

CEM KANER
JAMES BACH
BRET PETTICHORD

Lessons Learned



in ***SOFTWARE TESTING***

A Context-Driven Approach

"Pick up this book, open it anywhere, read it for two minutes, and take one lesson as a suggestion. Your testing, test planning, test management, or thinking about testing will improve dramatically."

Johanna Rothman, Rothman Consulting Group, Inc.

"If you test software, or depend on people who do, then read this book. Each page bubbles with hard-won advice for handling the practical problems you encounter every day."

*Sam Guckenheimer
Senior Director of Automated Testing Technology
Rational Software Corporation*

"Definitely a book worth reading and keeping around. Smart, practical, insightful and thought-provoking."

Ross Collard, Collard & Company

"These three distinguished test professionals have written a precisely-stated and thought-provoking book that offers a distinctive and important perspective on testing and test project management."

*Rex Black, Author of Managing the Testing Process
and Critical Testing Processes*

"This is the book the testing community has been looking for and didn't realize it. A must read for any test engineer or manager."

*George Hamblen Jr., Director of Software Quality Assurance
for a large financial services company*

"This isn't textbook stuff. It's better. It's real life under discussion and observation. I'm excited to see so many aspects of testing being brought together into one book. I expect great discussions to be had because of this book."

Steve Tolman, Manager of Software Quality, PowerQuest

"These lessons contain wonderful insights about software testing in the real world, from the leading practical experts on software testing. Whether you test software, or work with people who do, this book is great stuff."

Alan Myrvold

“Clear and succinct. It has brought clarity to many of my own learning experiences and provoked a lot of new thoughts.”

*Fran McKain
Software Test Manager
Hewlett-Packard Company*

“Reviewing this book was one of my greatest learning experiences. I warmly recommend this as a must-have for any testing professional.”

Hans Buwalda, Author of Integrated Test Design and Automation

“The book is packed with nuggets of gold derived from years of practical experience. The chapter on test automation alone is more useful than any of the books I’ve seen on test automation. The chapter on techniques has powerful ideas, simply stated!”

Doug Hoffman, Consultant, Software Quality Methods, LLC

“*Lessons Learned in Software Testing* is a must read for the beginner who needs tried and true tips and for the mature test manager who is looking for more refinements for his or her organization.”

Chris DeNardis, Supervisor of Software Engineering, Rockwell Automation

“ . . . offers an invaluable collection of real world practices based on years of experience shared by the authors collectively and many of their colleagues . . . an absolute must for anyone who has a serious interest in software testing.”

*Hung Q. Nguyen, President and CEO, LogiGear Corporation
Author of Testing Applications on the Web*

“The lessons format is simple and succinct, just the thing for us to use in late night test planning sessions. Where other books have been long on theory and are great for study, this is long on reality, practicality, and immediate usefulness.”

Mary Romero Sweeney, Author of Visual Basic for Testers

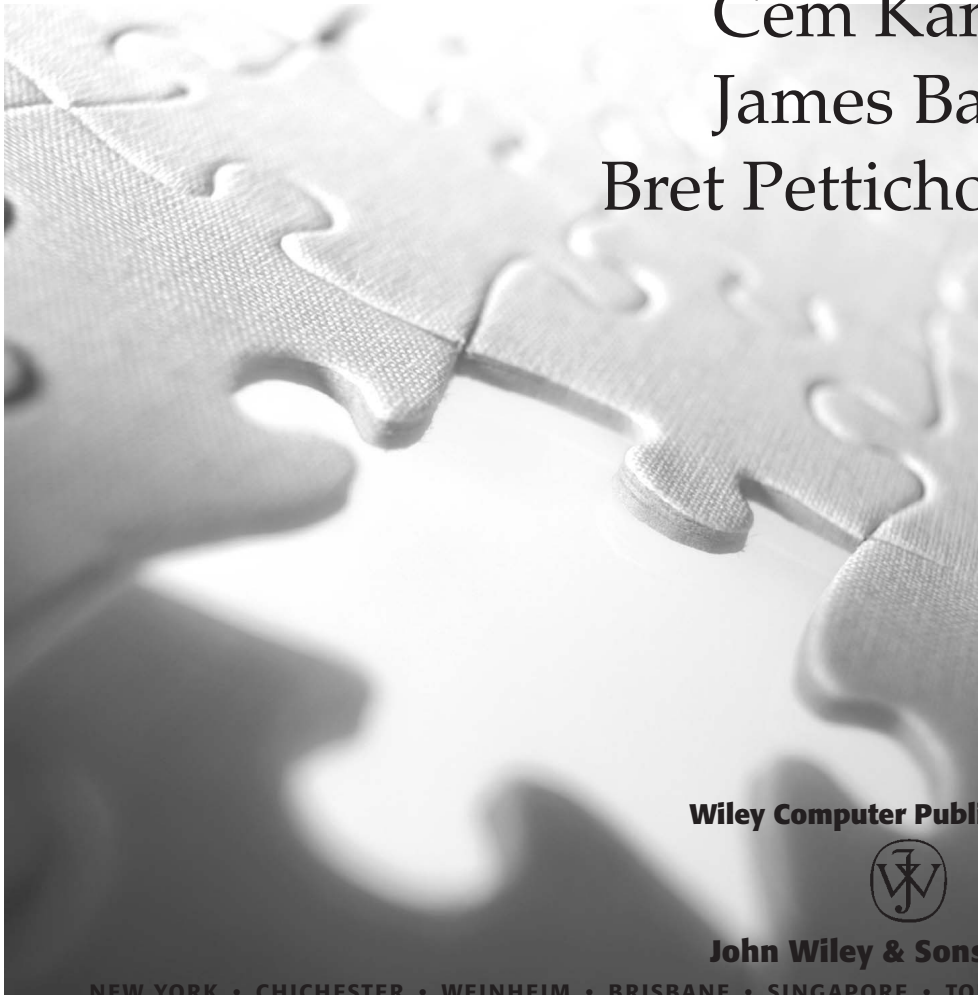
“This is an excellent book. I have had similar experiences as documented in this book, without being able to learn the lessons whilst in the middle of the problem.”

