SEI SERIES IN SOFTWARE ENGINEERING

Common System and Software Testing Pitfalls

How to Prevent and Mitigate Them

Descriptions, Symptoms, Consequences, Causes, and Recommendations

Donald G. Firesmith

Foreword by Capers Jones

Praise for *Common System and Software Testing Pitfalls*

"Firesmith's collection of actionable practices for real-world, non-trivial testing and the processes in which they're applied is comprehensive and uniquely valuable. Nothing published about software testing in recent years provides anything like it."

—**Robert V. Binder**, robertvbinder.com

"Don's compilation of real-world testing problems, symptoms, and solutions is the most comprehensive available. You can use it early in your project to prevent these problems. Or you can use it at the end, as a ready list of costly lessons you could have avoided, had you used it early on. I'm afraid this book's publication will undermine a lot of excuses for repeating these mistakes."

—**Vince Alcalde**, National Australia Bank

"Excellent, Excellent, Excellent! This book should be mandatory reading for anyone involved in product development. Donald's book addresses the pitfalls that need to be understood and allowed for in all product development verification and validation planning. While the focus of the book is on software projects, most of the pitfalls are equally applicable to any size project that involves both hardware and software."

—**Louis S. Wheatcraft**, Requirements Experts Inc.

"The potential impact of this book cannot be overstressed. Software systems that are not adequately tested do not adequately evolve. I highly recommend this book as a must-read for people directly involved in the development and management of software-intensive systems."

—**Dr. Kenneth E. Nidiffer**, Director of Strategic Plans for Government Programs, Software Engineering Institute, Carnegie Mellon University

"*Common System and Software Testing Pitfalls* identifies realistic testing pitfalls. More importantly, it also identifies solutions for avoiding them on your next project. Every manager should read this book and follow the recommendations."

—**Barry Stanly**, Enterprise Technology Alliance

"Whether you are a novice tester or a seasoned professional, you will find this book to be a valuable resource. The information on how to identify and prevent problem areas is clear, concise and, most importantly, actionable."

—**Allison Yeager**, Blackbaud

"First of all, this is great material! It contains probably all of the testing problems I have faced in my career and some that I wasn't aware of.... Thank you for the opportunity to read this superb material!"

—**Alexandru Cosma**, Frequentis

"As a tester, I consider *Common System and Software Testing Pitfalls* by Donald Firesmith to be a must-read book for all testers and QA engineers."

—**Thanh Huynh**, LogiGear

"Your book provides very good insight and knowledge. After working in IT for over thirty years, and focusing on software testing the past thirteen years, I still learned more tips and best practices in software testing."

—**Tom Zalewski**, Texas State Government

"This book is essential for the people in the cyber security business ... I can see it becoming a classic. Don has done a great job."

—**Michael Hom**, Compass360 Consulting

"Awesome work. Very mature."

—**Alejandro Salado**, **Kaiser**, Threde Gmbh

"All in all, a great document."

—**Peter Bolin**, Revolution IT Pty Ltd.

Common System and Software Testing Pitfalls

The SEI Series in Software Engineering

Software Engineering Institute of Carnegie Mellon University and Addison-Wesley

Software Engineering Institute | Carnegie Mellon

Visit **informit.com/sei** for a complete list of available publications.

The SEI Series in Software Engineering is a collaborative undertaking of the Carnegie Mellon Software Engineering Institute (SEI) and Addison-Wesley to develop and publish books on software engineering and related topics. The common goal of the SEI and Addison-Wesley is to provide the most current information on these topics in a form that is easily usable by practitioners and students.

Titles in the series describe frameworks, tools, methods, and technologies designed to help organizations, teams, and individuals improve their technical or management capabilities. Some books describe processes and practices for developing higher-quality software, acquiring programs for complex systems, or delivering services more effectively. Other books focus on software and system architecture and product-line development. Still others, from the SEI's CERT Program, describe technologies and practices needed to manage software and network security risk. These and all titles in the series address critical problems in software engineering for which practical solutions are available.

