 Code-Anweisungen durchgeführt [2]. Das Anforderungsdokument zu diesem Websystem enthielt 108 Seiten Text und Bilder. Darin wurden 167 Anwendungsfälle mit 279 Schritten, 536 einzelnen Aktionen, 173 Regeln und 258 Zustandsabfragen beschrieben. Daraus ergaben sich 1134 spezifizierte Nutzungsmöglichkeiten. Dies entsprach auch der Zahl der abgeleiteten Testfälle. Im Laufe des viermonatigen Systemtests kamen weitere 361 Testfälle aufgrund von Verfeinerungen und Änderungen zu den ursprünglichen Spezifikationen hinzu, sodass letztlich 1495 Testfälle existierten. Durch 352 der getesteten Fälle wurden 450 Fehler aufgedeckt. Das sind 13 Fehler per 1000 Anweisungen, bzw. 6,5 Fehler per 1000 Codezeilen. Die Wahrscheinlichkeit, dass ein Fehler durch eine bestimmte Nutzung entdeckt wird, lag also bei 0,24, was sehr hoch erscheint. Wenn man aber unten sieht, dass die mittlere Fehlerrate für Webapplikationen in den USA bei 0,011 liegt, dann liegt die o. g. Fehlerrate von 0,013 durchaus im Rahmen – insbesondere angesichts des aus Zeitgründen sehr eingeschränkten Entwicklertests und der mutmaßlich reduzierten Wirksamkeit von Modultests bei Webanwendungen mit zahlreichen Abhängigkeiten nach außen. Der Aufwand für den Systemtest betrug 392 Testertage. Dies ergibt eine Produktivität von 3,8 Testfällen pro Testertag [61].

Projekt GEOS in Wien über eine Periode von fünf Jahren eine konstante Fehlerrate von 0,18 Fehlern pro Personentag registriert (und das unabhängig von der Gesamtanzahl der Entwickler) [60]. Die Fehlerrate blieb während der gesamten Zeit der Neuentwicklung von Funktionalität konstant. Erst in der Konsolidierungsphase, in der das Wachstum der Systeme aufhörte und nur noch wenig geändert wurde, sank diese personenbezogene Fehlerrate. Möglicherweise ist diese *Fehleranzahl pro Personentag* ein besserer Indikator für die Fehlerhäufigkeit als die übliche Anzahl der Fehler pro Programmzeile (bzw. pro Größeneinheit).

Codezeilen sind nicht gleich Anweisungen
Es dürfte dem Leser aufgefallen sein, dass bei manchen Untersuchungen von Fehlern per Codezeilen und bei anderen von Fehlern per Code-Anweisungen die Rede ist. Diese beiden Maße sind nicht identisch. Eigentlich ist der Begriff Codezeile ein veraltetes Maß aus der Urzeit der Informatik als wirklich eine Assembler oder FORTRAN Anweisung einer physikalischen Zeile oder Lochkarte entsprach. Die Lochkarte begrenzte die Länge einer Zeile auf 80 Zeichen. Bei den heutigen Sprachen ist die Zeilenlänge willkürlich. Es gibt Zeilen bis zu 600 Zeichen in denen zwei oder mehr Anweisungen stehen und es gibt Anweisungen die über mehrere Zeilen gehen. Die Codezeile ist ein absolut unzuverlässiges Maß. Das wichtige Maß wäre die Anweisung als syntaktische Einheit die z. B. in Java und C mit einer Semikolon oder einer schweifenden Klammer beendet wird sowie ein deutscher Satz mit einem Punkt endet. Die Informatik sollte dazu übergehen Anweisungen zu zählen aber wir haben noch viele alte Studien die sich auf Codezeilen beziehen. Bei alten Sprachen wie COBOL und PL/I ist das Verhältnis von Anweisungen zu Zeilen circa 1:1,3. Bei modernen Sprachen wie C# und Java ist das Verhältnis circa 1:2. Diese Verhältnisse gehen aus den 36 Vermessungsprojekten, die der Autor Sneed in den letzten 15 Jahren durchgeführt hat.

Die Fehlerrate ist im Betrieb von der Softwarenutzung abhängig: Je stärker die Software genutzt bzw. getestet wird, desto mehr Fehler werden gefunden. Irgendwann sind die Fehler in den meistgenutzten Funktionen gefunden und die Fehlerrate reduziert sich. Entsprechend den Wartungsstudien müsste die Fehlerrate nach der ersten Freigabe allmählich zurückgehen, bis sie irgendwann bei unter 0,5 Fehlern pro 1000 Anweisungen liegt (vergleichbar der Fehlerrate in der amerikanischen Luft- und Raumfahrtsindustrie [30]). Bis es aber so weit ist, werden für die Fehlerbehebung ca. 10–20 % der ursprünglichen Entwicklungskosten ausgegeben [49].