Ståle Amland, Amland Consulting, Norway

Lessons Learned in Software Testing

A Context-Driven Approach

Cem Kaner
James Bach
Bret Pettichord



Wiley Computer Publishing



John Wiley & Sons, Inc.

NEW YORK • CHICHESTER • WEINHEIM • BRISBANE • SINGAPORE • TORONTO

Publisher: Robert Ipsen
Editor: Margaret Eldridge
Assistant Editor: Adaobi Obi
Managing Editor: Micheline Frederick
Text Design & Composition: D&G Limited, LLC

Designations used by companies to distinguish their products are often claimed as trademarks. In all instances where John Wiley & Sons, Inc., is aware of a claim, the product names appear in initial capital or ALL CAPITAL LETTERS. Readers, however, should contact the appropriate companies for more complete information regarding trademarks and registration.

This book is printed on acid-free paper. ∞

Copyright © 2002 by Cem Kaner, James Bach, Bret Pettichord. All rights reserved.
Published by John Wiley & Sons, Inc., New York.
Published simultaneously in Canada.

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning, or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 750-4744. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 605 Third Avenue, New York, NY 10158-0012, (212) 850-6011, fax (212) 850-6008, E-Mail: PERMREQ @ WILEY.COM.

This publication is designed to provide accurate and authoritative information in regard to the subject matter covered. It is sold with the understanding that the publisher is not engaged in professional services. If professional advice or other expert assistance is required, the services of a competent professional person should be sought.

Library of Congress Cataloging-in-Publication Data:

Kaner, Cem.

Lessons learned in software testing : a context-driven approach / Cem Kaner, James Bach, Bret Pettichord.

p. cm.

"Wiley Computer Publishing."

Includes bibliographical references and index.

ISBN 0-471-08112-4 (pbk. : alk. paper)

1. Computer software—Testing. I. Bach, James. II. Pettichord, Bret. III. Title.

QA76.76.T48 K34 2001

005.1'4—dc21

2001046886

Printed in the United States of America.

10 9 8 7 6 5 4 3 2 1

DEDICATION

To Brian Marick and Sam Guckenheimer, who set the spark for this book.

To Dave Gelperin, who believed in us and built a community.

To Jerry Weinberg, whose life and work embodies the highest ideals of an expert tester.

In memoriam, Anna Allison, colleague and friend, September 30, 1952–September 11, 2001.

	Lessons	ix
	Foreword	xvii
	Preface	xix
	Acknowledgments	xxvii
Chapter 1	The Role of the Tester	1
Chapter 2	Thinking Like a Tester	11
Chapter 3	Testing Techniques	31
Chapter 4	Bug Advocacy	65
Chapter 5	Automating Testing	93
Chapter 6	Documenting Testing	129
Chapter 7	Interacting with Programmers	143
Chapter 8	Managing the Testing Project	151
Chapter 9	Managing the Testing Group	189
Chapter 10	Your Career in Software Testing	209
Chapter 11	Planning the Testing Strategy	231
Appendix	The Context-Driven Approach to Software Testing	261
	Bibliography	265
	Index	275

Chapter 1 The Role of the Tester

Lesson 1	You are the headlights of the project	1
Lesson 2	Your mission drives everything you do	2
Lesson 3	You serve many clients	3
Lesson 4	You discover things that will “bug” someone whose opinion matters	4
Lesson 5	Find important bugs fast	4
Lesson 6	Run with the programmers	5
Lesson 7	Question everything, but not necessarily out loud	5
Lesson 8	You focus on failure, so your clients can focus on success	6
Lesson 9	You will not find all the bugs	6
Lesson 10	Beware of testing “completely”	7
Lesson 11	You don’t <i>assure</i> quality by testing	8
Lesson 12	Never be the gatekeeper!	8
Lesson 13	Beware of the not-my-job theory of testing	8
Lesson 14	Beware of becoming a process improvement group	9
Lesson 15	Don’t expect anyone to understand testing, or what you need to do it well	10

Chapter 2 Thinking Like a Tester

Lesson 16	Testing is applied epistemology	11
Lesson 17	Studying epistemology helps you test better	12
Lesson 18	Testing is grounded in cognitive psychology	13
Lesson 19	Testing is in your head	14
Lesson 20	Testing requires inference, not just comparison of output to expected results	14
Lesson 21	Good testers think technically, creatively, critically, and practically	15
Lesson 22	Black box testing is not ignorance-based testing	15
Lesson 23	A tester is more than a tourist	16
Lesson 24	All tests are an attempt to answer some question	16
Lesson 25	All testing is based on models	17
Lesson 26	Intuition is a fine beginning, but a lousy conclusion	17
Lesson 27	To test, you must explore	17
Lesson 28	Exploring involves a lot of thinking	18
Lesson 29	Use the logic of abductive inference to discover conjectures.	19
Lesson 30	Use the logic of conjecture and refutation to evaluate a product	20
Lesson 31	A <i>requirement</i> is a quality or condition that matters to someone who matters	20