Make sure to connect with us!
informit.com/socialconnect

Addison Wesley | informIT | Safari Books Online

ALWAYS LEARNING PEARSON

Common System and Software Testing Pitfalls

How to Prevent and Mitigate Them

Descriptions, Symptoms, Consequences, Causes, and Recommendations

Donald G. Firesmith

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Software Engineering Institute | Carnegie Mellon

The SEI Series in Software Engineering

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact international@pearsoned.com.

Visit us on the Web: informit.com/aw

Library of Congress Cataloging-in-Publication Data

Firesmith, Donald G., 1952–
Common system and software testing pitfalls : how to prevent and mitigate them : descriptions, symptoms, consequences, causes, and recommendations / Donald G. Firesmith.
pages cm.
Includes bibliographical references and index.
ISBN 978-0-13-374855-0 (pbk. : alk. paper)
1. Computer software—Testing—Automation. 2. Software failures—Prevention. I. Title.
QA76.76.T48F58 2013
005.3028'7—dc23 2013039235

ISBN-13: 978-0-13-374855-0
ISBN-10: 0-13-374855-3

Text printed in the United States on recycled paper at R.R. Donnelley in Crawfordsville, Indiana.
First printing, December 2013

CONTENTS

This page intentionally left blank

FOREWORD

As a general rule, about 50 cents out of every dollar spent on software projects goes toward finding and fixing bugs. About 40 cents out of that 50 is spent on various kinds of testing, of which there are more than twenty types in total.

Software testing is a curious part of software engineering. Given that it is the key cost driver for software projects, and that testing costs go up with application size, it is alarming that testing is seldom done well. And yet, the topic is covered by some of the best software engineering books by some of the best authors. A quick search of the web or online bookstores will turn up dozens of books about software testing, and many of these are quite good.

There seem to be some social reasons for why testing is not as sophisticated as it should be. One of these is that not every test team member actually reads any of the books on testing. Another is that tests by certified test personnel who do know how to test well are still outnumbered by amateur testers or developers who may lack training and who may not have read the testing books either. A third reason, addressed by this book, is that many of the older testing books cover only part of the problem of achieving good testing results.

Many testing books are, as might be expected, "how-to-do-it" books that step through a sequence of test planning, test-case design, test-case construction, test execution, defect identification, defect repair, and repair integration. These are all teachable skills, and they should be better known than they are.

Don Firesmith's new book on testing approaches testing from a different perspective. Rather than being another "how-to-do-it" book, this book examines testing from the opposite direction. It explains the errors and failures of testing that he has observed over his long career in software and his work with the Software Engineering Institute (SEI).

This reverse view makes Don's book a natural complement to "how-to-do-it" books. I think that this is the first book on testing that emphasizes what goes wrong and how these problems might be avoided.

In other fields, combining how to do something with avoiding common problems is a standard part of the instruction sequence. For example, when learning to play golf, modern golf schools have video cameras that film golf students during lessons. The golf instructors go through the videos with each student and show them exactly what they did well and what they did poorly. This is an effective training technique. It is actually much harder to stop doing the wrong things than it is to learn to do the right things.

Other fields, such as professional football, also divide training into what has to be done right and what happens when the basics are not done right. This is why coaches and players review films after every game.

Until this new book on software testing, the literature only emphasized doing things well; it was a bit short on what happens if things are not done well.

In this book, a "pitfall" is any action or decision that might lower the effectiveness of testing. Don identifies ninety-two of these pitfalls, which is certainly the largest number I've encountered to date.

This book somewhat resembles a classic medical textbook that defines various illnesses and then discusses the available therapies for them. Each of the ninety-two pitfalls is defined and discussed using a standard format and identical points, which makes the book very easy to follow.

