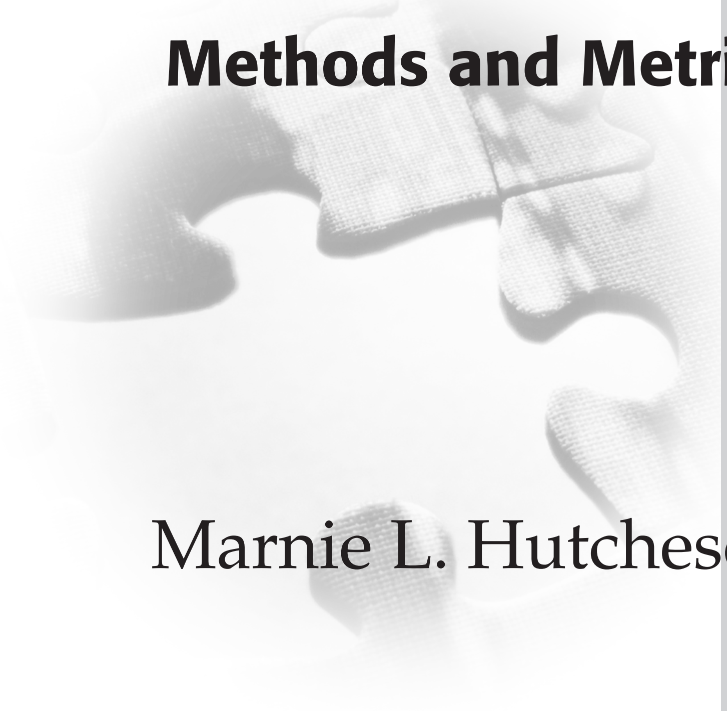
Software Testing Fundamentals

Methods and Metrics



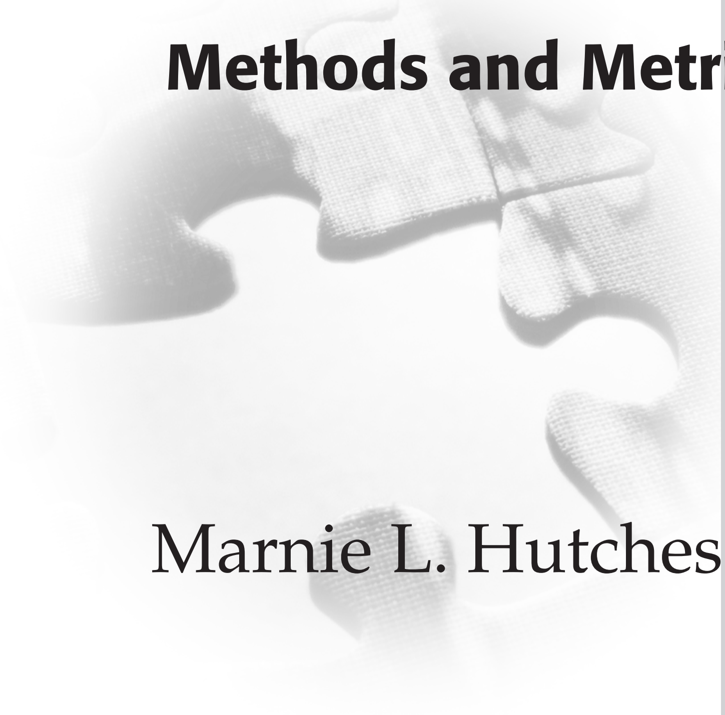
Marnie L. Hutcheson

WILEY

Wiley Publishing, Inc.

Software Testing Fundamentals

Methods and Metrics



Marnie L. Hutcheson

WILEY

Wiley Publishing, Inc.

Executive Publisher: Robert Ipsen
Senior Editor: Ben Ryan
Assistant Developmental Editor: Scott Amerman
Editorial Manager: Kathryn A. Malm
Assistant Managing Editor: Vincent Kunkemueller
Text Design & Composition: Wiley Composition Services

This book is printed on acid-free paper. ♾

Library of Congress Cataloging-in-Publication Data:

ISBN: 0-471-43020-X

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

CONTENTS

ACKNOWLEDGMENTS

When a subject is well understood, it can be explained in a few words, but the road to that understanding can be a long one, indeed. Every human being since the beginning of time has understood the effects of gravity—you trip, you fall down. Yet Sir Isaac Newton explained this phenomenon briefly and accurately only recently on the human time scale.

