2. Auflage

Stephan Grünfelder

Software-Test für Embedded Systems

Ein Praxishandbuch für Entwickler, Tester und technische Projektleiter

→ Mit Beiträgen von Michael González Harbour, Reinhard Wilhelm, Johannes Bergsmann und Oliver Alt

Stephan Grünfelder war Programmierer und Tester für die unbemannte Raumfahrt und Medizintechnik, später Projektleiter für Steuergeräte-Entwicklung im Automobilbereich und arbeitet nun als selbständiger Trainer für Software-Testing und als Senior Software Tester für Broadcast-Ausrüstung. Er hat bzw. hatte Lehraufträge an der Hochschule Reykjavik, der Fachhochschule Technikum Wien und der Technischen Universität Wien.

Papier plus+ PDF.

Zu diesem Buch – sowie zu vielen weiteren dpunkt.büchern – können Sie auch das entsprechende E-Book im PDF-Format herunterladen. Werden Sie dazu einfach Mitglied bei dpunkt.plus+:

www.dpunkt.plus

Stephan Grünfelder

Software-Test für Embedded Systems

Ein Praxishandbuch für Entwickler, Tester und technische Projektleiter

2., aktualisierte Auflage

dpunkt.verlag

Stephan Grünfelder
stephan.gruenfelder@aon.at

Lektorat: Sandra Bollenbacher
Copy-Editing: Geesche Kieckbusch, Hamburg
Satz: Frank Heidt
Herstellung: Susanne Bröckelmann
Umschlaggestaltung: Helmut Kraus, www.exclam.de
Druck und Bindung: M.P. Media-Print Informationstechnologie GmbH, 33100 Paderborn

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

ISBN:
Print 978-3-86490-448-6
PDF 978-3-96088-148-3
ePub 978-3-96088-149-0
mobi 978-3-96088-150-6

2., aktualisierte Auflage 2017
Copyright © 2017 dpunkt.verlag GmbH
Wieblinger Weg 17
69123 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.
Es wird darauf hingewiesen, dass die im Buch verwendeten Soft- und Hardware-Bezeichnungen sowie Markennamen und Produktbezeichnungen der jeweiligen Firmen im Allgemeinen warenzeichen-, marken- oder patentrechtlichem Schutz unterliegen.
Alle Angaben und Programme in diesem Buch wurden mit größter Sorgfalt kontrolliert. Weder Autor noch Verlag können jedoch für Schäden haftbar gemacht werden, die in Zusammenhang mit der Verwendung dieses Buches stehen.
5 4 3 2 1 0

Vorwort

Wie viele Artikel und Bücher habe ich schon gelesen, in denen im ersten Satz steht, dass Computersysteme immer komplexer werden oder Entwicklungszyklen immer kürzer und dass man deshalb von nun an Dinge besser/schneller machen muss als früher? Ich kann es nicht sagen. Diese Ausrede und Einleitung ist aber schon seit mehreren Jahrzehnten gültig, denn Techniken werden unentwegt weiterentwickelt, Technologien wandeln sich durch neue Einflüsse und die Zeit steht niemals still. Schon gar nicht rund um die Themen der Software-Entwicklung. Dieses Buch wurde nicht geschrieben, weil gerade jetzt alles komplexer wird und ich, als Autor, der Welt sagen will, wie man damit umgeht, sondern weil es im deutschsprachigen Raum noch kein brauchbares Buch zum Thema Test von Software eingebetteter Systeme gab und ich oft genug gesehen habe, wie schwer sich manche Leute damit tun.

Praxisbezug des Buchs

Das Buch vermittelt praxisnahes Wissen zum Test von Software eingebetteter Systeme, so wie es jetzt und heute gemacht wird oder gemäß dem industriellen Stand der Technik gemacht werden sollte. Auch wenn der Stand der Wissenschaft und Forschung für das Schreiben des Buchs wichtig war, so nehmen forschungsnahe Themen nur wenig Platz ein, wenn deren industrielle Umsetzung Probleme bereitet. Das Buch präsentiert viele Beispiele, persönliche Erfahrungsberichte und enthält praxisnahe Fragen *und Lösungen* zum Selbsttest. Die Terminologie des Buchs orientiert sich weitgehend an der des ISTQB-Certified-Tester-Schemas und der Normenreihe ISO 29119. Das Buch ist dabei an vielen Stellen komplementär zum Curriculum des ISTQB, speziell dann, wenn es um Echtzeit, Concurrency und maschinennahe Themen geht, und es ist – unvermeidbar – zum Teil redundant dazu.

Software für eingebettete Systeme unterscheidet sich von anderer Software dadurch, dass die Software Teil des Produkts ist, das der Kunde kauft, und nicht das Produkt selbst. Meist wird das Produkt, in das ein Prozessor mit zugehöriger Software eingebettet ist, wertlos, wenn die Software nicht zuverlässig funktioniert. Im schlimmsten Fall muss es vom Markt genommen werden. Kein Wunder

also, dass man sich in dieser Sparte der Software-Entwicklung besonders viele Gedanken zur Korrektheit von Software macht.

Aber durch Testen alleine erhält man kein fehlerfreies Produkt. Wie ein alter Freund von mir zu sagen pflegte: »Man kann Software nicht am Ende der Produktentwicklung ›gesund testen‹, wenn es in so manchen Entwicklungsschritten zuvor krankt.« Das vorliegende Buch beschäftigt sich daher auch mit Praktiken links und rechts vom Test, die die Entwicklung begleiten und dazu beitragen, mit höherer Wahrscheinlichkeit »gesunde« Software zu erhalten.

Wie man dieses Buch liest

Das Spektrum eingebetteter Systeme reicht von batteriebetriebenen 16-Bit-Controllern ohne Betriebssystem und mit geringen Anforderungen an die Systemintegrität bis hin zu im Internet Of Things vernetzten Multiprozessor-Systemen mit Echtzeitbetriebssystem, Mensch-Maschine-Schnittstelle und Sicherheitsrelevanz. Entsprechend viele Zielgruppen hat dieses Buch. Daher gibt es vermutlich Kapitel im Buch, die für Ihre speziellen Aufgaben irrelevant sein könnten. Sie dürfen diese Kapitel gerne überspringen, denn ich habe mir größte Mühe gegeben, jedes Kapitel weitgehend unabhängig von den anderen zu schreiben, und das Letzte, was dieses Buch tun sollte, ist, Sie zu langweilen. So können Systemtester zum Beispiel getrost die Kapitel über Reviews und statische Analyse auslassen. Die Unabhängigkeit der Kapitel macht es nötig, dass Inhalte teilweise wiederholt werden müssen.

Fans moderner Programmiersprachen sollten sich nicht daran stoßen, dass fast alle Beispielprogramme im Buch in der Programmiersprache C geschrieben sind. C ist weder komfortabel noch modern, doch hat C noch immer seinen festen Platz in der Entwicklung von Produkten mit höchsten Qualitätsanforderungen, und die meisten Entwickler verstehen die Basissyntax dieser Sprache ohne Probleme.

Ich wünsche Ihnen viel Freude beim Lesen und viel Erfolg bei der Umsetzung der in diesem Buch präsentierten Ideen.