Lesson 32	You discover requirements by conference, inference, and reference	21
Lesson 33	Use implicit as well as explicit specifications	22
Lesson 34	"It works" really means it appears to meet some requirement to some degree	23
Lesson 35	In the end, all you have is an impression of the product	23
Lesson 36	Don't confuse the test with the testing	23
Lesson 37	When testing a complex product: plunge in and quit	24
Lesson 38	Use heuristics to quickly generate ideas for tests	25
Lesson 39	You can't avoid bias, but you can manage it	25
Lesson 40	You're harder to fool if you know you're a fool	26
Lesson 41	When you miss a bug, check whether the miss is surprising or just the natural outcome of your strategy	27
Lesson 42	Confusion is a test tool	27
Lesson 43	Fresh eyes find failure	28
Lesson 44	Avoid following procedures unless they followed you first	28
Lesson 45	When you do create test procedures, avoid "1287"	29
Lesson 46	One important outcome of a test process is a better, smarter tester	29
Lesson 47	You can't master testing unless you reinvent it	30

Chapter 3 Testing Techniques

Lesson 48	Testing combines techniques that focus on testers, coverage, potential problems, activities, and evaluation	32
Lesson 49	People-based techniques focus on who does the testing	34
Lesson 50	Coverage-based techniques focus on what gets tested	35
Lesson 51	Problems-based techniques focus on why you're testing (the risks you're testing for)	39
Lesson 52	Activity-based techniques focus on how you test	40
Lesson 53	Evaluation-based techniques focus on how to tell whether the test passed or failed	42
Lesson 54	The classification of a technique depends on how you think about it	43

Chapter 4 Bug Advocacy

Lesson 55	You are what you write	65
Lesson 56	Your advocacy drives the repair of the bugs you report	66
Lesson 57	Make your bug report an effective sales tool	66
Lesson 58	Your bug report is your representative	67
Lesson 59	Take the time to make your bug reports valuable	68
Lesson 60	Any stakeholder should be able to report a bug	68
Lesson 61	Be careful about rewording other people's bug reports	69
Lesson 62	Report perceived quality gaps as bugs	69
Lesson 63	Some stakeholders cannot report bugs—you're their proxy	69
Lesson 64	Draw the affected stakeholder's attention to controversial bugs	70
Lesson 65	Never use the bug-tracking system to monitor programmers' performance	70
Lesson 66	Never use the bug-tracking system to monitor testers' performance	71
Lesson 67	Report defects promptly	71
Lesson 68	Never assume that an obvious bug has already been filed	71
Lesson 69	Report design errors	71

Lesson 70	Extreme-looking bugs are potential security flaws	73
Lesson 71	Uncorner your corner cases	73
Lesson 72	Minor bugs are worth reporting and fixing	74
Lesson 73	Keep clear the difference between severity and priority	75
Lesson 74	A failure is a symptom of an error, not the error itself	76
Lesson 75	Do follow-up testing on seemingly minor coding errors	76
Lesson 76	Always report nonreproducible errors; they may be time bombs	77
Lesson 77	Nonreproducible bugs are reproducible	78
Lesson 78	Be conscious of the processing cost of your bug reports	79
Lesson 79	Give special handling to bugs related to the tools or environment	80
Lesson 80	Ask before reporting bugs against prototypes or early private versions	81
Lesson 81	Duplicate bug reports are a self-correcting problem	82
Lesson 82	Every bug deserves its own report	82
Lesson 83	The summary line is the most important line in the bug report	83
Lesson 84	Never exaggerate your bugs	83
Lesson 85	Report the problem clearly, but don't try to solve it	84
Lesson 86	Be careful of your tone. Every person you criticize will see the report	85
Lesson 87	Make your reports readable, even to people who are exhausted and cranky	85
Lesson 88	Improve your reporting skills	86
Lesson 89	Use market or support data when appropriate	86
Lesson 90	Review each other's bug reports	87
Lesson 91	Meet the programmers who will read your reports	87
Lesson 92	The best approach may be to demonstrate your bugs to the programmers	88
Lesson 93	When the programmer says it's fixed, make sure it isn't still broken	88
Lesson 94	Verify bug fixes promptly	88
Lesson 95	When fixes fail, talk with the programmer	89
Lesson 96	Bug reports should be closed by testers	89
Lesson 97	Don't insist that every bug be fixed. Pick your battles	90
Lesson 98	Don't let deferred bugs disappear	90
Lesson 99	Testing inertia should never be the cause of bug deferral	91
Lesson 100	Appeal bug deferrals immediately	91
Lesson 101	When you decide to fight, decide to win!	91

Chapter 5 Automating Testing

Lesson 102	Speed the development process instead of trying to save a few dollars on testing	94
Lesson 103	Expand your reach instead of trying to repeat the same tests over and over	95
Lesson 104	Select your automation strategy based on your context	96
Lesson 105	Don't mandate 100 percent automation	97
Lesson 106	A test tool is not a strategy	98
Lesson 107	Don't automate a mess	98
Lesson 108	Don't equate manual testing to automated testing	99
Lesson 109	Don't estimate the value of a test in terms of how often you run it	100
Lesson 110	Automated regression tests find a minority of the bugs	101
Lesson 111	Consider what bugs you <i>aren't</i> finding while you automate tests	101
Lesson 112	The problem with bad automation is that no one may notice	102
Lesson 113	Capture replay fails	103