I have been working on developing the material in this book and circulating it back to testers and their management for many years. Most of the methods and techniques presented here are simple, but good answers don't have to be difficult. Many of these methods are about as old and patentable as oatmeal; others are new. Many of these methods have been discussed and debated for months and even years with colleagues.

The first four chapters of the original version of this book have been available online for four years. In that time the number of readers has risen steadily; presently about 350 visitors read these chapters each week. More than 2,500 software testers and managers in various industries reviewed this work and provided feedback on it. To all those who took part in those discussions, asked questions, picked nits, or came back and reported which steps they tried to implement and how it worked, I want to say again, "Thank you. And don't stop the feedback; that's how we improve our knowledge."

Thanks to Minna Beissinger, Joseph Mueller, Eric Mink, Beate Kim, Joan Rothman, L. Gary Nakashian, Joy Nemitz, Adrian Craig Wheeler, Lawrence Holland, John Chernievsky, Boris Beizer, Dorothy Graham, Roger Sherman, Greg Daich, and Tom Gilb.

I want to express my special thanks to my technical reviewer, researcher, and product developer, David Mayberry. This work would probably not exist except for his patience and continuing efforts. I also want to thank my editors at Wiley; Ben Ryan and Scott Amerman,

Vincent Kunkemueller, and all the others who were so patient and supportive as they added their talents to the making of this work. And finally, many thanks to my research assistants, and my artistic staff, Velvalee Boyd and Dane Boyd.

ABOUT THE AUTHOR

Marnie Hutcheson creates technical courseware for Microsoft Corporation and travels around the world training the trainers who teach these technologies to the world. She is an internationally published author and speaker in the areas of software development, testing and quality assurance, and systems administration.

She began her career in engineering at Prodigy Services Company in 1987 as the Lead Systems Integrator for Shopping, and later Banking and Financial Services. Over the years, she has become a leader in the development of the Web and has helped corporations like GTE and Microsoft develop and launch several major Internet technologies.

Prior to that, she was a student of the performing arts for over 25 years. She performed on stage, in classical ballet, musical theater, folk singing, and opera for 10 years in Denver, Montreal, Boston, and New York.

She also taught dance in institutions and colleges during those years. In the late 1970s, she was the dance instructor and assistant choreographer to U.S. Olympic skating coach, Don Laws, and Olympic choreographer Riki Harris. She worked with U.S. Olympians, elite skaters, and gymnasts in the United States and Canada.

INTRODUCTION

I live in a software development world where product development is not an orderly consistent march toward a tangible goal. "The Project Plan" usually consists of a laundry list of functions dropped off by somebody from marketing. Management embellishes "The Plan" with start and end dates that are of highly questionable origins and totally unreachable. The design and implementation of the product are clandestinely guarded by developers. The product routinely arrives in test virtually unannounced and several weeks late. The tester has not finished the test plan because no one is quite sure what the thing does. The only sure thing is the product must ship on time, next week.

That is software development—chaotic and harried. This book is dedicated to the proposition that this development system is primitive and enormously wasteful. This book presents several methods that provide better ways to perform the business of understanding, controlling, and delivering the right product to the market on time. These methods, taken singly or in groups, provide large cost savings and better-quality products for software developers.

I am a practitioner. I work where the rubber meets the road. I am often present when the user puts their hands on the product for the first time. I deal with real solutions to real problems. I also deal with the frustration of both the customers (who are losing money because the product is failing in some way) and the front-line support people. Front-line support is typically caught in the middle between development groups, who have other priorities, and the customer, who needs the system fixed "right now."

I work with the developer, whose job is to write good code. Developers do not have time to fill out all those forms quality assurance wants, or to compose an operations document that the test and support groups need. I work with the testers, who really don't know what's going on back there in the system. They keep breaking it, but they can't reproduce the problems for development. And I work with the document writers, who can't understand why the entire user interface changed just two weeks before the end of the test cycle.

My role is to prevent failures and enhance productivity through automation and process optimization. I work primarily on applications running in large networks. These systems are huge and contain a variety of components that need to be tested. Typically, there are object-oriented modules and graphical user interfaces (GUIs), and browser-based interfaces. These applications typically interact with databases, communications networks, specialized servers, and embedded code, driving specialized hardware—and all of them need to be tested. The methods in this book are distilled from experiences, both failures and successes, with projects that have touched all of these areas.

This is also a work about "how to solve problems," so it is rich with commentary on human factors. Systems are designed, written, integrated, tested, deployed, and supported by human beings, for human beings. We cannot ignore the fact that human factors play a major role in virtually all system failures.