Herzlichst
Stephan Grünfelder, Oktober 2016

Danksagung

Ich persönlich fand es immer ärgerlich ein Buch zu lesen, in dem ein Autor über Themen schrieb, bei denen er – wie ich später herausfand – nicht wirklich sattelfest war oder die er nur aus einem einzigen Projekt kannte. Nun habe ich selbst ein Buch verfasst und noch dazu eines mit dem Anspruch, praxisnah zu sein. Ich kann nicht für alle in diesem Buch beschriebenen Methoden ein Experte mit jahrelanger Erfahrung sein. Um meinen eigenen Ansprüchen an ein Fachbuch gerecht zu werden, war es daher unumgänglich, Partner zur Unterstützung hinzuzuziehen. Diesen Unterstützern gilt mein besonderer Dank. Ich nenne zunächst Personen, die ganze Kapitel (mit)gestaltet haben:

Reinhard Wilhelm, wissenschaftlicher Direktor des Leibniz-Zentrums für Informatik und Professor an der Universität des Saarlandes, hat das Kapitel über Worst Case Execution Timing Analysis verfasst.

Michael González Harbour, Professor des Computer- und Echtzeitteams der Fakultät für Computer und Elektronik der Universität Kantabrien, Spanien, hat das Kapitel über Schedulability-Analyse geschrieben.

Oliver Alt, Autor des Buchs »Modell-basierter Systemtest von Car Multimedia Systemen mit SysML« steuerte den größten Teil von Kapitel 15 bei.

Stoff und Idee für das Kapitel »Trace-Daten im Testumfeld« kamen von Alexander Weiss, Accemic GmbH, und von Martin Leucker, Professor am Institut für Softwaretechnik und Programmiersprachen an der Universität zu Lübeck.

Johannes Bergsmann, gerichtlich vereidigter Sachverständiger und Ziviltechniker für Informatik sowie Geschäftsführer des Unternehmens Software Quality Lab, hat mich durch eine Review des Kapitels 17 unterstützt und den Großteil von Kapitel 18 geschrieben.

Die Inhalte aus Kapitel 19 zum Thema Haftung stammen aus einer Publikation, die ich im Jahr 2009 gemeinsam mit dem Heidelberger Fachrechtsanwalt Tobias Sedlmeier und zuvor genanntem Johannes Bergsmann veröffentlicht habe. Für die unabhängige Durchsicht des auf Basis dieser Publikation neu geschriebenen Kapitels danke ich dem Münchner Rechtsanwalt Christian R. Kast recht herzlich.

Eine ganze Reihe von Personen hat mich durch eine Review oder Ergänzungen des Manuskripts unterstützt. Ich freue mich, dass die folgenden von mir ausgewählten Spezialisten meiner Bitte, Teile des Buchs zu prüfen und zu ergänzen, Folge geleistet haben:

Helmut Pichler, Präsident des Austrian Testing Boards, hat die Buchpassagen mit Bezug zu ISTQB reviewt.

Markus Unterauer, Trainer und Berater für Requirements Engineering, hat das Kapitel »Anforderungen und Test« kritisch gelesen.

Gernot Salzer, Professor am Institut für Computersprachen der Technischen Universität Wien, hat meine Ausführungen zu formalen Methoden korrigiert und ergänzt. Ich gebe unumwunden zu, dass formale Methoden nicht zu meiner Kernkompetenz gehören. Dementsprechend viel Rotstift fand sich in meinem Manuskript und dementsprechend kurz ist der betreffende Abschnitt in meinem Buch. Sein Kollege Georg Weissenbacher hat diese undankbare Aufgabe bei der Überarbeitung für die zweite Auflage übernommen.

Robert Mittermayr, Promovend auf dem Gebiet der Deadlock-Analyse, unterstützte mich durch eine Review der Kapitel zu Deadlocks und Race Conditions.

Daniel Kästner, Mitbegründer der AbsInt Angewandte Informatik GmbH, ergänzte für die zweite Auflage meine Ausführungen zur statischen Data-Race-Analyse um die Technik der abstrakten Interpretation.

Renate Gutjahr, Product Managerin der PLATO AG in Lübeck, sah Kapitel 14 durch und lieferte wertvolle Ergänzungen für die Abschnitte mit Bezug zur FMEA.

Matthias Daigl von der imbus AG hat mich unterstützt, indem er die Teile der zweiten Auflage des Buchs korrigierte, die Bezug zur ISO 29119 nehmen.

Aus dem Kreis der Abonnenten meines Newsletters meldeten sich viele Testleser für einzelne Kapitel dieses Buchs; Einsteiger wie ausgewiesene Experten. Für Hinweise und Kommentare aus diesem Leserkreis danke ich Oliver Bee, Frank Büchner, Christian Fuchs, Ralf Geiger, Alfred Guszmann, Wolfgang Höllmüller, Martin Horauer, Stefan Larndorfer, Sebastian Koopmann, Reinhard Meyer, Anke Mündler, Harald Nistelberger, Rudolf Ramler, Thilo Richard, Gerald Schröder, Christian Siemers, Arnd Strube, Andreas Weigl-Pollack und Bodo Wenzel.

Meiner Frau und meinen Kindern danke ich für ihre Geduld. Viele Monate lang war ich fast jede freie Minute mit dem Schreiben des Buchs beschäftigt.

Inhaltsverzeichnis

1 Einleitung 1

1.1 Motivation 1

1.2 Abgrenzung des Buchs zu ISTQB-Lehrplänen 1

1.3 Zur Gliederung dieses Buchs 2

1.4 Die wichtigsten Begriffe kurz erklärt 3
1.4.1 Definition von Fachbegriffen 3
1.4.2 Zu Definitionen und TesterInnen 5

1.5 Ein Überblick über das Umfeld des Software-Testing 5
1.5.1 Ursachen von Software-Fehlern 6
1.5.2 Warum Programmfehler nicht entdeckt werden 7
1.5.3 Angebrachter Testaufwand 8
1.5.4 Der Tester und der Testprozess 9
1.5.5 Modellieren der Software-Umgebung 10
1.5.6 Erstellen von Testfällen 12
1.5.7 Ausführen und Evaluieren der Tests 14
1.5.8 Messen des Testfortschritts 15
1.5.9 Testdesign und Testdokumentation im Software-Entwicklungsprozess 15
1.5.10 Verschiedene Teststufen und deren Zusammenspiel 16
1.5.11 Andere Verifikationsmethoden als Ergänzung zum Test 19
1.5.12 Agile Prozessmodelle 21
1.5.13 Der Software-Test in agilen Vorgehensmodellen 22
1.5.14 Wer testet die Tester? 24

2 Anforderungen und Test 25

2.1 Die Bedeutung textueller Anforderungen 25

2.2 Requirements Engineering im Projekt 26

2.3 Arten und Quellen von Anforderungen 27

2.4 Warum Anforderungen dokumentiert werden sollen 28

2.5 Die Review von Anforderungen 29
2.5.1 Testbarkeit von Anforderungen 30
2.5.2 Modifizierbarkeit und Erweiterbarkeit 31
2.5.3 Relevanz von Anforderungen 32

2.6 Der Umgang mit natürlicher Sprache 32
2.6.1 Einfache Sprache gegen Missverständnisse 32
2.6.2 Gelenkte Sprache 34

2.7 Hinweise zur Dokumentenform 35

2.8 Die Spezifikation an der Schnittstelle zum Testteam 38
2.8.1 Konfiguration von Testdesigns 38
2.8.2 Vollständigkeit von Spezifikationen 39

2.9 Werkzeuge zur Review von Anforderungen 40

2.10 Diskussion 41
2.10.1 Verifikation beim Requirements Engineering mit Augenmaß 41
2.10.2 Bewertung der Rolle des Requirements Engineering für den Testprozess 42