Lesson 114	Test tools are buggy	104
Lesson 115	User interfaces change	106
Lesson 116	Select GUI test tools based on compatibility, familiarity, and service	107
Lesson 117	Automated regression tests die	108
Lesson 118	Test automation is a software development process	109
Lesson 119	Test automation is a significant investment	109
Lesson 120	Test automation projects require skills in programming, testing, and project management	110
Lesson 121	Use pilot projects to prove feasibility	111
Lesson 122	Have testers and programmers charter automation projects	111
Lesson 123	Design automated tests to facilitate review	112
Lesson 124	Don't skimp on automated test design	112
Lesson 125	Avoid complex logic in your test scripts	113
Lesson 126	Don't build test libraries simply to avoid repeating code	113
Lesson 127	Data-driven test automation makes it easy to run lots of test variants	114
Lesson 128	Keyword-driven test automation makes it easy for nonprogrammers to create tests	115
Lesson 129	Use automated techniques to generate test inputs	116
Lesson 130	Separate test generation from test execution	117
Lesson 131	Use standard scripting languages	117
Lesson 132	Automate tests using programming interfaces	119
Lesson 133	Encourage the development of unit test suites	120
Lesson 134	Beware of using automators who don't understand testing	121
Lesson 135	Avoid automators who don't respect testing	122
Lesson 136	Testability is often a better investment than automation	122
Lesson 137	Testability is visibility and control	123
Lesson 138	Start test automation early	124
Lesson 139	Give centralized automation teams clear charters	125
Lesson 140	Automate for immediate impact	126
Lesson 141	You may have more test tools than you realize	126

Chapter 6 Documenting Testing

Lesson 142	To apply a solution effectively, you need to understand the problem clearly	131
Lesson 143	Don't use test documentation templates: A template won't help unless you don't need it	131
Lesson 144	Use test documentation templates: They foster consistent communication	132
Lesson 145	Use the IEEE Standard 829 for test documentation	132
Lesson 146	Don't use the IEEE Standard 829	133
Lesson 147	Analyze your requirements before deciding what products to build; this applies as much to your documentation as to your software	136
Lesson 148	To analyze your test documentation requirements, ask questions like the ones in this list	136
Lesson 149	Summarize your core documentation requirements in one sentence with no more than three components	141

Chapter 7 Interacting with Programmers

Lesson 150	Understand how programmers think	144
Lesson 151	Develop programmers' trust	145
Lesson 152	Provide service	145
Lesson 153	Your integrity and competence will demand respect	146
Lesson 154	Focus on the work, not the person	147
Lesson 155	Programmers like to talk about their work. Ask them questions	148
Lesson 156	Programmers like to help with testability	149

Chapter 8 Managing the Testing Project

Lesson 157	Create a service culture	151
Lesson 158	Don't try to create a control culture	152
Lesson 159	Develop the power of the king's ear	152
Lesson 160	You manage the subproject that provides testing services, not the development project	153
Lesson 161	All projects evolve. Well-run projects evolve well	154
Lesson 162	There are always late changes	154
Lesson 163	Projects involve a tradeoff among features, reliability, time, and money	155
Lesson 164	Let the project manager choose the project lifecycle	156
Lesson 165	Waterfall lifecycles pit reliability against time	156
Lesson 166	Evolutionary lifecycles pit features against time	158
Lesson 167	Be willing to allocate resources to the project early in development	159
Lesson 168	Contract-driven development is different from market-seeking development	160
Lesson 169	Ask for testability features	161
Lesson 170	Negotiate the schedules for builds	161
Lesson 171	Understand what programmers do (and don't do) before delivering builds	162
Lesson 172	Be prepared for the build	162
Lesson 173	Sometimes you should refuse to test a build	162
Lesson 174	Use smoke tests to qualify a build	163
Lesson 175	Sometimes, the right decision is to stop the test and fix cycle and redesign the software	163
Lesson 176	Adapt your processes to the development practices that are actually in use	164
Lesson 177	"Project documents are interesting fictions: Useful, but never sufficient"	165
Lesson 178	Don't ask for items unless you will use them	165
Lesson 179	Take advantage of other sources of information	166
Lesson 180	Flag configuration management problems to the project manager	167
Lesson 181	Programmers are like tornadoes	168
Lesson 182	Great test planning makes late changes easy	168
Lesson 183	Test opportunities arise whenever one person hands off an artifact to another	170
Lesson 184	There is no universal formula for knowing how much testing is enough	170
Lesson 185	"Enough testing" means "enough information for my clients to make good decisions"	170
Lesson 186	Never budget for just two testing cycles	171

Lesson 187	To create a schedule for a set of tasks, estimate the amount of time needed for each task	172
Lesson 188	The person who will do the work should tell you how long a task will take	173
Lesson 189	There is no right ratio of testers to other developers	174
Lesson 190	Trade tasks or transfer people from tasks that they are failing at	174
Lesson 191	Rotate testers across features	175
Lesson 192	Try testing in pairs	175
Lesson 193	Assign a bug hunter to the project	176
Lesson 194	Charter testing sessions, especially exploratory testing sessions	176
Lesson 195	Test in sessions	177
Lesson 196	Use activity logs to reveal the interruptions that plague testers' work	177
Lesson 197	Regular status reports are powerful tools	178
Lesson 198	There's nothing more dangerous than a vice president with statistics	179
Lesson 199	Be cautious about measuring the project's progress in terms of bug counts	180
Lesson 200	The more independent coverage measures you use, the more you know	181
Lesson 201	Use a balanced scorecard to report status on multiple dimensions	182
Lesson 202	Here's a suggested structure for a weekly status report	183
Lesson 203	A project dashboard is another useful way for showing status	184
Lesson 204	Milestone reports are useful when milestones are well defined	185
Lesson 205	Don't sign-off to approve the release of a product	186
Lesson 206	Do sign-off that you have tested a product to your satisfaction	186
Lesson 207	If you write a release report, describe your testing work and results, not your opinion of the product	187
Lesson 208	List unfixed bugs in the final release report	187
Lesson 209	A useful release report lists the 10 worst things critics might say	187