Im Allgemeinen kann ein Systemtester mit 5 bis 10 Fehlern pro 1000 Anweisungen nach dem Entwicklertest rechnen. Ein brauchbarer Schätzwert

Tabelle 2

Fehlerraten pro Kilo Lines of Code nach Branche in den USA

Anwendungsdomäne	Anzahl Projekte	Fehlerrate [Fehler/KLOC]	Mittlere Fehlerrate [Fehler/KLOC]
Automatisierungstechnik	55	2,0–8,0	5,0
Finanzwirtschaft	30	3,0–10,0	6,0
Eingebettete Systeme	45	0,5–5,0	1,0
IT-Systeme (Datenbanken)	35	2,0–14,0	8,0
Entwicklungsumgebungen	75	5,0–12,0	8,0
Militär – gesamt	125	0,2–3,0	< 1,0
Militär – Luftfahrt	40	0,2–1,3	0,5
Militär – Boden	52	0,5–4,0	0,8
Militär – Raketentechnik	15	0,3–1,5	0,5
Militär – Raumfahrt	18	0,2–0,8	0,4
Wissenschaft	35	0,9–5,0	2,0
Telekommunikation (Switching)	50	3,0–12,0	6,0
Test	35	3,0–15,0	7,0
Trainer/Simulatoren	25	2,0–11,0	6,0
Webapplikationen	65	4,0–18,0	11,0
Sonstige	19	2,0–15,0	7,0

ist demnach 7 Fehler pro 1000 Zeilen, sofern man nicht auf Daten aus der eigenen Fehlerdatenbank zurückgreifen kann.

Weiters sollten die Fehler gewichtet werden. Bei der Bewertung eines Fehlers als „kritisch", „schwer", „mittel" oder „leicht" hilft der ANSI/IEEE-Standard für die Fehlerklassifizierung [32].

Ein Entwicklungsbetrieb sollte in der Lage sein, die Fehlerstatistiken aus vergangenen Projekten auf neue Projekte zu projizieren. Wenn die bisherige Fehlerrate 5 Fehler pro 1000 Anweisungen betrug und das neue System in der gleichen Sprache und der gleichen Umgebung auf 200.000 Anweisungen geschätzt wird, kann der Projektplaner mit rund 1000 Fehlern rechnen, wobei bis zu 80 % der Fehler durch die Tester mit einem vertretbaren Aufwand gefunden werden sollten. So wurden beispielsweise im GEOS-Projekt in den früheren Entwicklungsphasen 72 % aller Fehler von den Testern gemeldet, später dann (als Folge der Testautomatisierung) mehr als 85 % bei nur einem Drittel des früheren Testaufwands [67].

Testkosten

Die Kosten für den Test setzen sich aus den Personalkosten sowie den Ressourcenkosten zusammen. Die Kosten lassen sich natürlich reduzieren, wenn gewisse Testphasen weggelassen werden. Man könnte z. B. auf den Integrationstest verzichten und gleich vom Modultest zum Systemtest übergehen. Man könnte sogar sowohl den Integrationstest als auch den Systemtest streichen und nur einen oberflächlichen Abnahmetest durchführen. Viele Anwender versprechen sich dadurch, Testkosten einzusparen. Erfahrungsgemäß deckt aber jede Testphase *andere* Fehlerarten auf. Wir haben beim Space-Shuttle-Projekt gesehen, dass 76 % der Fehler *vor* dem Systemtest gefunden wurden. Hinzu kommt, dass die Fehlerbehebung immer teurer wird, je weiter sie von der Fehlerquelle weg verschoben wird. Eine Fehlerbehebung im Systemtest kostet das Dreifache im Vergleich zu einer Fehlerbehebung im Modultest – vorausgesetzt, dass man den Fehler dort finden kann. Viele Fehler, mindestens 40 %, können im Modultest nur schwer aufgedeckt werden. Es ist daher unwirtschaftlich, Testphasen auszulassen. Man würde, wie eine Studie der Andersen Consulting belegt [17], allenfalls Zeit gewinnen. Die Kosten der Fehler würden bleiben (Abb. 4).

Testressourcenkosten

Bei der Testdurchführung kommen Hardwaregeräte und Softwareprodukte als Ressourcen zum Einsatz. Werden die Ressourcen für die Dauer des Tests ge-