2.11 Fragen und Übungsaufgaben 43

3 Review des Designs 45

3.1 Ziele der Review des Architekturdesigns 45

3.2 Ziele der Review des Detaildesigns 46

3.3 Eigenschaften von gutem Software-Design 47

3.4 Hinweise zur Architektur-Design-Review 47

3.5 Embedded Design 51
3.5.1 Sicherheit, Verfügbarkeit & Co 51
3.5.2 Wartbarkeit des Geräts 51
3.5.3 Ressourcenverbrauch 52
3.5.4 Design von Echtzeitsystemen 52

3.6 Diskussion 52

3.7 Fragen und Übungsaufgaben 53

4 Automatische statische Code-Analyse 55

4.1 Motivation zum Einsatz von Analysewerkzeugen .55

4.2 Techniken von Analysewerkzeugen im unteren Preissegment 56
4.2.1 Sprachspezifische Fallstricke 58
4.2.2 Kontrollflussanalyse 59
4.2.3 Datenflussanalyse, Initialisation Tracking .60
4.2.4 Datenflussanalyse, Value Tracking 61
4.2.5 Semantische Analyse 62
4.2.6 Starke Typenprüfung 64

4.3 Techniken von Analysewerkzeugen im oberen Preissegment 64
4.3.1 Größerer Komfort für den Benutzer 65
4.3.2 Concurrency Checks 66
4.3.3 Stack-Analyse und erweiterte Kontrollflussanalyse 66
4.3.4 Erschöpfende Analyse des Zustandsbaums 67

4.4 Statische Security-Analyse (SSA) 67

4.5 Code-Metriken 69

4.6 Werkzeuge für die Automatische Code-Analyse .71

4.7 Diskussion 73

4.8 Fragen und Übungsaufgaben 74

5 Code-Reviews 75

5.1 Review-Arten 75
5.1.1 Code-Inspektionen 75
5.1.2 Walkthrough 76
5.1.3 Peer-Review 77

5.2 Pair Programming 78

5.3 Werkzeuge zur Code-Review 78

5.4 Diskussion 81

5.5 Fragen und Übungsaufgaben 84

6 Unit-Tests 85

6.1 Der Unit-Test im Entwicklungsprozess 85

6.2 Zur Definition von Unit-Test und Modultest 86

6.3 Black-Box-Testfälle beim White-Box-Test 86
6.3.1 Äquivalenzklassenbildung 87
6.3.2 Grenzwertanalyse 88
6.3.3 Andere Methoden 89

6.4 Stubs und Treiber 90

6.5 Verschiedene Typen von Werkzeugen beim White-Box-Test 98
6.5.1 Unit-Test-Frameworks 98
6.5.2 Werkzeuge zur Testerstellung 99
6.5.3 Werkzeuge zur Messung der Testabdeckung 101

6.6 Testabdeckung 102
6.6.1 Statement Coverage 102
6.6.2 Branch Coverage und Decision Coverage 103
6.6.3 Decision/Condition Coverage 104
6.6.4 Modified Condition/Decision Coverage 104
6.6.5 Andere Testabdeckungen 105
6.6.6 Testabdeckung bei modellbasierter Entwicklung 105
6.6.7 Messung der Testabdeckung 105

6.7 Basis Path Testing 107

6.8 Host oder Target Testing? 109

6.9 Den Code immer unverändert testen? 110

6.10 Unit-Tests bei objektorientierten Sprachen 111

6.11 Grenzen des Unit-Tests 112

6.12 Werkzeuge für den Unit-Test 113
6.12.1 Unit-Test-Frameworks 113
6.12.2 Werkzeuge zur Testerstellung 113
6.12.3 Coverage-Analyse 116

6.13 Diskussion 116
6.13.1 Testabdeckung 116
6.13.2 Organisation von Unit-Tests 118

6.14 Fragen und Übungsaufgaben 119

7 Integrationstests 123

7.1 Software/Software-Integrationstest 123
7.1.1 Bottom-up-Unit-Tests als Integrationstest 123
7.1.2 Strukturierter Integrationstest 126
7.1.3 Testabdeckung der Aufrufe von Unterprogrammen 128
7.1.4 Vergleich der Teststrategien 130
7.1.5 Grenzen des Software/Software-Integrationstests 132
7.1.6 Diskussion des Software/Software-Integrationstests 133

7.2 Ressourcentests 135
7.2.1 Statischer Ressourcentest 135
7.2.2 Dynamischer Ressourcentest 135

7.3 Hardware/Software-Integrationstest 138
7.3.1 Bottom-up-Verfahren 139
7.3.2 Regressionsverfahren 139
7.3.3 Black-Box-Verfahren 139
7.3.4 Test und Analysen bei Sicherheitsrelevanz 140
7.3.5 Diskussion des Hardware/Software-Integrationstests 140

7.4 Systemintegrationstest 141

7.5 Werkzeuge für den Integrationstest 142

7.6 Fragen und Übungsaufgaben 142

8 Systemtests 145

8.1 Funktionale Systemtests 145
8.1.1 Zuordnung funktionaler Systemtests zu Anforderungen 145
8.1.2 Äquivalenzklassen und Grenzwerte im Black-Box-Test 146
8.1.3 Zustandsbasierter Test 149
8.1.4 Ursache-Wirkungs-Analyse 153
8.1.5 CECIL-Methode 162
8.1.6 Entscheidungstabellentechnik 163
8.1.7 Paarweises Testen und Klassifikationsbaum-Methode 163
8.1.8 Back To Back Testing 165
8.1.9 Erfahrungsbasierter Test 165
8.1.10 Diskussion des Black-Box-Tests 167
8.1.11 Auswahl eines Black-Box-Testverfahrens für eine Aufgabe 167
8.1.12 Werkzeuge für Funktionstests 168

8.2 Test der Benutzerschnittstelle 169
8.2.1 Grafische Benutzerschnittstelle 169
8.2.2 Werkzeuge für GUI-Tests 169
8.2.3 Eingebettete Benutzerschnittstellen 171
8.2.4 Werkzeuge für den Test von eingebetteten Benutzerschnittstellen 172

8.3 Performanztest und Lasttest 173

8.4 Stresstest 174

8.5 Volumentest 175

8.6 Failover und Recovery Testing 175

8.7 Ressourcentests 177

8.8 Installationstests 178

8.9 Konfigurationstests 179

8.10 Security-Tests 180

8.11 Dokumententests 182

8.12 Testumgebung und Testdaten 183

8.13 Formale Methoden 183
8.13.1 Symbolischer Test 184
8.13.2 Deduktive Verifikation von funktionalen Anforderungen 184
8.13.3 Model Checking 185

8.14 Automation von Systemtests 187
8.14.1 Vor- und Nachteile der Testautomation 188
8.14.2 Tipps zur Automation von Systemtests 189

8.15 Dokumentation des Testdesigns und der Testergebnisse 194

8.16 Grenzen des Systemtests 195

8.17 Fragen und Übungsaufgaben 195

9 Testen von RTOS und Middleware 199

9.1 Definition und Motivation 199

9.2 White-Box-Requirements-Test 200

9.3 Test eines Interrupt-Managers 201

9.4 Test eines Schedulers 202

9.5 Fragen und Übungsaufgaben 204

10 Race Conditions 205

10.1 Definition von Data Races 205

10.2 Dynamische Data-Race-Analyse 210
10.2.1 Eraser 210
10.2.2 Lamports Happens-Before-Relation 213

10.3 Statische Data-Race-Analyse 216
10.3.1 Ansätze zur statischen Data-Race-Analyse 216
10.3.2 Vergleich zur dynamischen Data-Race-Analyse 218
10.4 Werkzeuge für die Data-Race-Analyse 218
10.5 Diskussion 219
10.6 Fragen und Übungsaufgaben 221

