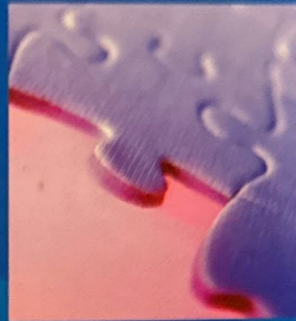


CEM KANER  
JAMES BACH  
BRET PETTICHORD

Lessons  
Learned



in ***SOFTWARE  
TESTING***

A Context-Driven Approach

# **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

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.)

when it occurred. If you think you can do that, you have either a very simple product or a very limited imagination.

You have to make choices about how to spend your time, knowing and accepting that you can't do everything.

## Lesson 10

### Beware of testing "completely."

Some testers who agree they can't know they've found all the bugs in a product still talk loosely about what it means to be finished testing. Saying "it will take me five days to test that" can be interpreted to mean that you think you will have *completely* tested that part of the product in five calendar days. And that might be taken to mean you will find every bug in five days. *Completeness* is more often implied than stated. Either way, it's a concept you must treat with great care. Think about what complete testing might mean:

- Completed the discovery of every bug in the product.
- Completely examined every aspect of the product.
- Completed the testing that you believe is useful and cost-effective to perform at this time.
- Completely fulfilled the stated objectives of the project to the best of your ability.
- Completed the agreed-upon tests.
- Completed everything humanly possible to test, under the circumstances.
- Completed everything you knew how to do.
- Completed your part of the testing, regardless of anyone else's part.
- Completed a broad, but not deep, pass of testing on the product.
- Completed one kind of testing on the product.
- Completed the time period assigned to testing.

If you take care to clarify what you mean and don't mean by "complete" or "finished" or "done," then you're probably safe. You're less likely to be blamed for not doing your job and better able to defend yourself if you are blamed. Be aware that the definition of "complete" is not the kind of thing that can be settled conclusively at the start of the project. You have to reconsider it as the test project evolves and as new test tasks crop up.

To cope with the general problem of miscommunication about completeness, share some of the details of your test process with your clients. Summarize the testing you do and why that testing is worthwhile, and tell your clients

investigates how people use causal theories to reason about systems. This explains how testing is more than simply looking at external behavior and checking it against simple expectations.

**Lesson  
19****Testing is in your head.**

The difference between excellent testing and mediocre testing is how you think: your test design choices, your ability to interpret what you observe, and your ability to tell a compelling story about it. The rest of testing is ordinary office work, for the most part. If you see two testers working, side-by-side, you can't necessarily tell whether one is testing better than the other. The visible part of their work looks the same, which has two implications:

- Many people think testing is easy, because they can easily copy the visible behavior of a good tester, and they have no other standard for good testing.
- If you want to be a good tester, learn to think like one, not look like one.

**Lesson  
20****Testing requires inference, not just comparison of output to expected results.**

There's a popular view that testers just execute test cases and compare what happens against expected results. This view makes testing seem like a straightforward comparison activity and ignores the fact that some clever person must *design* the tests and *determine* the expectations. Guess what? That test designer almost never has access to an authoritative guide to what should be tested, let alone what should be expected. And what guides are available are subject to interpretation. In real life, most test design is based on inferences or taken from experience that the tester infers is relevant. Moreover, these inferences change over time. To think like a tester is to be adept at the art of exploratory inference.

Exploratory inference may sound like a strange idea. It means one idea leading to another and another in ways you can't predict in advance. For a great introduction to exploratory inference, see *Proofs and Refutations: The Logic of Mathematical Discovery* (Lakatos 1976). The great thing about this book is how Lakatos shows that the process of mathematical and scientific reasoning is exploratory, not scripted. Even mathematicians reason actively

intentionally uneven. We expect that you'll learn more about the details of most techniques in other books and classes.

- Second, even though this is not primarily a how-to chapter on techniques, we couldn't bring ourselves to write a chapter on testing techniques without describing at least a few techniques in enough detail that you could actually use them. Hence the *Addendum*, which describes five techniques that we find useful, in ways that have worked well for our students in professional-level seminars and university courses on software testing.

Lesson 48

## Testing combines techniques that focus on testers, coverage, potential problems, activities, and evaluation.

Our primary goal in this chapter is to present a classification system for testing techniques. We call it the *Five-fold Testing System*. Any testing that you do can be described in terms of five dimensions:

- *Testers.* Who does the testing. For example, user testing is focused on testing by members of your target market, people who would normally use the product.
- *Coverage.* What gets tested. For example, in function testing, you test every function.
- *Potential problems.* Why you're testing (what risk you're testing for). For example, testing for extreme value errors.
- *Activities.* How you test. For example: exploratory testing.
- *Evaluation.* How to tell whether the test passed or failed. For example, comparison to a known good result.

We also describe a few techniques in detail in this chapter and present insights about the use of a few others, but our primary goal is to explain the classification system.

*All testing involves all five dimensions.* A testing technique focuses your attention on one or a few dimensions, leaving the others open to your judgment. You can combine a technique that is focused on one dimension with techniques focused on the other dimensions to achieve the result you want. You might call the result of such a combination a new technique (some people do), but we think the process of thinking is more useful than adding another name to the ever-expanding list of inconsistently defined techniques in use in our field. Our classification scheme can help you make those combinations consciously and thoughtfully.

determine its objectives before deciding how it will be scheduled and conducted.

**Bug bashes.** In-house testing using secretaries, programmers, marketers, and anyone who is available. A typical bug-bash lasts a half-day and is done when the software is close to being ready to release. (Note: we're listing this technique as an example, not endorsing it. Some companies have found it useful for various reasons; others have not.)