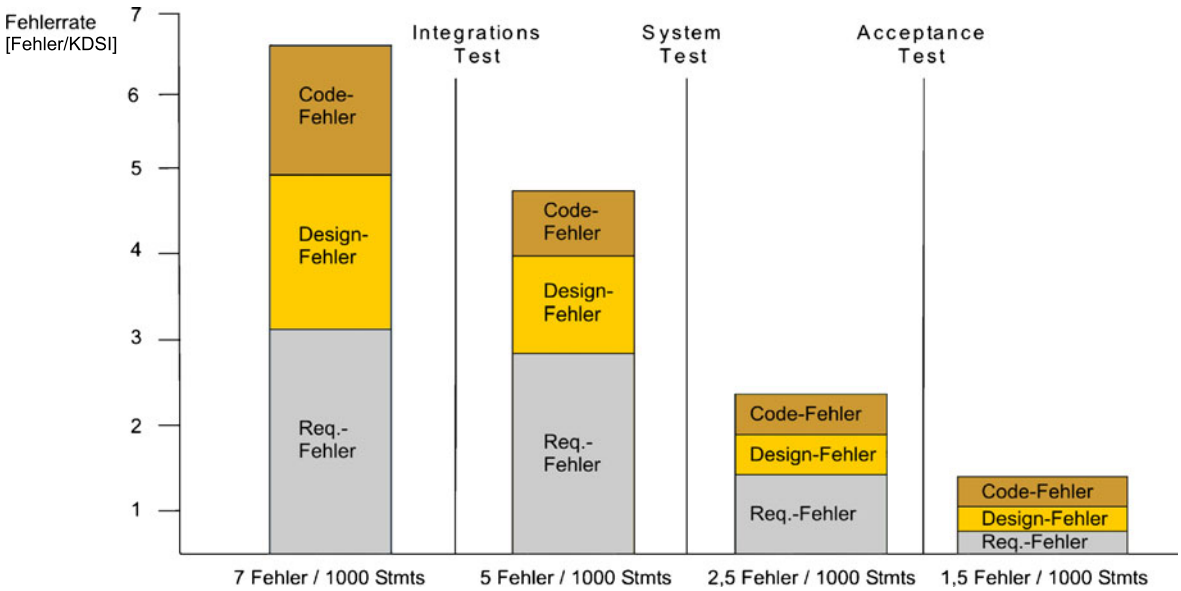

Abb. 4 Fehlerentfernungsrate nach Phase im V-Modell

mietet, so entsprechen die Ressourcenkosten den Mietkosten. Gehören die Ressourcen der Testorganisation, so muss der für den Test benutzte Anteil berechnet werden. Testressourcen können sehr teuer sein, wie z. B. Testwerkzeuge oder Simulationsumgebungen für eingebettete Echtzeitsysteme. Mit dem Grad der Testautomation steigen in der Regel die Kosten für die Testumgebung.

Die Testressourcenkosten können durchaus höher als die Personalkosten sein, insbesondere wenn z. B. in einem Niedriglohnland getestet wird. Wurde der Test in eine andere Firma (im Ausland) ausgelagert, so fallen ggf. noch erhebliche Transaktions- bzw. Kommunikationskosten für den Informationsaustausch zwischen den Testern und den Entwicklern an [22]. Insgesamt ergibt sich somit:

$$\text{Testressourcenkosten} = \text{Hardwarekosten} + \text{Softwarekosten} + \text{Kommunikationskosten}\,.$$

Testpersonalkosten

Der Testaufwand entspricht der Summe aller Personentage, die für den Test aufgewendet werden. Die Personalkosten ergeben sich aus dieser Summe, multipliziert mit dem entsprechenden Tagessatz:

$$\text{Personalkosten} = \text{Personentage} \times \text{Tagessatz}\,.$$

Wer mit fremdem Personal testet, bekommt die Personentage in Rechnung gestellt. Wer mit eigenem Personal testet, muss deren Kostenanteil je nach Prozentanteil ihrer Zeit, die sie am Test arbeiten, ausrechnen. Testautomatisierung verursacht in der Regel weniger Personentage aber einen höheren Tagessatz, weil die Tester eine höhere Qualifikation benötigen.

In fast jedem größeren Softwareprojekt mit mehr als drei Beteiligten ist der Test die teuerste Phase, da er viele Spezialisten beansprucht und lange dauert. Weil der Test komplexer Systeme im Prinzip nie abgeschlossen ist, bieten sich diverse Testüberdeckungsmaße, wie z. B. die Codeüberdeckung, die Datenüberdeckung und die Funktionsüberdeckung als Testabbruchkriterien an. Diese Kriterien sind allerdings sowohl schwer zu messen als auch schwer zu erreichen und bieten darüber hinaus keine Garantie für Fehlerfreiheit [70]. Die einzigen leicht messbaren Grenzen sind Zeit und Geld. Es ist daher notwendig, die Zeit und den Aufwand für den Test im Voraus einzugrenzen, um ein Ausufern der Kosten zu vermeiden. Damit sind wir bei der Frage des Sollaufwands angelangt. In der gängigen Praxis ist der Test dann zu Ende, wenn der Istaufwand den Sollaufwand erreicht oder überschreitet.

Anzahl Testfälle

Der geplante Aufwand (Sollaufwand) für den Softwaretest hängt von der Größe, der Komplexität und der Testbarkeit des Zielsystems sowie von der angestrebten Testüberdeckung und dem Grad der

Testautomatisierung ab. Je größer und komplexer ein System ist, desto mehr Testfälle werden benötigt. Für die Überdeckung der Anforderungen ist ein Testfall für jede Aktion, jede Regel und jeden Zustand erforderlich. Für die Überdeckung des Codes sollten jede Methode, jeder mögliche Objektzustand und jede Ausnahme getestet werden.

Die angestrebte Testüberdeckung richtet sich nach dem Qualitätsanspruch. Nehmen wir an, es stellt sich bei der Analyse der Anforderungsdokumentation heraus, dass eine Anwendung 1000 Testfälle benötigt, um alle funktionalen und nichtfunktionalen Anforderungen abzudecken. Bei einem hohen Qualitätsanspruch von 0,99 auf einer Skala von 0 bis 1, müsste man auch 990 dieser Fälle testen. Wenn aber die angestrebte Qualität nur bei 60 % des Erreichbaren läge, würden schon 600 Testfälle genügen. Es bliebe dann zu entscheiden, welche der 600 Testfälle den größten Nutzen bringen.

Testbarkeitsfaktor

Der Testaufwand hängt auch vom Grad der Testbarkeit eines Systems ab. Anwendungssysteme sind nicht immer gleich gut testbar. Es gibt Systeme, die leicht zu testen sind und andere, die nur schwer zu testen sind. Ein System, in dem die Anwendungslogik mit der Benutzeroberfläche verwoben ist, ist schwerer zu testen als ein System, in dem die Anwendungslogik von der Benutzeroberfläche getrennt ist. Die Komponenten des zweiten Systems können über eine Batch-Schnittstelle getestet werden, die Komponenten des ersten Systems können hingegen nur über die Benutzeroberfläche getestet werden. Systeme mit breiten Import- und Exportschnittstellen sind auch schwerer zu testen, weil die Tester mehr Parameter mit mehr Kombinationsmöglichkeiten generieren müssen. Die Größe der Datenbanken beeinflusst ebenfalls die Testbarkeit. Je mehr Attribute eine Datenbank hat, desto mehr Daten müssen generiert und validiert werden [35].