What This Book Is About

This book is a software tester's guide to managing the software test effort. This is not a formula book of test techniques, though some powerful test techniques are presented. This book is about defensible test methods. It offers methods and metrics that improve the test effort, whether or not formal test techniques are used. It is about how to use metrics in the test effort. There is no incentive to take measurements if you don't know how to use the results to help your case, or if those results might be turned against you. This book shows how to use measurement to discover, to communicate those discoveries to others, and to make improvements.

Some time back I was presenting an overview of these methods at a conference. Part of the presentation was a case study. After these methods were applied, a test inventory was built, and the risk analysis was performed for the system, it was determined within this case study that the optimal test coverage given the time and resources allowed was 67 percent coverage of the entire test inventory.

During the question-and-answer session that followed my presentation, a very distinguished and tall fellow practitioner (he stands well over six feet) said, “Excuse me for mentioning this, but it strikes me that you are a very small person. I was wondering where you find the courage to tell your managing director that you only plan to test 67 percent of the system?”

My answer: “It is true that I am only 5'6'', but I am big on the truth. If management wants to give me enough time and resources to test every item on the inventory, I will be happy to do so. But if they want me to do with less than that, I am not going to soft sell the fact that they will get less than 100 percent test coverage. If there isn’t time and resources to test everything, then I want to be sure that the tests conducted are the *most important tests*.”

I am also going to tell management how good that selection of tests was, how many bugs the test effort found, how serious they were and how much it cost to find them, and if possible, how much was saved because we found and removed them. I will measure the performance of the test effort and be able to show at any time whether we are on schedule or not, if the error densities are too high, or if the bug-fix rate is too low. If we cannot stay on schedule, I can give management the high-quality information it needs to do what it does best, specifically, manage the situation.

Industry and Technology Trends: Why I Think It’s Time to Publish This Book

I began developing these methods in the late 1980s when I worked at Prodigy. They evolved to suit the needs of a fast-paced development environment feeding a large, complex real-time system. They were called the Most Important Tests method, or MITs. MITs quickly became the standard for testing methods at Prodigy, and I began writing and publishing case studies of MITs projects in 1990. I took MITs with me when I left Prodigy in 1993, and it continued to evolve as I tackled more and more testing projects in other industries. I spent most of the last 10 years helping businesses embrace and profit from integrating large systems and the Internet.

The (PowerPoint-based) syllabus that I developed to teach MITs since 1993 is based on the first seven chapters that I wrote for the original book, *Software Testing Methods and Metrics*. The emphasis then was on client/server testing, not the Internet, and that is reflected in the original chapters.

First offered in 1993, the course has been taught several times each year ever since. I put the original first four chapters on my Web site in 1997. The number of people reading these four chapters has increased steadily over the years. This year some 17,000 visitors have downloaded these chapters. The most popular is Chapter 2, "Fundamental Methods." Because of its popularity and the many e-mail discussions it has sparked, it has been expanded here into Chapter 3: "Approaches to Managing Software Testing," and Chapter 4: "The Most Important Tests (MITs) Method."

Changing Times

I spent most of 2001 working on Microsoft's .NET developer training materials, and in the process, I became very familiar with most of the facets of .NET. Bottom line is, the new .NET architecture, with its unified libraries, its "all languages are equal" attitude about development languages, and its enablement of copy-and-run applications and Web services brings us back to the way we did things in the early 1990s—those heady days I spent at Prodigy. The big and exciting difference is that Prodigy was small and proprietary; .NET will be global and public (as well as private).

It will be two years before we really begin to see the global effects of this latest shortening of the software development cycle. Literally, anyone can deploy and sell software as a Web service on global scale, without ever burning a CD, or writing a manual, or paying for an ad in a magazine. And at some point, that software will be tested.

The picture becomes even more interesting when you consider that we are now just beginning the next wave of Internet evolution; the mobile Internet. Just as the PC revolutionized the way we do business today, the smart phones and pocket PCs will allow more people than ever before to access dynamic applications on small screens, with tenuous data links. The methods in this book evolved in just such an environment, and were successful.

Software testing must show that it adds value, and that it is necessary for product success. Otherwise, market forces will encourage competitive shops to forego testing and give the product to the users as fast as they can write it and copy it to the server.

This book is about fundamentals, and fortunately, "fundamental" concepts, while sometimes out of style, evolve very slowly. The examples in the original Prodigy work were out of style in the client/server days; they are very much back in style in the .NET world. In many ways, revising this work to be current today is actually taking it back to its original state.

Scope of This Book and Who Will Find It Useful

This book is a field guide aimed squarely at testers and management involved with developing software systems and applications. It contains practical solutions, not theory. Theory and background are presented only to support the practical solutions.