**Subject-matter expert testing.** Give the product to an expert on some issues addressed by the software and request feedback (bugs, criticisms, and compliments). The expert may or may not be someone you would expect to use the product—her value is her knowledge, not her representativeness of your market.

**Paired testing.** Two testers work together to find bugs. Typically, they share one computer and trade control of it while they test.

**Eat your own dogfood.** Your company uses and relies on prerelease versions of its own software, typically waiting until the software is reliable enough for real use before selling it.

Lesson  
50

## Coverage-based techniques focus on what gets tested.

---

You could class several of these techniques differently, as problem-focused, depending on what you have in mind when you use the technique. For example, feature integration testing is coverage-oriented if you use it to check that every function behaves well when used in combination with any other function. It's problem-oriented if you have a theory of error for functions interacting together and you want to track it down. (For example, it's problem oriented if your intent is to demonstrate errors in the ways that functions pass data to each other.)

We spend some extra space on domain testing in these definitions and at the end of the chapter because the domain-related techniques are so widely used and so important in the field. You should know them.

**Function testing.** Test every function, one by one. Test the function thoroughly, to the extent that you can say with confidence that the function works. White box function testing is usually called unit testing and concentrates on the functions as you see them in the code. Black box function testing focuses on commands and features, things the user can do

interest are 10 (smallest), 9 (largest integer that is too small), 50 (largest), and 51 (smallest integer that is too large).

**Best representative testing.** A best representative of an equivalence class is a value that is at least as likely as any other value in the class to expose an error in the software. In boundary testing, the boundary cases are almost always best representatives. But suppose that you cannot map an equivalence class onto a number line. For example, the printers that are Hewlett-Packard PCL-5 compatible are (or should be) an equivalence class because they should all work the same way. Now suppose that for a specific task, one of the printers is slightly more likely to have trouble than the others. This printer would be a best representative for that class. If it doesn't fail, we have some confidence that the other printers also wouldn't.

**Input field test catalogs or matrices.** For each type of input field, you can develop a fairly standard set of test cases and reuse it for similar fields in this product and later products. We give an example of this technique later in the chapter. (See the How to Create a Test Matrix for an Input Field.)

**Map and test all the ways to edit a field.** You can often change the value of a field in several ways. For example, you might be able to import data into the field, enter data directly into the field, have the program copy a calculated result into the field, have the program copy a recalculated result into the field, and so on. The field has constraints (restrictions on what values the field can take). Some constraints will be constant, while others will depend on the values of other fields. For example, if J and K are unsigned integers, they're constrained to the values of 0 through MaxInt. These are constant constraints. They depend on the programming language's definition of unsigned integers. However, suppose that N is also an unsigned integer, that  $N = J + K$ , and that  $N = 5$ . In this case,  $J = 5 - K$ , and J cannot possibly be bigger than 5 (the value of N). This is a variable constraint, whose range of allowable values depends on the value of N. To check that J is kept within its allowable range ( $5 - K$ ), you would try changing its value using each way that you can enter data into J.

**Logic testing.** Variables have relationships in the program. For example, the program might have a decision rule that says that if PERSON-AGE is greater than 50 and if SMOKER is YES, then OFFER-INSURANCE must be NO. The decision rule expresses a logical relationship. Logic testing attempts to check every logical relationship in the program. *Cause-effect graphing* is a technique for designing an extensive set of logic-based tests.

**State-based testing.** A program moves from state to state. In a given state, some inputs are valid, and others are ignored or rejected. In response to a valid input, the program under test does something that it can do and does

not attempt something that it cannot do. In state-based testing, you walk the program through a large set of state transitions (state changes) and check the results carefully, every time.

**Path testing.** A path includes all of the steps that you took or all of the statements that the program passed through in order to get to your current state. Path testing involves testing many paths through the program. You cannot test all the paths through a nontrivial program. Therefore, some testers do *subpath testing*, testing many partial paths. *Basis-path testing*, for example, involves testing most or all subpaths of a certain type (the basis paths), under the assumption that if you get all of these, few tests of longer paths would be likely to find bugs that these tests missed.

**Statement and branch coverage.** You achieve 100 percent statement coverage if your tests execute every statement (or line of code) in the program. You achieve 100 percent statement and branch coverage if you execute every statement and every branch from one statement to another. Designing your tests to achieve a high percentage of line and branch coverage is sometimes called "Coverage-based testing." (And after you achieve that, you can quit testing or quit designing additional tests). We call this *statement-and-branch coverage* to differentiate it from all of the other types of testing that focus on some other type of coverage. Configuration coverage is an excellent example of a technique that hits the same statements many times but with potentially very different results. There are many, many other examples (Kaner 1995a). Testing that is focused on achieving high statement-and-branch coverage numbers will characteristically miss many types of bugs, such as (but not only) bugs involving missing code, incorrect handling of boundary values, timing problems, problems of compatibility with hardware and software configurations, delayed-fuse bugs like wild pointers, memory leaks or stack corruption that eventually leads to stack overflow, usability problems, and other failures to meet customer requirements. This technique is much more valuable to identify incomplete testing (what code has not yet been tested), than as a minimum standard for the amount of testing needed. Indeed, it's dangerous to allow testers to stop merely because they achieved X percent coverage (Marick 1999).

**Configuration coverage.** If you have to test compatibility with 100 printers, and you have tested with 10, you have achieved 10 percent printer coverage. More generally, configuration coverage measures the percentage of configuration tests that you have run (and the program has passed), compared to the total number of configuration tests that you plan to run. *Why do we call this a test technique?* Ordinarily, we would just consider this a measure of how much of a certain type of testing we had achieved. However, some testers craft a special series of tests that will make high-