Testbarkeit lässt sich im Prinzip mittels einer statischen Analyse der Programm-Sourcen, der Oberflächendefinitionen (z. B. der HTML- oder XSL-Sourcen), der Schnittstellendefinitionen (z. B. der XML- oder IDL-Sourcen), der Datenbankschemen und der Systemarchitektur (z. B. der UML-Diagramme) bewerten. Aus der Analyse der Softwareartefakte ergibt sich ein rationaler Wert auf einer Skala von 0,2 bis 0,8. Der Skalenmittelwert 0,5 wird durch den Messwert der statischen Analyse (z. B. 0,45) dividiert, um den Testbarkeitsmultiplikator zu bekommen (z. B. 1,11). Das bedeutet, der Testaufwand wird wegen der unterdurchschnittlichen Testbarkeit der Software um 11 % höher sein. Durch Reengineering-Maßnahmen lässt sich die Testbarkeit in begrenztem Maße steigern [62].

Je schlechter die Testbarkeit ist, desto aufwendiger wird es, die Tests auszuführen. Der Testbarkeitsmultiplikator bewegt sich im Bereich von 2,5 bis 0,62, d. h. die Testbarkeit des Testobjektes kann den Testaufwand verdoppeln oder um 38 % reduzieren [60]. Empirische Untersuchungen zur Quantifizierung der Wirkung der Testbarkeit auf den Testaufwand sind den Autoren nicht bekannt. Nach Schätzungen von mehr als 45 Testexperten sind Testbarkeitsfaktoren von ca. 1,5–1,85 durchaus in der Praxis anzutreffen [36].

Testautomatisierungsfaktor

Der Grad der Testautomatisierung hat einen starken Einfluss auf den Testaufwand bzw. die Testkosten und variiert in der Praxis. Null ist er dann, wenn überhaupt keine Werkzeuge eingesetzt, alle Eingaben manuell über die Benutzeroberfläche eingegeben und auch die Ausgaben und internen Datenzustände manuell kontrolliert werden. Vollautomatisiert ist ein Test, wenn die Eingaben generiert und automatisch dem System zugeführt, die internen Abläufe und Zustände automatisch überwacht und alle Ausgaben automatisch validiert werden. Doch auch bei einem vollautomatisierten Test muss der Mensch die Vorgaben formulieren und die Auswertungen kontrollieren, d. h. ein gewisser manueller Arbeitsanteil wird immer bleiben, und zwar erfahrungsgemäß mindestens 25 %. Der Automatisierungsgrad liegt daher in der Praxis zwischen 0 und 75 %, d. h. durch die Automatisierung des Tests können höchstens 75 % des Aufwands eingespart werden.

Häufig sind aktuelle Testprojekte teilautomatisiert, wobei die Testwerkzeuge einige Routineaufgaben wie die Simulation der Benutzeroberfläche, den Abgleich der Soll- und Istausgaben sowie die Messung der Testüberdeckung übernehmen – der Automatisierungsgrad liegt dabei bei circa 50 %.

Testkostenkalkulation

Zur Schätzung der Testkosten kann das COCOMO-II-Modell von Boehm in einer von uns modifizierten Form eingesetzt werden [64]. Basis für die Schätzung

ist die Anzahl der Solltestfälle. Faktoren mit einem Einfluss auf den Testaufwand (wie Testbarkeit, Testautomatisierung, Reife des Testprozesses, Erfahrung der Testmannschaft und Güte der Testumgebung) werden über einen Skalierungsexponenten berücksichtigt. Produktivitätsbeeinflussende Faktoren wie Automatisierungsgrad und Testbarkeit fließen über Parameter in den Schätzwert mit ein. Deshalb wird hier eine modifizierte COCOMO-Formel für die Schätzung des Testaufwands empfohlen.

Die sieben wichtigsten Parameter für den Testaufwand sind:

- Anzahl der Solltestfälle (TF),
- angestrebte Testüberdeckung (TU),
- manuelle Testproduktivität (TP),
- Grad der Testautomatisierung (TA),
- Testbarkeitsmetrik (TB),
- mittlere Testbarkeit (MT),
- Testskalierungsexponent (TE) und
- Justierungsfaktor (AF).

Der Justierungsfaktor (AF) wird vom Anwender selbst bestimmt, um Differenzen zwischen Projekttypen abzufangen. Wenn z. B. eine Code-Anweisung in einem Echtzeitsystem 3,5-mal mehr kostet als die Code-Anweisung in einem integrierten IT-System (und der Anwender beide Systemtypen entwickelt), dann wäre der Justierungsfaktor für die Echtzeitprojekte 3,5 und für die IT-Systeme 1,0. Wenn der gleiche Anwender auch noch verteilte Systeme baut und diese um 50 % aufwendiger als die integrierten IT-Systeme sind, wäre der Faktor 1,5 [12]. Jedenfalls muss der Anwender sehr vorsichtig mit solchen Justierungsfaktoren umgehen, da sie den geschätzten Aufwand stark beeinflussen können.