Chapter 9 Managing the Testing Group

Lesson 210	Mediocrity is a self-fulfilling prophecy	189
Lesson 211	Treat your staff as executives	190
Lesson 212	Read your staff's bug reports	191
Lesson 213	Evaluate your staff as executives	191
Lesson 214	If you really want to know what's going on, test with your staff	193
Lesson 215	Don't expect people to handle multiple projects efficiently	193
Lesson 216	Build your testing staff's domain expertise	194
Lesson 217	Build your testing staff's expertise in the relevant technology	194
Lesson 218	Work actively on skills improvement	195
Lesson 219	Review technical support logs	195
Lesson 220	Help new testers succeed	195
Lesson 221	Have new testers check the documentation against the software	196
Lesson 222	Familiarize new testers with the product through positive testing	197
Lesson 223	Have novice testers edit old bug reports before writing new ones	197
Lesson 224	Have new testers retest old bugs before testing for new bugs	197
Lesson 225	Don't put novice testers on nearly finished projects	198
Lesson 226	The morale of your staff is an important asset	199
Lesson 227	Don't let yourself be abused	200
Lesson 228	Don't abuse your staff with overtime	200
Lesson 229	Don't let your staff be abused	202

Lesson 230	Create training opportunities	202
Lesson 231	Your hiring decisions are your most important decisions	203
Lesson 232	Hire contractors to give you breathing room during recruiting	203
Lesson 233	Rarely accept rejects from other groups into testing	203
Lesson 234	Plan in terms of the tasks you need to do in your group and the skills needed to do them	204
Lesson 235	Staff the testing team with diverse backgrounds	204
Lesson 236	Hire opportunity candidates	205
Lesson 237	Hire by consensus	206
Lesson 238	Hire people who love their work	206
Lesson 239	Hire integrity	206
Lesson 240	During the interview, have the tester demonstrate the skills you're hiring him for	206
Lesson 241	During the interview, have the tester demonstrate skills he'll actually use on the job over informal aptitude tests	207
Lesson 242	When recruiting, ask for work samples	207
Lesson 243	Hire quickly after you make up your mind	208
Lesson 244	Put your hiring promises in writing and keep them	208

Chapter 10 Your Career in Software Testing

Lesson 245	Choose a career track and pursue it	209
Lesson 246	Testers' incomes can be higher than programmers' incomes	211
Lesson 247	Feel free to change your track and pursue something else	212
Lesson 248	Whatever path you take, pursue it actively	212
Lesson 249	Extend your career beyond software testing	213
Lesson 250	Extend your career beyond your company	213
Lesson 251	Conferences are for conferring	214
Lesson 252	Lots of other companies are as screwed up as yours	214
Lesson 253	If you don't like your company, look for a different job	215
Lesson 254	Be prepared in case you have to bet your job (and lose)	215
Lesson 255	Build and maintain a list of companies where you'd like to work	216
Lesson 256	Build a portfolio	216
Lesson 257	Use your resume as a sales tool	217
Lesson 258	Get an inside referral	218
Lesson 259	Research salary data	218
Lesson 260	If you're answering an advertisement, tailor your answer to the advertisement	218
Lesson 261	Take advantage of opportunities to interview	218
Lesson 262	Learn about companies when you apply for jobs with them	219
Lesson 263	Ask questions during job interviews	220
Lesson 264	Negotiate your position	221
Lesson 265	Be cautious about Human Resources	223
Lesson 266	Learn Perl	223
Lesson 267	Learn Java or C++	223
Lesson 268	Download demo copies of testing tools and try them out	224
Lesson 269	Improve your writing skills	224
Lesson 270	Improve your public speaking skills	224

Lesson 271	Think about getting certified	224
Lesson 272	If you can get a black belt in only two weeks, avoid fights	226
Lesson 273	A warning about the efforts to license software engineers	226

Chapter 11 Planning the Testing Strategy

Lesson 274	Three basic questions to ask about test strategy are “why bother?”, “who cares?”, and “how much?”	231
Lesson 275	There are many possible test strategies	232
Lesson 276	The real test plan is the set of ideas that guides your test process	233
Lesson 277	Design your test plan to fit your context	233
Lesson 278	Use the test plan to express choices about strategy, logistics, and work products	234
Lesson 279	Don’t let logistics and work products blind you to strategy	235
Lesson 280	How to lie with test cases	235
Lesson 281	Your test strategy is more than your tests	236
Lesson 282	Your test strategy explains your testing	236
Lesson 283	Apply diverse half-measures	237
Lesson 284	Cultivate the raw materials of powerful test strategies	238
Lesson 285	Your first strategy on a project is always wrong	238
Lesson 286	At every phase of the project, ask yourself “what can I test now and how can I test it?”	239
Lesson 287	Test to the maturity of the product	239
Lesson 288	Use test levels to simplify discussions of test complexity	241
Lesson 289	Test the gray box	242
Lesson 290	Beware of ancestor worship when reusing test materials	242
Lesson 291	Two testers testing the same thing are probably not duplicating efforts	243
Lesson 292	Design your test strategy in response to project factors as well as product risks	243
Lesson 293	Treat test cycles as the heartbeat of the test process	244

Imagine that you are holding a bottle of 50-year-old port. There is a way to drink port. It is not the only way, but most folks who have enjoyed port for many years have found some guidelines that help them maximize their port-drinking experience. Here are just a few:

Lesson 1: Don't drink straight from the bottle. If you have no glass, nor any other possible containers available, pour a tiny amount of port into the palm of your hand and sip from there. The port aroma should be smelled as you sip. Let the port swirl over your tongue. Don't gulp down the port.

Lesson 2: Don't drink the entire bottle. If you are drinking because you are thirsty, put down the port and drink a big glass of water. A small amount of port each time maximizes the enjoyment of the entire bottle.

Lesson 3: Don't pollute the port. If someone tells you that you ought to try a new cocktail made with orange juice, seawater, and port, politely refuse. With a big smile, say, "But I would enjoy a glass of port."