11 Deadlocks 223

11.1 Über die Entstehung von Deadlocks 223
11.2 Verschiedene Arten der Deadlock-Analyse 224
11.3 Dynamische Deadlock-Analyse 225
11.4 Statische Deadlock-Analyse 225
11.5 Werkzeuge zur Deadlock-Detektion 226
11.6 Diskussion 227
11.7 Fragen und Übungsaufgaben 227

12 Echtzeit-Verifikation 229

12.1 Antwortzeiten bei funktionalen Tests 229
12.2 WCET-Analyse 230
12.2.1 Problemstellung 230
12.2.2 Laufzeitanalyse 232
12.3 Werkzeuge für die WCET-Analyse 235
12.4 Diskussion 236
12.5 Fragen und Übungsaufgaben 237

13 Schedulability-Analyse 239

13.1 Aufgaben der Schedulability-Analyse 239
13.2 Definitionen 240
13.3 Diskussion der Scheduling-Strategien 241
13.3.1 Statisches Scheduling 242
13.3.2 Dynamisches Scheduling 243

13.4 Analyse bei Fixed-Priority-Single-CPU-Systemen ... 245
13.4.1 Optimale Prioritätsvergabe ... 245
13.4.2 Rate Monotonic Analysis ... 246
13.4.3 Exakte Antwortzeitenanalyse ... 247
13.4.4 Gegenseitiger Ausschluss ... 252
13.4.5 Aperiodische Aufgaben ... 254
13.4.6 Kontextwechsel ... 255
13.4.7 Cache und Out Of Order Execution ... 255
13.4.8 Input-Jitter ... 255
13.4.9 Interrupts ... 256

13.5 Multi-CPU-Systeme ... 256
13.5.1 Multicore- und Multiprozessor-Systeme ... 257
13.5.2 Verteilte Systeme ... 257

13.6 Scheduling-Analyse für CAN ... 259

13.7 Werkzeuge ... 261

13.8 Diskussion ... 262

13.9 Fragen und Übungsaufgaben ... 263

14 Hardware/Software-Interaktionsanalyse 265

14.1 Die FMEA als Grundlage der HSIA ... 265

14.2 Die HSIA als Quelle für Software-Anforderungen ... 269

14.3 Software-Kritikalitätsanalyse ... 270

14.4 Software-FMEA ... 271

14.5 Werkzeuge ... 272

14.6 Diskussion ... 273

14.7 Fragen und Übungsaufgaben ... 273

15 Modellbasierter Test 275

15.1 Begriffsdefinition ... 275

15.2 MBT und Testautomation ... 276

15.3 Modelle ... 276
15.3.1 Statecharts ... 276
15.3.2 SDL ... 276
15.3.3 Message Sequence Charts ... 277
15.3.4 UML Version 2 ... 277
15.3.5 SysML ... 278
15.3.6 Funktionsmodellierung ... 278

15.4 Testmodell vs. Implementierungsmodell 278
15.5 Werkzeuge 279
15.6 Diskussion 280
15.7 Fragen und Übungsaufgaben 280

16 Trace-Daten im Testumfeld 281

16.1 Das Dilemma mit instrumentiertem Code 281
16.2 Embedded-Trace-Schnittstellen 282
16.3 Werkzeuge 283
16.4 Diskussion 283

17 Testmanagement 287

17.1 Testplanung 287
17.2 Teststeuerung 289
17.3 Abweichungsmanagement 291
17.4 Bewertung und Anpassung des Testprozesses 293
17.4.1 Formale Reifegradmodelle für den Software-Test 293
17.4.2 Prozessbewertung in agilen Projekten 294
17.4.3 Mit Augenmaß ins Kostenoptimum 294
17.5 Risikobasierter Test 297
17.6 Werkzeuge 300
17.7 Diskussion 300
17.8 Fragen und Übungsaufgaben 302

18 Qualitätsmanagement 303

18.1 Definition 303
18.2 Qualitätsmanagement-Standards 304
18.3 Kosten und Haftungsrelevanz des QM 307
18.4 Umsetzung des Qualitätsmanagements 308
18.5 Die Rolle des Qualitätsmanagers 309
18.6 Mit Metriken die Qualität steuern 310
18.7 Die Wirtschaftlichkeit von QM 313

18.8 Werkzeuge 314

18.9 Diskussion 314

18.10 Fragen und Übungsaufgaben 315

19 Software-Test und Haftungsrisiko 317

19.1 Ein Software-Fehler im Sinne des Gesetzes 317

19.2 Vertragliche Gewährleistung und Haftung 318

19.3 Vertragliche Beschränkung der Haftung 319

19.4 Produzentenhaftung bei Software 320

19.5 Produkthaftung 320

19.6 Sorgfaltspflicht des Software-Herstellers 321

19.7 Technische Normen mit Bezug zum Software-Test 324
19.7.1 DIN IEC 56/575/CD 324
19.7.2 IEEE Std 1012 324
19.7.3 IEEE Std 829 325
19.7.4 IEEE Std 1008-1987 325
19.7.5 ISO/IEC 29119 326
19.7.6 IEC/EN 61508 330
19.7.7 ISO 26262 333
19.7.8 Normenreihe 250XX 333

19.8 Tipps vom Rechtsanwalt und vom Techniker 336

19.9 Fragen und Übungsaufgaben 338

Nachwort 339

Anhang

Anhang A – Lösungen zu den Übungsaufgaben 343

Anhang B – Dokumentation des Testdesigns 369

Anhang C – Software-Verifikationsplan 371

Anhang D –Software-Verifikationsreport 375

Quellenverzeichnis 377

Index 387

1 Einleitung

Fast jeder fortgeschrittene Programmierer hat ein oder mehrere Bücher über Programmiersprachen und Software-Entwicklung gelesen. Bücher über das Testen von Software stehen aber weit weniger oft in den Regalen. Das liegt vermutlich unter anderem am hartnäckigen Gerücht, dass Testen von Software langweilig sei. Um diesem Gerücht entschieden entgegenzutreten, ist dieses Kapitel geschrieben worden. Es soll ein Appetitanreger auf die im Buch behandelten Testthemen sein, beschreibt die Eingliederung des Tests in den Software-Entwicklungsprozess und analysiert zunächst einmal den Feind: den Software-Fehler. Doch zuvor noch ein paar Worte zur Motivation des Software-Tests, zur Eingliederung dieses Buchs in andere Literatur und ein paar Definitionen.

1.1 Motivation

Fehler in der Software eingebetteter Systeme können teure Rückholaktionen zur Folge haben. Selten ist es den Anbietern von eingebetteten Systemen möglich, einfach einen Bugfix per E-Mail zu verschicken und wieder dem Tagesgeschäft nachzugehen. Daher sollte man in Branchen mit hoher Anforderung an die Software-Integrität erstens besonders bedacht sein, Software-Fehler zu vermeiden, und zweitens, gemachte Fehler zu erkennen. Punkt eins zielt auf die Prozesslandschaft, die Qualifikation der Mitarbeiter und die Unternehmensstruktur ab. Er betrachtet also, wie sehr das Talent oder – viel wichtiger – das fehlende Talent eines Mitarbeiters die Qualität des Produkts beeinflussen kann. Punkt zwei heißt, die Software und begleitende Dokumente sorgfältig zu verifizieren. Solche Dokumente können zum Beispiel die Anforderungsdefinition, Analysen der zu erwartenden CPU-Last oder das Design festhalten.