Die modifizierte COCOM-II-Formel für die Kalkulation des Testaufwands ist:

$$\text{Aufwand} = \text{AF} \times \left\{ \left[\frac{\text{TF} \times \text{TU}}{\text{TP} + \text{TP} \times \left(1 - \text{TA}\right)} \right]^{\text{TE}} \times \left(\frac{\text{MT}}{\text{TB}} \right) \right\}.$$

Beispiel

Für ein Anwendungssystem mit 100.000 Anweisungen, 120 Anwendungsfällen und 600 Anforderungen sind erfahrungsgemäß circa 5000 Testfälle erforderlich, um die funktionalen und nichtfunktionalen Anforderungen voll abzudecken (siehe Exkurs Erfahrungswerte Testfallanzahl).

Aufgrund der bisherigen Erfahrungen im Projekt rechnen wir mit einer Fehlerrate von 6 Fehlern pro 1000 Anweisungen. Somit ergeben sich (bei 100.000 Anweisungen) insgesamt 600 Fehler. Bei einer angestrebten funktionalen Überdeckung von 67 % reicht es aus, 3370 der 5000 potenziellen Fälle zu testen, um 400 der 600 Fehler zu finden. Die geschätzte manuelle Testproduktivität liegt bei 4 Testfällen pro Testertag und der Grad der Testautomatisierung liegt bei 40 %.

Testkosten

Der Testbedingungsexponent liegt etwas über dem Durchschnitt von 1,03 und die Testbarkeit liegt bei 0,45 auf der Skala von 0,2–0,8. Der Systemtyp, der getestet wird, entspricht den integrierten Systemen, die bisher getestet wurden. Der Wert 1 bedeutet, es gäbe keine signifikante Abweichung von den Systemen, aus denen die Produktivitätsdaten stammen.

Exkurs Erfahrungswerte Testfallanzahl:
Man kann davon ausgehen, dass mindestens 1 Testfall pro 20 Anweisungen einer modernen objektorientierten Programmiersprache benötigt wird. Diese Erfahrung stammt aus einer Reihe von Testprojekten, in denen der Autor Harry Sneed mitgearbeitet hat: Das Webportalsystem des Freistaates Sachsen hatte 33.000 Java Anweisungen und benötigte 1500 Testfälle, um seine Funktionalität voll abzudecken. Das System GEOS hatte 2,5 Mio. C++ Anweisungen und benötigte 120.000 Testfälle, um voll abgedeckt zu werden. Das Gebühreneinzugssystem der österreichischen Wirtschaftskammer hatte 162.000 C# Anweisungen und benötigte 8500 Testfälle für die volle funktionale Abdeckung. Es gebe also ein erkennbares Verhältnis zwischen Systemgröße in Anweisungen und Anzahl der erforderlichen Testfälle. Funktionale Abdeckung in diesem Sinne heißt alle funktionale und nicht-funktionale Anforderungen sowie alle Nutzungsmöglichkeiten mindestens einmal zu erproben. Dies dürfte mit der Codeüberdeckung nicht verwechselt werden. Dafür brauchte man viel mehr Testfälle.

Der Testaufwand wird daher wie folgt abgeschätzt:

$$\text{Aufwand} = 1 \times \left\{ \left[\frac{5000 \times 0{,}67}{4 + (4 \times (1 - 0{,}4))} \right]^{1{,}03} \times \left(\frac{0{,}5}{0{,}45} \right) \right\},$$

$$\text{Aufwand} = 1 \times \left\{ \left[\frac{3350}{6{,}4} \right]^{1{,}03} \times 1{,}11 \right\},$$

$$\text{Aufwand} = 1 \times \left\{ 523^{1{,}03} \times 1{,}11 \right\},$$

$$\text{Aufwand} = 1 \times \{631 \times 1{,}11\} = 700 \text{ Personentage.}$$

Ein alternativer Ansatz zur Schätzung der Testkosten geht vom Verhältnis des Testaufwands zum Entwicklungsaufwand aus: Wenn die Systemkosten weggelassen werden und nur die Entwicklung einschließlich des Modul- und Integrationstests betrachtet wird, kann man mit einem Entwicklungsaufwand von mindestens 125 Personenmonaten bei einer hohen Produktivität von 800 Anweisungen pro Personenmonat rechnen. Unter der Vorgabe, dass die Kosten des System- und Abnahmetests nicht mehr als 25 % der Entwicklungskosten betragen sollen, dann ergibt sich ein Testaufwand von immerhin 31 Personenmonaten bzw. 682 Personentagen.

Test-ROI

Die 600 zu erwartenden Fehler werden sich erfahrungsgemäß wie folgt verteilen:

- 40 % (d. h. 240) Anforderungsfehler,
- 30 % (d. h. 180) Entwurfsfehler und
- 30 % (d. h. 180) Kodierfehler.

Das System soll durch 100 Sachbearbeiter bedient werden, die jeweils 40 € die Stunde kosten. Im Falle schwerer Systemausfälle können die 100 Sachbearbeiter jeweils eine Arbeitsstunde pro Ausfall nicht arbeiten, das sind 100 verlorene Stunden pro Ausfall. Im Falle leichter Systemausfälle verliert jeder Sachbearbeiter nur eine halbe Stunde, bzw. 50 verlorene Arbeitsstunden. Ein falsches Ergebnis führt im schlimmsten Falle bei *jedem* Sachbearbeiter zu einem Zeitverlust von einer halben Stunde bzw. 50 Stunden, im besten Fall bei *nur zehn* der Sachbearbeiter zu einem Zeitverlust von einer Stunde, bzw. 10 Stunden.