Lesson 4: Don't covet the port. Hoarding your port means that you will never have the pleasure of gentle conversation while sipping. Port is best shared with friends who also enjoy a glass. Remember, they have a bottle somewhere, too.

You are not holding a bottle of port in your hands. You are holding *Lessons Learned in Software Testing*, a very valuable book about software testing. This book has been ripening to perfection throughout the 50 work-years of the authors' experiences. While port is for your taste buds, this book is for your brain. I think that you will find any other differences to be marginal. I have

savored this book and offer you this advice to maximize your reading experience.

Lesson 1. Don't drink straight from the bottle. Bring your own vessel to the reading of this book. That is, bring all of your experiences in software development and testing. If you have never participated in a serious software effort, this book will be too heady for you. It will leave you confused and unable to operate heavy machinery for some time. If you are experienced, savor the book's contents in the context of your own projects.

Lesson 2. Don't drink the entire bottle. Don't read this book in one sitting. Read a lesson or two, close the book, and decide how you respond to the words of Messrs. Kaner, Bach, and Pettichord. As you will discover, they call their approach to testing "context-driven." Only you know the context of your own work. You must determine where there is a fit between a given lesson and your particular work.

Lesson 3. Don't pollute the port. Somebody out there is going to make a list of the titles of the 293 lessons contained in this book. Please don't let it be you. The heart of the book is the explanation attached to each of the lessons. Be warned as well that someone else will immediately try to ISO-ify or CMM-ummify the contents. I can see the article title now, "Getting to CMM Level 293 using the 293 Lessons of *Lessons Learned in Software Testing*." Argh! As the authors explain, "... we don't believe in 'best practices.' We believe some practices are more useful than others under some circumstances." Written by pros, those statements represent the quintessence of attaining the state of Master-crafter of Software Testing.

Lesson 4. Don't covet the port. If there ever is a book to be read with your colleagues, this is it. Buy a carton-load and hand a copy out to everyone who tests and to everyone who thinks that she or he manages testers. Read a few selected lessons at a time and get together to talk about them over coffee, lunch, even port! Read, ruminate, enjoy. Cheers!

Tim Lister
August 17, 2001
lister@acm.org
The Atlantic Systems Guild, Inc.
New York City

The *Software Engineering Body of Knowledge* (SWEBOK) is being proposed as a suitable foundation for government licensing, for the regulation of software engineers, and for the development of university curricula in software engineering. The SWEBOK document claims to be consensus-based. One would expect such a document to carry the accumulated knowledge and wisdom (the accumulated *lessons*) of the field.

Here is everything that SWEBOK has to say about exploratory testing:

Perhaps the most widely practiced technique remains ad hoc testing: tests are derived relying on the tester skill and intuition (“exploratory” testing), and on his/her experience with similar programs. While a more systematic approach is advised, ad hoc testing might be useful (but only if the tester is really expert!) to identify special tests, not easily “captured” by formalized techniques. Moreover it must be reminded that this technique may yield largely varying degrees of effectiveness. (SWEBOK 0.95, 2001, 5-9)

How does SWEBOK treat what it acknowledges is the field’s most widely practiced technique? Nothing about how to practice the technique well. Only a statement that exploration should be done only by *real experts*, that other approaches are advised, and the suggestion that other, formalized techniques will yield less varying degrees of effectiveness.

Hah!

We *don't* pretend to offer a consensus document that describes our field's alleged *body of knowledge*, but *we do have a lot more* to say about our field's most common practices. Rather than dismissing exploratory testing, this book shows you what testing looks like through the eyes of people who use exploratory methods (and many other methods) in a drive to achieve excellent testing under real-life conditions.

Welcome to *Lessons Learned in Software Testing*

This book is about software development as we've experienced it. We've had a combined total of 50 to 60 years of development experience (depending on how you count). We've been at this for a while. We've seen a lot of great work and a lot of not-so-great work.

This book is not about how software engineering might be in a tidier and more controlled world. We're writing about the world as we've worked in it.

In our world, software development teams often work under ambitious deadlines, discovering what needs to be done at the same time as they're discovering how to do it. Sometimes their approaches are more formal, sometimes less. It depends on a wide range of circumstances.

We follow the *context-driven approach* in software testing. We expect that a method that works wonderfully under some circumstances will not work under others. Rather than talk about best practices, we talk about practices that are well-suited to the context at hand. We discuss the context-driven approach at the end of the book, but in essence, context-driven testing looks at the "what" of testing (the techniques, tools, strategies, and so on) in terms of "who," "when," "where," "why," and "what if."

Our goal is to match our selection of practices to the circumstances at hand in a way that achieves excellent testing. We don't expect to achieve great testing by taking over the project, nor by stamping our feet and telling the project manager (or executives) how Real Professionals would run the project. We don't expect to achieve great testing by intimidating programmers, nor by being obsequious to them. We don't expect to do great testing by filling out thousands of little pieces of paper (or comparable electronic records), nor by wasting everyone else's time on unnecessary processes.

We don't have a political or a bureaucratic or a formal-methods recipe for great testing.

That recipe doesn't exist!

We think great testing involves skillful technical work (searching for defects) and accurate, persuasive communication.

The skilled search is always exploratory. There's an infinite amount of testing to do, and a tiny amount of time in which to do just a little bit of that infinity. Every test we do, every document we write, every meeting we attend takes time away from running other tests that could expose a key defect. Facing that constraint, we optimize our testing processes so that they take advantage of our ever-growing knowledge of the product, its market, its applications, and its weaknesses. What we learn today gets reflected in more powerful tests tomorrow.