1.2 Abgrenzung des Buchs zu ISTQB-Lehrplänen

Leider wurde lange Zeit nicht nur im deutschsprachigen Raum das Thema Verifikation von Software, wozu auch Testen gehört, wenig an Hochschulen gelehrt. Eine Konsequenz daraus ist, dass Vertreter der Industrie das International Soft-

ware Testing Qualifications Board (ISTQB) ins Leben gerufen haben, das eine »Certified Tester«-Ausbildung definiert. Der Inhalt des vorliegenden Buchs deckt sich *nicht* mit den Lehrplänen des ISTQB. In diesem Buch werden Themen genauer als durch das ISTQB behandelt, wenn sie für Embedded-Software besonders wichtig sind, und es werden Methoden präsentiert, die für eingebettete Software wichtig sein können, sich aber zurzeit nicht in den ISTQB-Lehrplänen befinden. Ebenso werden Themen der ISTQB-Lehrpläne hier nicht behandelt, wenn sie nicht technischer Natur sind oder nur Multisysteme oder reine Business-Anwendungen betreffen.

Das Buch ist für Personen *ohne* Vorwissen aus dem Bereich Software-Testing geschrieben. Absolventen der ISTQB-Certified-Tester-Lehrgänge werden im vorliegenden Buch daher viel Bekanntes wiederfinden und aus oben beschriebenen Gründen trotzdem ebenso viel Neues erfahren.

1.3 Zur Gliederung dieses Buchs

Die technischen Grundlagenkapitel dieses Buchs orientieren sich dabei am zeitlichen Verlauf eines Projekts. Das heißt, auch wenn das wichtigste Thema dieses Buchs der Test ist, handeln die ersten der folgenden Kapitel noch nicht vom Test, denn am Anfang eines Projekts gibt es zunächst noch keine Software, die man testen könnte. Wohl aber gibt es schon Dokumente (bei phasenorientierten Projekten) oder Vorgänge (bei agilen Vorgehensweisen), für die ein Tester einen Beitrag zur Software-Qualität leisten kann.

Kapitel 2 beschreibt, wie dieser Beitrag bei der Review von Anforderungstexten aussehen kann. Kapitel 3 beschreibt kurz die möglichen Verifikationsschritte beim Design. In den Kapiteln 4 und 5 wird beschrieben, wie Code-Reviews durchgeführt werden können und wie Werkzeuge funktionieren, die teilweise oder gänzlich automatisch Review-Aufgaben übernehmen können.

Dann erst geht es mit dem Testen los. In Kapitel 6 werden Unit-Tests, in Kapitel 7 Integrationstests und in Kapitel 8 Systemtests beschrieben. Bei Tests von Middleware können diese Teststufen alle stark verschwimmen, wie Kapitel 9 zeigt. Im Anschluss daran beschäftigt sich Kapitel 10 mit einer Art von Fehler, die nur durch Zufall in den zuvor beschriebenen Tests gefunden werden kann: Race Conditions. Das Kapitel zeigt, wie sich Data Races zuverlässig auffinden lassen, gefolgt von einem verwandten Thema: Kapitel 11 zeigt, wie sich Deadlocks automatisch auffinden lassen.

Das darauf folgende Kapitel 12 beschreibt Verfahren zur Bestimmung der maximalen Ausführungszeit von Code. Das Ergebnis kann ein wichtiger Beitrag für die Schedulability-Analyse sein, die in Kapitel 13 behandelt wird.

Spätestens nach diesem Kapitel hält sich die Reihung der weiteren Kapitel nicht mehr an den zeitlichen Verlauf eines Projekts. Die nun folgenden Kapitel sind auch weit weniger umfangreich. In Kapitel 14 wird die Hardware/Software-

Interaktionsanalyse vorgestellt, die auf Produkt/System-Ebene stattfindet. Mit einem kurzen Kapitel 15 über modellbasierten Test und einem technischen Ausblick in Kapitel 16 verlässt das Buch das technische Terrain.

In Kapitel 17 werden Hinweise für das Testmanagement gegeben und Kapitel 18 beschreibt die Aufgaben und möglichen Vorgehensweisen des Qualitätsbeauftragten. Auch diese beiden Kapitel sind bewusst kurz gehalten und geben eher nützliche Tipps, als dass man lange Abhandlungen darin findet, denn zu diesen Themen gibt es jede Menge Spezialliteratur. Das Kapitel 19 widmet sich dem Thema Haftung: In welchem Maß ist ein Programmierer oder ein Tester für Fehler haftbar? Dort erfahren Sie es.

Die meisten Kapitel schließen mit einem Fragenkatalog und Übungsaufgaben zur Kontrolle des Lernziels. Am Ende des Buchs finden Sie Lösungen dazu und Quellenverzeichnisse für referenzierte Literatur.

1.4 Die wichtigsten Begriffe kurz erklärt

1.4.1 Definition von Fachbegriffen

Die ISO 29119-1 und das ISTQB-Glossary of Terms definieren eine ganze Reihe von Fachwörtern rund ums Testen. In Einzelfällen sind sie allerdings nicht übereinstimmend. Dabei ist zu erwarten, dass das ISTQB-Glossar früher oder später an die ISO 29119-1 angepasst wird. Wenn man in einem Unternehmen Dinge rund um das Thema Software-Qualität benennen möchte, ist man gut beraten, sich an diesen Definitionen zu orientieren. Das ISTQB-Glossar kann man im WWW nachschlagen, siehe [URL: ISTQB] für die englische Fassung und [URL: ISTBQ/D] für die deutschsprachige, die ISO-Norm ist kostenpflichtig.

Es ergibt nicht viel Sinn, diese Definitionen hier zu kopieren. Stattdessen enthält die folgende Aufstellung die wichtigsten Begriffe, die im vorliegenden Buch Verwendung finden, unmittelbar gefolgt von der Definition aus dem ISTQB-Glossar und einer kurzen Ergänzung:

Agile Software-Entwicklung ist eine auf iterativer und inkrementeller Entwicklung basierende Gruppe von Software-Entwicklungsmethoden, wobei sich Anforderungen und Lösungen durch die Zusammenarbeit von selbstorganisierenden funktionsübergreifenden Teams entwickeln.

Der mit großem Abstand bedeutendste Vertreter agiler Entwicklungsmodelle ist Scrum, gefolgt von Extreme Programming. Speziell in Projekten mit begrenzter Größe und nur vage definierten Anforderungen haben diese Entwicklungsmodelle ihre Berechtigung. Diese »agilen Methoden« kommen in Reinform ohne Projektleiter aus. Die Teams sind selbstorganisierend.

Audit: Ein unabhängiges Prüfen von Software-Produkten und -prozessen, um die Konformität mit Standards, Richtlinien, Spezifikationen und/oder Prozeduren basierend auf objektiven Kriterien zu bestimmen, einschließlich der Dokumente, die (1) die Gestaltung oder den Inhalt der zu erstellenden Produkte festlegen, (2) den Prozess der Erstellung der Produkte beschreiben (3) und spezifizieren, wie die Übereinstimmung mit den Standards und Richtlinien nachgewiesen beziehungsweise gemessen werden kann.

Ein Audit führt also ein externer Fachmann durch, der sich nicht nur das Produkt ansieht, sondern auch und vor allem den Entstehungsprozess des Produkts. Ein Auditor prüft, ob ein Unternehmen ein Prozessumfeld schafft, das die Wahrscheinlichkeit von guter (beziehungsweise angepasster) Software-Qualität maximiert.