Wenn das System tatsächlich mit 600 Fehlern ausgeliefert wird, hätten wir bei einer Ausfallquote von nur 10 % mit mindestens 60 Ausfällen und bei einer Fehlerwirkung von 50 % mit 300 falschen Ergebnissen zu rechnen. Der maximale Schaden (unter obigen Annahmen) durch 60 Systemausfälle und 300 falsche Ergebnisse wäre:

$$\begin{aligned} 60 \times (100\,\text{h a } 40\,€) &= 240.000\,€ \\ + 300 \times (50\,\text{h a } 40\,€) &= 600.000\,€ \\ &\quad 840.000\,€. \end{aligned}$$

Der minimale Schaden (unter obigen Annahmen) durch Systemausfälle und falsche Ergebnisse wäre:

$$\begin{aligned} 60 \times (50\,\text{h a } 40\,€) &= 120.000\,€ \\ + 300 \times (10\,\text{h a } 40\,€) &= 120.000\,€ \\ &\quad 240.000\,€. \end{aligned}$$

Der wahrscheinliche Schaden durch Systemausfälle und falsche Ergebnisse wäre nach der Regel der Risikoanalyse [72]:

$$\text{wahrscheinlicher Schaden} = (\text{maximaler\_Schaden} + \text{minimaler\_Schaden})/2$$

$$(840.000\,€ + 240.000\,€)/2 = 540.000\,€.$$

Ausgehend von der Annahme, dass ein systematischer Systemtest mindestens zwei Drittel der vorhandenen Fehler aufdeckt, können wir mit einer Verminderung der Fehleranzahl von 600 auf 200 rechnen. Dies entspricht 400 Fehlern, die in der Wartung nicht behoben werden müssen, davon:

160 Anforderungsfehler	×2,5 =	400 Personentage
120 Entwurfsfehler	×1,0 =	120 Personentage
120 Kodierfehler	×0,5 =	60 Personentage
Summe		580 Personentage

Wenn ein Entwicklerpersonentag 800 € kostet, so betragen die Fehlerbehebungskosten 464.000 €. Zusammen mit den Fehlerfolgekosten von 540.000 € ergibt das ein Einsparungspotenzial von 1.004.000 €.

Der Testaufwand wurde auf 700 Personentage, geschätzt. Die Kosten der Tester nehmen wir im Beispiel mit 720 € pro Tag an. Die Testkosten berechnen sich somit wie folgt:

$$700 \times 720\,€ = 504.000\,€$$

bzw. 1260 € pro gefundenem Fehler.

Das ROI für dieses Testprojekt berechnet sich wie folgt:

$$\text{ROI} = (\text{Nutzen} - \text{Kosten})/\text{Kosten}$$

$$\text{ROI} = (1.004.000 - 504.000)/504.000 = 0{,}99.$$

Alternative Szenarien

Ohne Testautomatisierung würde die Testproduktivität bei 4 Testfällen pro Personentag liegen (statt bei 6,4 mit Testautomatisierung). In diesem Szenario ergibt sich ein Testaufwand von 1138 Personentagen bzw. Testkosten von 819.360 €.

Dies führt zu einem ROI von lediglich 0,224, weil die Kosten pro gemeldeten Fehler mit 2048 € relativ hoch sind. Die Projektleitung würde versuchen, die Anzahl der durchzuführenden Testfälle zu senken, um Kosten zu sparen. Eine solche Maßnahme verringert aber proportional die Anzahl der gefundenen Fehler. Bei einer reduzierten Testüberdeckung von 50 % würden nur 300 der 600 Fehler aufgedeckt werden. Damit verringert sich der Testnutzen. Andererseits könnten wir die Testproduktivität durch weitere Testautomation steigern, z. B. auf 8 Testfälle pro Testertag. Dann würden die Testkosten von 700 Personentagen auf 557 Tage zurückgehen. Das entspräche nur 401.040 bzw. 1003 € pro Fehler. Damit würde der Test-ROI auf 1,5 steigen. Dies unterstreicht die Bedeutung der Testautomation. Mit Testautomation kann das Test-ROI um das Dreifache erhöht werden. Alternative Schätzungen mit unterschiedlichen Parametern haben den zusätzlichen Wert, dass sie den Testplaner zwingen, überhaupt über Alternativen nachzudenken und seine Vermutungen mit Zahlen zu begründen.

Fazit

Das Beispiel aus dem vorhergehenden Kapitel zeigt die starke Abhängigkeit der Testwirtschaftlichkeit (ROI) von den folgenden Faktoren:

- Testproduktivität,
- Testüberdeckung,
- Testbedingungen,
- Testbarkeit des Produkts und
- Fehlerfindungsrate.