Even if:

- the product is well specified, and
- the specification accurately reflects a requirements document, and
- the requirements document accurately represents the actual needs of the product's stakeholders

(have you *ever* been on a project where all these "even if's" were true?); we will still learn a lot about how to test the product while we are testing it. In particular, as we discover errors, we learn how this group of programmers can go wrong. The specifications tell us about how the program is supposed to work when it is coded correctly. They don't tell us what mistakes to anticipate, nor how to design tests to find them. At that task, our key task, we get better from the start to the end of the project, on every project.

No matter what it looks like from the outside, whenever we are testing with our brains engaged, our work is exploratory.

Who This Book Is For

This book is for anyone who tests software, anyone who manages testers, and anyone who has to deal with testers in their software development projects. That includes project managers and executives.

The primary reader that we have in mind, the "you" in this book, has been testing for a few years and might recently have been promoted to a supervisory role. We hope that you'll see a lot in this book that matches your experience, that you'll gain new insights from our lessons, that you'll find lessons that are useful for quoting to your manager, that you'll like a few of our statements so much that you'll photocopy them and stick them on the

outside of your cubicle, and maybe that you'll react to at least one statement so strongly that you'll stick a copy on the middle of your dart board. (We want to stimulate thought, not only agreement.)

Newcomers to testing (and those of you who are just applying for a testing job) won't have as many occasions to feel that you've already experienced what we're writing about. For you, this book might provide some early insights and warnings, giving you a good taste of what issues testers face.

Hint: If you are absolutely new to testing and you are looking for a book to study from in order to prepare for a job interview, this is the wrong book. If this is the only book available to you, pay your most careful attention to the chapters on "Testing Techniques" and "Your Career in Software Testing." If you have a choice of books, we suggest the first five chapters of *Testing Computer Software* (Kaner *et al.*, 1993).

Programmers, project managers, and executives who have to work with testers will find this book a useful source of ideas for setting your expectations of the testing group. We hope that it will help you evaluate and discuss your concerns with the testing group if you disagree with their policies or feel that they are spending their time unwisely.

What This Book Is About

Over the years, we've learned many useful practices and helpful ways of evaluating situations. Our conclusions are based on experience. We summarize much of our experience in this book, in a series of short, readable descriptions of a few hundred *lessons*.

We adopted several criteria for including a lesson in this book. They are:

- The lesson should be useful or should provide insight.
- The lesson should pass the 90-minutes-thinking-alone test. A lesson is not worth including in the book if almost anyone who thought about testing for 90 minutes of undistracted time could come up with it.
- The lesson must be based on our actual experience. At least one of us (preferably all three of us) had to have successfully applied the advice we give. At least two of us had to have been burned by attempting to follow a practice we criticize. (*Note:* Sometimes, we come to different conclusions, based on different experiences. Occasionally, you'll see that we choose to provide two viewpoints rather than one. Even if only one view is presented, you cannot assume that all three of us fully agree with it—in the event of disagreement, we are likely, among the three of us, to defer to the one or two who have the most extensive experience with a given situation.)

- Lessons should be tempered by the experiences of our colleagues. We've been collecting detailed experience reports at the Los Altos Workshops on Software Testing, the Software Test Managers Round Tables, the Workshops on Heuristic and Exploratory Techniques, the Austin Workshops on Test Automation, the Patterns of Software Testing Workshops, the Workshop on Model-Based Automated Testing, Systems Effectiveness Management Groups, at dozens of software testing conferences, and at less formal peer cross-trainings (such as the annual Consultants' Camps in Crested Butte).
- The lesson should be brief and to the point but easily understood.
- A lesson can be longer, but only to the extent needed to explain how to do something or to provide a useful tool. Long descriptions and detailed background information are for the textbooks.
- Lessons should be self-contained. You should be able to start reading anywhere in the book.
- The collection of lessons should give you a feel for how we do and think about testing.

What This Book Is Not

This book is not a comprehensive guide to software testing.

This book is not a collection of lessons that are always true. These are *our* lessons, based on *our* experiences. We believe these are broadly applicable, but some lessons that have been useful and important in our careers may not work for you. You have to use your own judgment. As a particular limiter on the generality of this work, we should note that we've worked more on the development of software for the mass market and on contract-specified software than we have on software developed for in-house use. Our experience with life-critical software and embedded software is limited.

This book is not a collection of best practices. In fact, we don't believe in "best practices." We believe some practices are more useful than others under some circumstances. We are concerned that many things sold as best practices are pushed (and applied) uncritically, to situations that aren't appropriate to them.

How to Use This Book

We structured this book to make it easy for you to skim it or flip through it, rather than trying to read it from start to finish. At some point (we hope), you'll find a nugget, an idea that is very appealing to you. We cannot

recommend that you apply this idea uncritically. (Our lessons are not best practices.) Instead, we urge you to evaluate the lesson in terms of its appropriateness to your situation.

Here are some questions that might help you perform that evaluation:

- Under what circumstances would an application of this lesson work in your company?
- Under what circumstances would the application not work?
- Has anyone you know tried anything like this lesson before? What happened? Why? What's different about your current project from that person's project? Should that difference matter?
- Who is most likely to benefit from the attempt to apply this lesson?
- Who is most likely to be disadvantaged as a result of an attempt to apply the lesson?
- What will you learn from attempting to apply this lesson?
- Trying anything new adds risk. Does it make sense to see how well this lesson would apply to your company by running a pilot study, a situation in which you try out the lesson without committing yourself to it, or under a situation of low risk? Is it feasible to conduct a pilot study in your company, on (or in the service of) your current project?
- How will you know whether your application of this lesson worked? If it did work, how will you know whether the success came more from the intrinsic value of the lesson, which will last as you keep applying it, than from your enthusiasm in trying out the idea?
- What are the best and worst things that could happen to you as a result of trying to apply this lesson?
- What are the best and worst things that could happen to another stakeholder, such as a user or another member of the development team?
- What if a key person in your company disagrees with a lesson that you want to follow? How will you overcome their objections and then sell it to them?