This is a tester's survival guide because it helps testers supply answers that management understands and respects. Testers need to answer the question, "Why can't you have it tested by next week?" This work is also a manager's survival guide because managers have to explain why things are the way they are, how much it's going to cost, and why.

The methods presented here were developed in large networks. Often these networks are running a combination of Web-based and client/server-based applications, some on the public Internet and some running privately behind the firewall. These systems are generally written at least in part using object-oriented languages, and all are accessed using graphical user interfaces, be they dedicated clients or Web pages running in a browser. These test methods have been used to test a rich variety of other software systems as well, including telecommunications, business applications, embedded firmware, and game software.

The process described in this book is a *top-down approach* to testing. These methods can be used to test at the unit level, but they are more useful in integration, system, and end-to-end test efforts. These test methods are often used later in the project life cycle, in load testing, performance testing, and production system monitoring. Opportunities for automation or test reuse are noted as appropriate.

Last, this is not an all-or-none test guide. A process improvement can result from implementing parts of these methods, like adding a metric to test tracking, or prioritizing tests and keeping track of how long each one takes to run.

How This Book Is Organized

The chapters have been organized to parallel the process flow in most software development and test efforts. Successive chapters tend to build on what came in the chapters before; so jumping right into the middle of the book may not be a good idea. It is best to proceed sequentially.

Case studies, notes on automation, test techniques, usability issues, and human factors appear throughout the text. It is broken into three main blocks:

Chapters 1 to 5 concentrate on background and concepts.

Chapters 6 to 8 focus on the inventory and how to make it.

Chapters 9 to 14 cover tools and analysis techniques for test estimation, sizing, and planning.

The Standard for Definitions in This Book

The standard for all definitions given in this book is *Webster's New World Dictionary of the American Language* (College Edition, Prentice Hall). However, any good dictionary should be acceptable. When I refer to a definition from some other work, I will cite the work. I have tried to limit such references to works that are readily available to everyone today.

One of the major stumbling blocks I have encountered in educating people involved in developing and testing software is the lack of consensus on the meaning of basic terms. This is richly illustrated in the test survey discussed in Chapter 1 and presented in Appendix B of this book.

The biggest reason for this lack of consensus is that while there are plenty of standards published in this industry, they are not readily available or easy to understand. The second reason for the lack of consensus is simple disagreement. I have often heard the argument, "That's fine, but it doesn't apply here." It's usually true. Using the dictionary as the standard solves both of these problems. It is a starting point to which most people have access and can acknowledge. It is also necessary to go back to basics. In my research I am continually confronted with the fact that most people do not know the true meaning of words we use constantly, such as *test*, *verify*, *validate*, *quality*, *performance*, *effectiveness*, *efficiency*, *science*, *art*, and *engineering*.

To those who feel I am taking a step backward with this approach, it is a requirement of human development that we must learn to creep before we can crawl, to crawl before we can walk, and to walk before we can run. The level of mastery that can be achieved in any phase of development is directly dependent on the level of mastery achieved in the previous phase. I will make as few assumptions as possible about my readers' level of knowledge.

We software developers and testers came to this industry from many directions, many disciplines, and many points of view. Because of this, consensus is difficult. Nevertheless, I believe that our diversity gives us great strength. The interplay of so many ideas constantly sparks invention and innovation. The computer industry is probably home to the largest cooperative inventive undertaking in human history. It simply needs to be managed.

CHAPTER 1

The State of Software Testing Today

The director asked the tester, "So you tested it? It's ready to go to production?

The tester responded, "Yes, I tested it. It's ready to go."

The director asked, "Well, what did you test?"

The tester responded, "I tested *it*."

In this conversation, I was the tester. It was 1987 and I had just completed my first test assignment on a commercial software system.[1] I had spent six months working with and learning from some very good testers. They were very good at finding bugs, nailing them down, and getting development to fix them. But once you got beyond the bug statistics, the testers didn't seem to have much to go on except *it.* Happily, the director never asked what exactly *it* was.

The experience made me resolve to never again be caught with such a poor answer. I could not always be so lucky as to have management

[1] Commercial software is software that is commercially available and can be purchased by the public. This distinguishes it from safety-critical, proprietary, or military software.

that would accept *it* for an answer. All my training as a structural engineer had prepared me to give my management a much better answer than *it*.