Der Testmanager muss diese Faktoren berücksichtigen, wenn er die Testkosten plant. Er steht in der Pflicht, dem Kunden genau vorzurechnen, was er für sein Geld bekommt, d. h. was er an Fehlerfolgekosten und Fehlerbehebungskosten einspart. Der Nutzen des Tests muss für jeden ersichtlich sein. Dies war auch das Ziel des Autors Harry Sneed, als er 1978 das erste Value-driven-Testprojekt für die Firma Siemens in Budapest durchführte. Der Projektvertrag sah vor, dass der Kunde für jeden gemeldeten Fehler einen Betrag in der Höhe der Fehlerklasse bezahlte. Hinzu kam ein Betrag für jeden getesteten Testfall, der die Testüberdeckung erhöhte. Dies diente dem zweiten Ziel des Testens, nämlich Vertrauen zu schaffen. Da die Tester auch nach Testfall und nach gefundenen Fehlern bezahlt wurden, waren sie motiviert, möglichst viele Fehler mit einer möglichst hohen Testüberdeckung in möglichst kurzer Zeit zu finden. Gleichzeitig wurde der Wert ihrer Leistung dadurch für jeden klar ersichtlich [56].

Die Transparenz der Testarbeit ist eine wichtige Vorbedingung für das Vertrauen des Kunden in den vom Softwarelieferanten durchgeführten Test. Denn nur im Vertrauen auf einen quantifizierten Nutzen wird er bereit sein, die hohen Kosten des Testens zu tragen. Das Ziel muss es sein, möglichst viele – vor allem kritische – Fehler zu finden und dies mit einem möglichst geringen Aufwand in einer möglichst kurzen Zeit. Dies setzt wiederum eine Steigerung der Testbarkeit und der Testproduktivität voraus. Beides verlangt eine Vorinvestition, die ausschließlich über den Nutzen des Tests zu rechtfertigen ist. Deshalb ist die Berechnung des Test-ROI unverzichtbar.

Literatur

1. Basili V, Briand L, Melo W (1996) How Reuse influences productivity of object-oriented systems. Commun ACM 39(10):104
2. Baumgartner M (2005) Bericht zum Testprojekt – Portal für den Freistaat Sachsen. ANECON Internal Report, Wien
3. Berg KP (2008) Testen von Web-Services im SOA Umfeld – Theorie und Praxis. OBJEKTspektrum 5:32
4. Biffl S et al (2006) Value-based Software Engineering. Springer, Berlin, pp 21
5. Blom S, Gruhn V, Koehler A, Schaefer C (2008) Methoden und Grundlagen der wertebasierten Softwareentwicklung. OBJEKTspektrum 1:12
6. Boeckle G, Clements P, McGregor J, Muthig D (2004) Calculating ROI for software product lines. IEEE Software 3:23
7. Boehm B (1975) The high costs of software. In: Horowitz E (Hrsg) Practical Strategies for Developing Large Software Systems. Addison-Wesley, Reading MA, p 3
8. Boehm B (1981) Software Engineering Economics. Prentice-Hall, Englewood Cliffs, p 597
9. Boehm B (1983) The Economics of Software Maintenance. In: Proceedings of 1st ICSM Conference, IEEE Computer Society Press, Monterey, p 9
10. Boehm B, Huang L (2003) Value-based software engineering. IEEE Computer 3:33
11. Boehm B, Turner R (2005) Management challenges to implementing agile processes. IEEE Software 5:30
12. Boehm B et al (2000) Software Cost Estimation with COCOMO-II. Prentice-Hall, Upper Saddle River, NJ
13. Boehm B, Huang L, Apurva J, Madachy R (2004) The ROI of software dependability – the IDAVE model. IEEE Software 3:54
14. Britcher R (1998) Software reliabilty – numbers aren't the whole story. IEEE Computer 10:128
15. Broessler P, Sneed H (2003) Critical Success Factors in Software Maintenance. In: Proceedings of ICSM 2003, IEEE Computer Society Press, Amsterdam, p 190
16. Brown D, Maghsoodloo S, Deason W (1989) A cost model for determining the optimal number of software test cases. Transact Softw Eng 15(2):218
17. Cole J, Gorham T, McDonald M (1997) Reinventing the testing process. Am Program 10(8):3
18. Dyer M (1992) The Cleanroom Approach to Quality Software Development. Wiley, New York, p 15
19. Endres A (2003) Softwarequalität aus Nutzersicht und ihre wirtschaftliche Bewertung. Informatik-Spektrum 18(1):20