Validierung ist die Bestätigung durch Bereitstellung eines objektiven Nachweises, dass die Anforderungen für einen spezifischen beabsichtigten Gebrauch oder eine spezifische beabsichtigte Anwendung erfüllt worden sind.

Wer diese Definition aus der deutschen Version des ISTQB-Software-Testing-Glossars beim ersten Durchlesen versteht, gewinnt einen Preis. Weiter auf Deutsch übersetzt könnte man sagen: Validierung bestätigt, dass das Produkt, so wie es vorliegt, seinen beabsichtigten Verwendungszweck erfüllen kann. Validierung stellt also sicher, dass »das richtige Ding erzeugt wird«. Etwas weniger streng definierte die nun abgelöste IEEE 829-2008 diesen Begriff und verstand Validierung einfach als Nachweis der Erfüllung der Anforderungen.

Verifikation ist die Bestätigung durch Bereitstellung eines objektiven Nachweises, dass festgelegte Anforderungen erfüllt worden sind.

Das IEEE Glossary of Software Engineering Terminology [IEEE 610.12] beschränkt den Begriff nicht auf Anforderungen, sondern bezieht ihn auf Entwicklungsphasen: »Verifikation ist die Prüfung eines Systems oder einer Komponente, mit dem Ziel zu bestimmen, ob die Produkte einer Entwicklungsphase die Vorgaben erfüllen, die zum Start der Phase auferlegt wurden.« Verifikation subsumiert gemäß IEEE also alle Techniken, die sicherstellen, dass man »das Ding richtig erzeugt«. Zu diesen Techniken gehören der Test des Produkts, Reviews, die Sicherstellung eines Zusammenhangs von Design, Anforderungen und Tests (Traceability) sowie Audits [PSS-05-0].

Im allgemeinen Sprachgebrauch kann man ein Ding nicht testen, das es noch nicht gibt. Das ISTQB-Glossar hat eine Definition des Begriffs Testen, der dem allgemeinen Sprachgebrauch widerspricht:

Test: Der Prozess, der aus allen Aktivitäten des Lebenszyklus besteht (sowohl statisch als auch dynamisch), die sich mit der Planung, Vorbereitung und Bewertung eines Software-Produkts und dazugehöriger Arbeitsergebnisse befassen. Ziel des Prozesses ist sicherzustellen, dass diese allen festgelegten Anforderungen genügen, dass sie ihren Zweck erfüllen, und etwaige Fehlerzustände zu finden.

Bei dieser Definition subsumiert der Begriff Test Aktivitäten, die in der älteren Definition von IEEE der Verifikation zugeordnet werden, und enthält auch Planungsschritte für das Software-Produkt. So weit gefasst ist dann fast alle Projektarbeit, außer der Produktentwicklung selbst, ein Test.

In diesem Buch wird, so wie bei der Definition von IEEE, dann vom *Test* gesprochen, wenn das Produkt Software, zumindest in Teilen, Gegenstand des Tests ist. Alle Schritte zur »Bewertung eines Software-Produkts und dazugehöriger Arbeitsergebnisse« davor werden aber nicht »Test« genannt, damit dem allgemeinen Sprachgebrauch und den IEEE-Standards nicht widersprochen wird.

Ansonsten entspricht die Verwendung von Fachwörtern in diesem Buch den Definitionen des ISTQB-Glossars und der ISO 29119-1. Und Hand aufs Herz: Ob man die Review dem Test (wie ISTQB) oder der Verifikation (wie IEEE) zuordnet, ist wohl weniger wichtig, als dass man sie ordentlich macht.

1.4.2 Zu Definitionen und TesterInnen

Mit der Niederschrift der Definition der verwendeten Fachbegriffe *zu Beginn* eines Dokuments sehen Sie eine Praxis, die auch im Umfeld der Software-Entwicklung gut aufgehoben ist und in einschlägigen Standards so empfohlen wird, zum Beispiel [IEEE 829]. Welche Zeitverschwendung, wenn Sie als Leser einen Begriff anders verstehen als der Autor, aber erst im Laufe des Lesens dahinter kommen und deshalb zum besseren Verständnis die ersten Passagen des Buchs noch einmal lesen müssen!

Apropos Missverständnis: Wenn in diesem Buch vom Beruf des Testers oder von anderen Rollen in der Software-Entwicklung gesprochen wird, so wird immer die männliche Form verwendet. Dies soll nur der Leserlichkeit dienen, aber in keiner Weise die Frauen diskriminieren. Ein Trost für alle Leserinnen: Wahre Top-Experten nennt man Koryphäen, auch wenn es Männer sind. Und zum Wort »Koryphäe« gibt es keine männliche Form.

1.5 Ein Überblick über das Umfeld des Software-Testing

In diesem Teil der Einleitung wird das Umfeld des Software-Tests erörtert und ein Überblick über Möglichkeiten und Grenzen des Testens gegeben. Für das Verstehen der späteren Kapitel ist das Lesen *nicht* notwendig, weil große Teile von Abschnitt 1.5 redundant zum Inhalt der vertiefenden Kapitel sind. Lesern, die

sich schon mit dem Thema Test auseinandergesetzt haben und die wissen, wie sie eine Brücke vom V-Modell zu agilen Methoden schlagen können, wird daher empfohlen, bei Kapitel 2 weiterzulesen.

Ohne Zweifel helfen die folgenden Seiten aber Neueinsteigern, die später vorgestellten Methoden im Software-Entwicklungsprozess besser einzuordnen. Und – wie gesagt – diese Seiten sind ein Appetitanreger auf das, was später noch im Buch behandelt wird.

Der Protagonist schlechthin im Umfeld des Software-Tests ist der Software-Fehler. Als Einstieg in das Thema Testen werden wir daher zunächst seine möglichen Ursachen untersuchen.

1.5.1 Ursachen von Software-Fehlern

Warum hat Software überhaupt Fehler? Die einfachste Antwort auf diese Frage ist, dass Software von Menschen geschrieben wird, und Menschen machen nun einmal Fehler. Wenn die Antwort auf eine komplexe Frage sehr einfach ist, dann übersieht sie meistens viele Facetten des zugrunde liegenden Problems. So auch in diesem Fall. Viele Software-Fehler, die sehr viel Geld kosteten, waren alles andere als einfache Programmierfehler. Die folgende Aufzählung skizziert einige typische, aber sehr unterschiedliche Ursachen für Fehler in Software.

- *Fehlerhafte Kommunikation oder keine Kommunikation bei der Anforderungsspezifikation.* In einem Projekt würde das bedeuten, dass nie ganz genau klar war, was die Software tun soll und was nicht. Oder, noch tückischer: Auftraggeber und Entwickler verstehen die Niederschrift der Anforderung unterschiedlich und sind sich der Möglichkeit einer anderen Interpretation gar nicht bewusst.
- *Psychologische oder kulturelle Gründe* können Auslöser für Kommunikationsprobleme sein: Manchen Personen fällt es schwer zuzugeben, wenn sie die Aufgabenstellung nicht verstehen. In manchen Kulturkreisen ist es üblich, aus Höflichkeit so zu tun, als verstünde man alles. Nicht selten hört man von euphorischen Verkäufern, die dem Kunden Produkteigenschaften zusichern, deren Realisierung nicht oder nur teilweise möglich ist. Mit Recht reklamiert der Kunde dann die entstehende Abweichung als Fehler.
- *Sich ändernde Anforderungen.* In welcher Softwareversion ist welche Version welcher Anforderung realisiert? Welche Tests müssen aufgrund einer Änderung modifiziert, welche Tests neu durchlaufen werden?
- *Softwarekomplexität.* Applikationen benutzen Module von Zulieferern oder anderen Projekten, bedienen sich komplizierter Algorithmen, unterstützen eine Vielzahl von Plattformen und so weiter. In großen Projekten können wenige Entwickler behaupten, die von ihnen erzeugte Software in allen Details zu verstehen.