Suppose the supervisor on a building project asked if I tested the steel superstructure on Floor 34 and needed to know if it was safe to build Floor 35. If I said "yes" and if the supervisor then asked, "What did you test?" I would have a whole checklist of answers on the clipboard in my hand. I would have a list with every bolt connection, the patterns of those connections, the specified torque wrench loading used to test the bolts, and the results from every bolt I had touched. I would know exactly which bolts I had touched because each would be marked with fluorescent paint, both on my chart and on the steel.

Why should software testing be any different? I could certainly give my management a better answer than *it*. Many of these better answers were around when the pyramids were being built. When I am asked those questions today, my answer sounds something like this:

> *As per our agreement, we have tested 67 percent of the test inventory. The tests we ran represent the most important tests in the inventory as determined by our joint risk analysis. The bug find rates and the severity composition of the bugs we found were within the expected range. Our bug fix rate is 85 percent.*
>
> *It has been three weeks since we found a Severity 1 issue. There are currently no known Severity 1 issues open. Fixes for the last Severity 2 issues were regression-tested and approved a week ago. The testers have conducted some additional testing in a couple of the newer modules. Overall, the system seems to be stable.*
>
> *The load testing has been concluded. The system failed at 90 percent of the design load. The system engineers believe they understand the problem, but they say they will need three months to implement the fix. Projections say the peak load should only be at 75 percent by then. If the actual loading goes above 90 percent, the system will fail.*
>
> *Our recommendation is to ship on schedule, with the understanding that we have an exposure if the system utilization exceeds the projections before we have a chance to install the previously noted fix.*

The thing that I find most amazing is that answers like these are not widely used in the industry today. I regularly hear testers and developers using *it* metrics.

Throughout the 1990s I gave out a survey every time I taught a testing course. Probably 60 percent of the students taking these courses were new to testing, with less than one year as a tester. About 20 percent had from one to five years' experience, and the remainder were expert testers. The survey asked the student to define common testing terms like *test*, and it asked them to identify the methods and metrics that they regularly used as testers. The complete results of these surveys are presented in the appendix of this book. I will mention some of the highlights here.

- The only type of metrics used regularly have to do with counting bugs and ranking them by severity. Only a small percentage of respondents measure the bug find rate or the bug fix rate. No other metrics are widely used in development or testing, even among the best-educated and seemingly most competent testers. It can also be inferred from these results that the companies for which these testers work do not have a tradition of measuring their software development or test processes.
- Few respondents reported using formal methods such as *inspection* or *structured analysis*, meaning some documented structured or systematic method of analyzing the test needs of a system. The most commonly cited reason for attending the seminar was to learn some software testing methods.
- The majority of testers taking the survey (76 percent) had had some experience with automated test tools. Today an even greater percent of testers report that they have used automated test tools, but test automation is also voted as the most difficult test technique to implement and maintain in the test effort.
- The respondents who are not actively testing provided the most accurate definitions of the testing terms. The people performing the testing supplied the poorest definitions of the testing tasks that they are performing most frequently.

Overall, I believe that the quality of the tester's level of awareness is improving, and, certainly, software testing practices have improved in the commercial software development sector. Today there are more publications, tools, and discussion groups available than there were 10 years ago. There are certainly more shops attempting to use formal methods and testing tools. But the survey results haven't changed much over the years.

How did we get to this mind-set? How did these limitations in perceptions—and, unfortunately, all too often in practice—come about? To understand that, we need to examine the evolution of software development and testing during the last two decades.

A Quick Look at How We Got Where We Are

Most of the formal methods and metrics around today had their start back in the 1970s and 1980s when industry began to use computers. Computer professionals of that time were scientists, usually mathematicians and electrical engineers. Their ideas about how to conduct business were based on older, established industries like manufacturing; civil projects like power plants; and military interests like avionics and ballistics.

The 1980s: The Big Blue and Big Iron Ruled

By the 1980s, computers were widely used in industries that required lots of computation and data processing. Software compilers were empowering a new generation of programmers to write machine-specific programs.

In the 1980s computers were mainframes: big iron. Large corporations like IBM and Honeywell ruled the day. These computers were expensive and long-lived. We expected software to last for five years, and we expected hardware to last even longer—that is, how long it takes to depreciate the investment. As a result, buying decisions were not made lightly. The investments involved were large ones, and commitments were for the long term, so decisions were made only after careful consideration and multiyear projections.

Fact: Computers in the 1980s: expensive, long-term commitment, lots of technical knowledge required

Normally a vendor during the 1980s would sell hardware, software, support, education, and consulting. A partnership-style relationship existed between the customer and the vendor. Once a vendor was selected, the company was pretty much stuck with that company until the hardware and software were depreciated; a process that could take 10 or more years.