20. Endres A, Rombach H-D (2004) Handbook on Software Engineering. Springer, Berlin, p 131
21. Erdogmus H, Favaro J, Strigel W (2004) Return on Investment. IEEE Software 3:18
22. Everett GD (2008) The value of software testing to business. Test Exp 2:6
23. Fagan M (1976) Design and code inspections to reduce errors in program development. IBM Syst J 15(3):182
24. Fagan M (1986) Advances in software inspections. IEEE Transact Softw Eng 12(7):744
25. Fetzer J (1988) Program verification – the very Idea. Commun ACM 31(9):1048
26. Gaffney J (1984) Estimating the number of faults in code. IEEE Transact Softw Eng 19(4):454
27. Gilb T, Graham D (1996) Software Inspections. Addison-Wesley, Reading
28. Graham D (2002) Requirements and testing – seven missing link myths. IEEE Softw 5:15
29. Harrison M, Walton G (2002) Identifying high maintenance legacy software. J Softw Maint Evol 14(6):429
30. Hasitschka M, Teichmann MT, Sneed H (2009) Software quality management. OBJEKTspektrum 1:74
31. Hillson D, Simon P (2007) Practical Project Risk Management – the Atom Methodology. Management Concepts Inc, New York
32. IEEE Standard 1044-1995 (1995) IEEE Guide to Classification for Software Anomalies. Institute of Electrical and Electronic Engineers, New York, NY
33. Jones TC (1991) Applied Software Measurements. Mcgraw-Hill, New York, p 43
34. Jörgensen M, Sjöberg D (2002) Impact of experience on maintenance skills. J Softw Maint 14(2):123–146
35. Jungmayr S (2004) Improving Testability of object-oriented Systems. Dissertation, Berlin, p 123
36. Jungmayr S (2009) Anforderungen an eine testbare Softwarearchitektur. Präsentation im Rahmen der Software Quality Days 2009, Wien, 20.–22. Januar 2009
37. Kajko-Mattsson M, Forsannander S, Andersson G (2000) Software problem reporting and resolution process at ABB. J Softw Maint 12(5):255–286
38. Kajko-Mattsson M (2003) Elicitating and Rating Problems within Support. In: Proceedings of ICSM-2003, IEEE Computer Society Press, Amsterdam, p 199
39. Kajko-Mattsson M (2006) Evaluation of CM^3: Front-End Problem Management within Industry. In: Proceedings of CSMR-2006, IEEE Computer Society Press, Bari, March 2006, p 367
40. Karner G (1993) Metrics for Objectory. Diplomarbeit, Nr. LiTH-IDA-Ex-9344:21, Universität Linköping, Schweden
41. Kan SH (1995) Metrics and Models in Software Quality Engineering. Addison-Wesley, Reading MA, p 178
42. Koster P, Peterson T (1994) Fault estimation and removal from the Space Shuttle software. Am Program 7(4):13
43. Laemmel A (1980) A statistical theory of computer program testing. Polytechnic Institute of NY, Report SRS 119/POLYEE 80-004
44. Laird L, Brennan C (2006) Software Measurement and Estimation – a Practical Approach. Wiley, New York, p 133
45. Lientz B, Swanson EB (1980) Software Maintenance Management. Addison-Wesley, Reading, p 105
46. Lipow M (1982) Number of faults per line of code. IEEE Transact Softw Eng 8(4):503
47. Little T (2004) Value creation and capture – a model of the software development process. IEEE Softw 3:48
48. McMenamin S, Palmer J (1984) Essential Systems Analysis. Yourdon Press, New York, p 7
49. Mookerjee R (2005) Maintaining enterprise software applications. Commun ACM 48(11):75
50. National Institute of Standards and Technology (2002) The economic impacts of inadequate infrastructure for software testing. Planning report 02-3, Triangle Park, NC
51. Nguyen HQ, Johnson B, Hackett M (2002) Testing Applications on the Web. Wiley, Indianapolis, p 22
52. Ostrand T, Weyuker E (2005) Predicting the location and number of faults in large software systems. IEEE Transact Softw Eng 31(4):340
53. Redaktion (2004) Bei Softwarequalität haperts – Meta Group Bericht über mangelnde Softwarequalität. Computerzeitung 28:4
54. Redaktion (2008) ALU-II Pannensoftware vor der Ablösung. Comp Wkly 11:8
55. Remus H (1983) Integrated Software Validation in the View of Inspections & Reviews. In: Proceedings of Symposium on Software Validation, Darmstadt, Germany, North-Holland, Amsterdam, p 57
56. Sneed HM (1979) Das Software-Testlabor. In: Heilmann H (Hrsg) 8. Jahrbuch der EDV, Forkel, Stuttgart, S 31
57. Sneed HM (1988) Software Qualitätssicherung. Rudolf Müller, Köln, S 58
58. Sneed HM (1996) Schätzung der Entwicklungskosten objektorientierter Software. Informatik-Spektrum 19(3):133
59. Sneed HM (1997) Measuring the Performance of Software Maintenance Departments. In: Proceedings of 1st European Conf. on Software Maintenance and Reengineering (CSMR), IEEE Computer Society Press, Berlin, p 119
60. Sneed HM (2003) Selective regression testing of large application systems. Softwaretechnik-Trends 23(4)
61. Sneed HM (2006) Testing an e-Government Website. In: Proceedings of IEEE Symposium on Web Site Evolution, IEEE Computer Society Press, Budapest, p 3
62. Sneed HM (2006) Reengineering for testability. Softwaretechnik-Trends 26(2):8
63. Sneed HM (2007) Qualitätsnachweis für Web Services. 2. Workshop Bewertungsaspekte serviceorientierter Architekturen, Shaker, Karlsruhe, S 1
64. Sneed HM, Jungmayr S (2006) Produkt- und Prozessmetriken für den Softwaretest. Informatik-Spektrum 29(1):23
65. Sneed HM, Winter M (2001) Testen objektorientierter Software. Hanser, München, S 21
66. Sneed HM, Hasitschka M, Teichmann MT (2004) Software-Produktmanagement. dpunkt, Heidelberg, S 358
67. Sneed HM, Baumgartner M, Seidl R (2008) Der Systemtest – Requirements-based Testing. Hanser, München, S 16
68. Solingen R (2004) Measuring the ROI of software process improvement. IEEE Software 3:32
69. Spillner A (2008) Systematisches Testen von Software. dpunkt, Heidelberg, S 3
70. Spillner A, Koch T (2003) Basiswissen Softwaretest. dpunkt, Heidelberg, S 146
71. Vokac M (2004) Defect frequency and design patterns – an empirical study of industrial code. IEEE Transact Softw Eng 30(12):904
72. Wallmüller E (2004) Risikomanagement für IT- und Softwareprojekte. Hanser, München, S 137