- *Zeitdruck*. Entwicklungszeitpläne beruhen alle auf Aufwandsschätzungen. Wenn man hier sehr daneben lag und der Abgabetermin näher rückt, dann ist Eile angesagt. Zeitdruck ist ein perfekter Nährboden für Fehler.
- *Schlecht dokumentierter Code*. Der Programmierer, dessen Code im neuen Projekt wiederverwendet werden muss, war kein Freund der Dokumentation. Das Programm war für ihn schwer zu schreiben. Aus seiner Sicht ist es daher zumutbar, wenn auch die Dokumentation schwer zu lesen ist.
- *Programmierfehler.* Programmierer sind auch nur Menschen und machen Fehler im Software-Design oder bei der Umsetzung des Designs.

Ein Software-Fehler hat also viele verschiedene Ursachen. Tests sind nur geeignet, klassische Programmier- und Designfehler zu finden und sind bei allen, bis auf die letztgenannte Fehlerursache ein vergleichsweise wirkungsloses Instrument. Wenn zum Beispiel ein Kommunikationsproblem vorliegt und daher gegen eine falsche Anforderungsspezifikation getestet wird, dann nützt der gewissenhafteste Test nichts. Bis auf den Programmierfehler kann man aber allen anderen genannten Fehlerursachen durch Methoden des Managements begegnen.

Mit dem Wissen, dass Testen nur eine von vielen Maßnahmen im Kampf gegen Software-Fehler ist, nehmen wir nun den Programmierfehler, den Software-»Bug«, genauer ins Visier und erörtern, warum es so schwierig ist, alle Bugs zu finden. Die Bezeichnung Bug stammt übrigens laut Gerüchteküche von einer Motte, die auf einer Speicherplatine des Computers eines Zerstörers der US-Navy landete. Der resultierende Kurzschluss grillte nicht nur das unglückliche Insekt, sondern führte auch zu einer Fehlfunktion des Computers. Seitdem werden Software-Fehler nach dem Insekt benannt.

1.5.2 Warum Programmfehler nicht entdeckt werden

Vielen Software-Entwicklern ist Folgendes nicht unbekannt: Der Kunde meldet einen Programmierfehler, obwohl unzählige Stunden mit gewissenhafter Code-Review verbracht wurden, obwohl tagelang getestet wurde. Wie konnte dieser Fehler unbemerkt das Haus verlassen?

Unter der Annahme, dass sich der Kunde nicht irrt, kann einer der folgenden Punkte diese Frage beantworten:

- *Der Kunde führte Programmteile aus, die noch nie getestet wurden*. Entweder absichtlich (bei Zeitdruck oder Kostendruck) oder durch Unachtsamkeit wurden Programmteile nicht oder unzureichend getestet.
- *Die Reihenfolge, in der der Kunde Programmanweisungen ausführte, ist anders als die Reihenfolge, in der die Anweisungen getestet wurden*. Die Reihenfolge kann aber über Funktion und Fehler entscheiden.

- *Der Kunde verwendete eine Kombination von Eingangsgrößen, die nie getestet wurden.* Software wird in den seltensten Fällen mit allen möglichen Eingangsgrößen getestet. Der Tester muss daher eine geringe Zahl von Kombinationen selektieren und daraus schließen, dass die anderen Kombinationen auch funktionieren. Wenn dieser Schluss falsch ist, rutscht der Fehler durch.
- *Die operative Umgebung der Software ist beim Kunden eine andere.* Der Kunde verwendet eine andere Betriebssystemversion oder eine andere Hardware. Vielleicht stand dem Testteam die Anwendungsumgebung überhaupt nicht zur Verfügung und man musste diese simulieren oder von Annahmen ausgehen.

Software wird fast nie zu 100 % getestet. Das gilt auch für sicherheitskritische Anwendungen. [Hayhurst 01] beschreibt zum Beispiel, dass Flugkontrollsoftware bis zu 36 verschiedene Eingangsgrößen verarbeitet. Wollten wir alle möglichen Eingangskombinationen durchtesten und damit beweisen, dass keine ungewollten Wechselbeziehungen zwischen den Eingängen bestehen, so müssten wir 21 Jahre lang testen, selbst wenn wir pro Sekunde 100 Testfälle erstellen und durchführen könnten. In Kapitel 6 werden Techniken vorgestellt, die durch Analyse des Quellcodes den Testaufwand dramatisch reduzieren und trotzdem die Testschärfe nur gering beschneiden.

1.5.3 Angebrachter Testaufwand

Die Aufgabe des Testers ist es nun, mit den vorhandenen Ressourcen so zu testen, dass er mit größter Wahrscheinlichkeit alle Fehler findet, die in der Software stecken. Kapitel 8 wird dazu viele Techniken vorstellen. Die Aufgabe des Managements ist, dem Tester Ressourcen so zu genehmigen, dass der maximale Projektgewinn resultiert. Abbildung 1–1 zeigt dazu die anzustellende, wirtschaftliche Überlegung. Wird zu wenig gestestet und drohen dem Unternehmen daher Schadenersatzzahlungen oder Imageverlust, dann ist der Testaufwand zu erhöhen. Wird andererseits zu viel getestet, dann ist das Software-Produkt zwar vielleicht von guter Qualität, aber nicht mehr wirtschaftlich zu erzeugen.

Die Entscheidung, wie viel Aufwand in den Test zu stecken ist, ist eine rein wirtschaftliche und stützt sich im Idealfall auf sogenannte Prozessmetriken. Derartige Metriken erfassen beispielsweise wie viel Zeit (und damit Geld) es kostet, einen Fehler zu finden, und wie viel es kostet, einen Fehler nicht zu finden. Prozessmetriken sind ein wichtiges Werkzeug der Wirtschaftlichkeitsanalyse und des Software-Qualitätsmanagements.

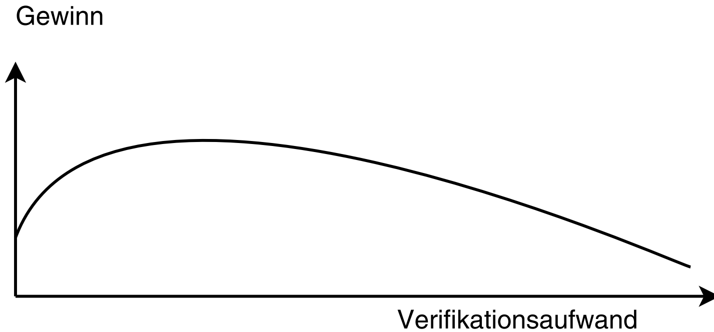

Abb. 1–1 *Der Testaufwand sollte sich am Betriebsgewinn orientieren.*

1.5.4 Der Tester und der Testprozess

Bei der Erstellung von Testfällen muss der Tester die Software, ihre Funktionalität, die Eingangsgrößen, die möglichen Kombinationen der Eingangsgrößen und die Testumgebung berücksichtigen. Dieser Prozess ist kein Kinderspiel, benötigt Zeit, Erfahrung und Planung. In der Zeitskala dieser Planung muss auch Zeit vorgesehen werden, die Testfälle zu revidieren, gefundene Bugs zu entfernen und die Software von Neuem zu testen. Im Zeitplan nur den Wonnefall vorzusehen – alles funktioniert klaglos, das Testfalldesign ist fehlerfrei –, ist ein Fehler, der Anfängern vorbehalten bleiben soll.