We Hope This Book Stimulates Conversations and Debates

This book draws sharp contrasts between our views and some others. We think that clarity will help fuel debates that the field should be having. We don't believe there is an accepted paradigm in software testing or in software engineering as a whole. That's why we're not among the folks pushing

government licensing and regulation of software engineers after standardization on a *body of knowledge*. In our experience, there are remarkably different, credible views as to the best ways to proceed.

We want to be very clear about this. We often critique the work of people we respect highly. In many cases, we refer to work done by good friends of ours. Don't mistake an attack on an idea with an attack on the proponent or clearest author of that idea.

We think the field will benefit by head-on comparisons and contrasts of those views. It's important for our field to have conversations about our methods. We advocate the further development of skilled practices within every one of the main approaches. Eventually, we will all learn the circumstances under which the different approaches are the best. Until then, let the thousand flowers bloom.

A Few Notes on Vocabulary

Here are some of the key words in this book, and how we use them:

We. The authors.

You. The reader.

A *fault* is an error, a mistake in the implementation or design of the program.

As we use the term, an *error* is a fault.

A *failure* is the misbehavior of the program, resulting from the program encountering a fault.

Failures occur under *conditions*. For example, a program will crash when it attempts to divide by zero. The fault in this case is in the code that allows division by zero. The failure is the crash. But you don't see a failure unless the critical variable in the division has the value zero. That variable, having a value of zero, is the critical condition that must be met for the failure to occur.

A *symptom* is like a failure but less serious. For example, if the program has a memory leak, it might start slowing down long before it fails with an out-of-memory error message. The slowdown is a symptom of an underlying problem (memory shortage) that is not directly visible.

The word *bug* is a catch-all term. It could refer to anything wrong with the software. Someone reporting a bug might describe a fault, a failure, or a limitation of the program that makes it less valuable to a stakeholder.

If we define *quality* as value to some person (Weinberg, 1998, 2.i.), a *bug report* is a statement that some person considers the product less valuable because of the thing being described as a bug.

The word *defect* carries a legal charge. It means Something Is Definitely Wrong With The Product. Some companies don't allow the words *defect* or *defective* to appear in their bug tracking systems or bug-related memos.

Some companies prefer to say *anomaly*, *problem*, or *issue* instead of *bug*.

After you report a bug, the programmers (or the Change Control Board) will fix it or decide not to fix it. They will mark the bug *resolved* (fixed, deferred, not reproducible, works as designed, and so on).

Black box testing. Testing the external behavior of the program, by feeding the program inputs and examining the outputs. In typical black box testing, the tester has no knowledge of the internals of the code and is instead more familiar with the explicit and implicit product requirements, such as the ways the program will be used, the types of data likely to be fed to the program, any regulatory issues associated with the problem the software is attempting to solve or help solve, and the hardware and software environments in which the software will run.

Behavioral testing. Testing the external behavior of the program, similar to black box testing, but using as much knowledge of the internals of the program as is available to the tester and relevant to the test.

Functional testing. Black box or behavioral testing.

White box or glass box testing. Testing with knowledge of the internals of the program.

Structural testing. White box testing that is focused on the internal structure of the program, such as the flow of control from one decision or action to the next.

Smoke testing or build verification testing. A standard suite of tests applied to a new build. The tests look for fundamental instability or key things missing or broken. If the build fails these tests, you do not test it any further. Instead, you keep testing the old build or wait for the next one.

Project manager. The person who is accountable for shipping the right product on time and within budget. Some companies split the work that we ascribe to the project manager between a program manager and a development manager.

MaxInt. The largest integer possible on the user's platform or the programmer's development language. A number larger than MaxInt cannot be stored as an integer.

Client. Someone whose interests it is your job to serve. This probably includes everyone on the project to some extent, as well as the ultimate users of the product.

ACKNOWLEDGMENTS

This book would not have been possible without the support and assistance of a great many people. We thank Lenore Bach, Jon Bach, Becky Fiedler, Leslie Smart, and Zach Pettichord for support, understanding, and assistance while three frantic men thought of nothing but their book. We thank Pat McGee for providing research assistance at a crucial time.

We benefited from detailed and thoughtful reviews of early drafts. We've added several of our reviewers' examples and descriptions of alternative perspectives to the book. We thank Ståle Amland, Rex Black, Jeffrey Bleiberg, Hans Buwalda, Ross Collard, Lisa Crispin, Chris DeNardis, Marge Farrell, Dorothy Graham, Erick Griffin, Rocky Grober, Sam Guckenheimer, George Hamblen, Elisabeth Hendrickson, Doug Hoffman, Kathy Iberle, Bob Johnson, Karen Johnson, Ginny Kaner, Barton Layne, Pat McGee, Fran McKain, Pat McQuaid, Brian Marick, Alan Myrvold, Hung Nguyen, Noel Nyman, Erik Petersen, Johanna Rothman, Jane Stepak, Melora Svoboda, Mary Romero Sweeney, Paul Szymkowiak, Andy Tinkham, Steve Tolman, and Tamar Yaron.

This book has benefited enormously from numerous discussions we've had at the Los Altos Workshops on Test Automation, the Workshops on Heuristic & Exploratory Techniques, the Software Test Managers' Roundtable, the Austin Workshops on Test Automation, the Patterns of Software Testing Workshops, and at many other workshops, conferences, classes, and worksites with so many people who have put their hearts into finding better ways to test software. We thank you all.