Vom Tester wird destruktive Kreativität verlangt, wenn seine Tests auch trickreiche Bugs finden sollen. Je mehr Freude der Tester am Finden von Fehlern hat, desto mehr wird er sich anstrengen, auch tatsächlich welche zu finden. Wenig Freude bereitet es, seine eigenen Fehler zu finden, und so ist eine der Grundregeln des Testens, dass Programmautor und -tester verschiedene Personen sein müssen. Glenford Myers, der erste nennenswerte Autor von Literatur über Software-Tests, schreibt dazu folgenden Absatz [Myers 78]:

> »*Testen ist ein destruktiver Prozess. Man versucht die Software zu brechen, ihre Schwachpunkte zu finden, Fehler aufzudecken. Es ist sehr schwierig für einen Programmierer, nachdem er während des Designs und des Programmierens konstruktiv war, plötzlich seine Perspektive zu ändern und eine destruktive Haltung gegenüber dem Programm einzunehmen. Aus diesem Grund können die meisten Programmierer ihre eigene Software nicht effektiv testen, weil sie es nicht schaffen, diese destruktive Haltung einzunehmen.*«

Um den Testprozess genauer zu beschreiben, unterteilt ihn James Whittaker in einem sehr gelungenen Überblicksartikel in vier Phasen [Whittaker 00]:

- Modellieren der Software-Umgebung
- Erstellen von Testfällen
- Ausführen und Evaluieren der Tests
- Messen des Testfortschritts

Diese Phasen und die damit verbundenen Probleme und Lösungsansätze wollen wir uns nun genauer ansehen.

1.5.5 Modellieren der Software-Umgebung

Eine der Aufgaben des Testers ist es, die Interaktion der Software mit ihrer Umgebung zu prüfen und dabei diese Umgebung zu simulieren. Das kann eine enorme Aufgabe sein, wenn man die Vielfalt von Schnittstellen bedenkt:

- Die *klassische Mensch/Maschine-Schnittstelle*: Tastatur, Bildschirm, Maus. Der Tester muss sich überlegen, wie er alle erwarteten und unerwarteten Mausklicks, Tastatureingaben und Bildschirminhalte in den Tests organisiert. Dabei können ihn Capture/Replay-Tools unterstützen, die diese Eingaben simulieren und die resultierende Bildschirmdarstellung mit einer gespeicherten Soll-Darstellung vergleichen. Der Einsatz solcher Tools ist mit einer nicht zu unterschätzenden Lernkurve verbunden. Billige Tools haben obendrein das Problem, dass man bei einem Wechsel der GUI-Version oder des Betriebssystems oft auch gleich viele Tool-Aufzeichnungen wiederholen muss.
- Das Testen der *Schnittstellen zur Hardware* kann eine Herausforderung für sich sein. Beim Test von eingebetteten Systemen muss oft erst ein Testgerät entwickelt beziehungsweise gekauft werden, damit die zu testende Software in ihrer Zielhardware auch getestet werden kann (*Hardware in the Loop*). Für die auf diesen Schnittstellen basierenden Kommunikationsprotokolle muss der Tester erwartete und unerwartete, gültige und ungültige Daten oder Kommandos versenden. Kapitel 8 zeigt Testtechniken dazu und Kapitel 14 stellt eine ergänzende Analysetechnik vor.
- Die *Schnittstelle zum Betriebssystem* sollte ebenfalls Gegenstand von Tests sein. Wenn die zu testende Software einige Dienste des Betriebssystems in Anspruch nimmt, dann muss auch geprüft werden, was passiert, wenn das Betriebssystem diese Dienste verweigert: Was ist, wenn das Speichermedium voll ist oder der Zugriff darauf misslingt? Um das ohne Kenntnis des Codes zu testen, könnte man beispielsweise Werkzeuge des Security-Testings verwenden, die die zu testende Software in einer Betriebssystem-Simulation laufen lassen [Whittaker 03]. In dieser Simulationsumgebung kann man dann zum Beispiel per Knopfdruck eine volle Festplatte simulieren. Manche sprechen in diesem Zusammenhang auch von *Sandbox Testing*.

- Auch *Dateisystem-Schnittstellen* findet man bei einigen eingebetteten Systemen. Der Tester muss Dateien mit erlaubtem und unerlaubtem Inhalt und Format bereitstellen.

Es ist unschwer zu erkennen, dass die Modellierung der Systemschnittstellen ein sehr aufwändiges Projekt werden kann. Stellen wir uns zum Beispiel Software für ein Motorsteuergerät in einem Kraftfahrzeug vor: Manche Prozesse im Motor und Interaktionen mit anderen Steuergeräten (wie ABS, ESP, ...) sind so komplex, dass sie nur unter großem Aufwand modelliert werden können. Es ist in der Testplanung abzuwägen, wie exakt das Modell der Umgebung sein soll. So muss etwa bei einem Steuergeräte-Test für Fahrzeuge geplant werden, was in einer Umgebungssimulation geprüft wird (siehe Abb. 1–2) und was etwa an einem Motorprüfstand oder in einem Testfahrzeug getestet wird (siehe Abb. 1–3).

Das Modell der Umgebung, beziehungsweise die Testausrüstung für Tests am Zielsystem, muss auch außergewöhnliche Situationen vorsehen. Etwa den Neustart der Hardware während der Kommunikation mit einem externen Gerät, die Parallelnutzung eines Betriebssystemservices durch andere Applikationen, einen Fehler im Speichermedium und so weiter.

Abb. 1–2 *Beispiel einer Testumgebung. Diese Ausrüstung wird verwendet, um Software für Steuergeräte automatisiert zu testen. Produziert die Testumgebung, wie hier, auch alle Stimuli für Sensoren und prüft sie die Funktion von Aktoren, so spricht man vom Hardware-In-The-Loop-Test. Foto © dSPACE GmbH.*

Abb. 1–3 *Eine Testschnittstelle ermöglicht während der Fahrt, Interna der Steuergeräte-Software per Laptop zu überwachen. Die Testfälle sind in diesem Testfahrzeug per Definition real. Ihr Design ist aber im Vergleich zur Simulation eingeschränkter, zum Beispiel weil Unfallszenarien nicht einfach zu durchlaufen sind.*

1.5.6 Erstellen von Testfällen

Software kann mitunter eine unendlich große Anzahl von verschiedenen Eingangsdaten verarbeiten. Denken wir nur an ein Textverarbeitungsprogramm oder an eine Signalverarbeitungssoftware, wo die Reihenfolge der Eingabe von Daten relevant ist. Die schwierige Aufgabe des Testers ist es nun, einige wenige Testszenarien zu entwerfen, die ein Programm prüfen sollen, das eigentlich unendlich viele Szenarien verarbeiten kann.

Bei der Durchführung der Tests wird man nach der strukturellen Testabdeckung (*Structural Test Coverage*) fragen: Welche Teile des Codes sind noch ungetestet? Um diese Frage zu beantworten, bedient man sich in den meisten Fällen eines Werkzeugs. Kapitel 6 wird die Funktionsweise solcher Werkzeuge erklären.

Abbildung 1–4 zeigt, wie so ein Werkzeug verschiedene Arten von Testabdeckungen nach Messung anzeigt und so den Testfortschritt visualisiert.

Mit dem Ziel, die gewünschte Testabdeckung am Quellcode zu erreichen wird der Tester daher Szenarien auswählen, die