The pitfall descriptions use common headings, such as:

- Descriptions
- Potential Applicability
- Characteristic Symptoms
- Potential Negative Consequences
- Potential Causes
- Recommendations
- Related Pitfalls

This format is similar to a book I wrote some years ago, entitled *Assessment and Control of Software Risks,* published by Prentice Hall. My book used an actual medical text, *Control of Communicable Diseases in Man,* published by the U.S. Public Health Service, as the pattern.

Having worked as a programmer and systems analyst for the U.S. Public Health Service, I was in close contact with medical diagnostic methods, and they seemed to be appropriate for diagnosing software problems.

Some of my topics are very similar to Don's, such as:

- Definition
- Severity
- Susceptibility and Resistance
- Methods of Prevention
- Methods of Control

Applying some of the diagnostic patterns from medical practice to software engineering problems is a useful way to understand serious and common conditions, and Don has taken this idea to a new level for software engineering, and especially so for testing.

Testing is the main form of defect removal for software applications, but it is not the only form. A synergistic combination of defect prevention; pretest removal, such as inspections and static analysis; and formal testing by trained and certified test personnel can approach or exceed 99% in cumulative defect-removal efficiency levels. Even better, these good results come with shorter schedules and lower costs since the main source of software project delay is excessive defects during testing, which stretches out test schedules to several times their planned durations.

I recommend *Common System and Software Testing Pitfalls* for project managers, test personnel, quality-assurance personnel, and software engineers at all levels. All of us in software should know the common problems that we face during testing and how these problems might be avoided or minimized.

Don's book is a very good addition both to the testing literature and to the literature on quality assurance and software engineering. It is likely to become a standard for test training as well as a good reference for professional testers and developers. It would be a good teaching aid for young software engineers and a good handbook for all of us

I would also recommend this book as background material for negotiating outsourced software contracts. I often work as an expert witness in litigation for software with very poor quality, and this book might well reduce or eliminate these lawsuits if it were consulted before contract negotiations.

—Capers Jones, VP and CTO
Namcook Analytics LLC

This page intentionally left blank

PREFACE

There are numerous good books on systems and software testing, and most testers probably already have several on their shelves. There seems to be no scarcity of how-to books on testing, and they are full of excellent advice on how to test software-reliant systems. They cover test planning, the many different types of tests, how testing fits in to the development cycle, test-case design—including test-case-selection and test-completion criteria—test tools and environments, and many other interesting and useful topics.

Yet we spend a huge amount of time and effort testing, and in spite of this, we deliver systems with dozens if not hundreds of residual defects. In addition to being a tester, I have also taken part in numerous internal as well as independent technical assessments (ITAs) of system and software development projects, including their testing organizations and programs. And in every single case, regardless of whether I was a member of the test team or the assessment team, I have always observed several significant testing problems. More specifically, I have observed testers and other developers falling into the same pitfalls over and over again. Clearly, how-to books—while highly useful—are not sufficient to make testing either efficient or effective.

The frustration I experienced by enduring and observing these commonly occurring testing pitfalls led to this book. If the many how-to books are insufficient by themselves, then clearly it is time to try something different: a *how-not-to* book.

You can think of this book as a catalog and repository of testing antipatterns: the pitfalls to avoid, how to mitigate their negative consequences if you can't avoid them, and how to escape from them once you've fallen in. Like a naturalist's field guide to wild animals, let this be your guidebook to the dangerous world of testing mistakes and its denizens—the many creative ways people have discovered to botch testing.

Scope

The scope of this book is *testing*, which is only one of several methods commonly used to validate that a system meets its stakeholders' needs and to verify that the system conforms to its specified requirements. Although other such methods (for example, inspections, demonstrations, reviews, analysis, simulation, reuse, and certification) exist and could be documented in a similar manner, they are beyond the scope of this book.

The scope of this book is also the testing of *software-reliant systems*, which often are heterogeneous aggregations of subsystems, hardware, software, data, facilities, material, and personnel. This includes the testing of pure software applications and their components. For simplicity's sake, I will use the term *system* to mean heterogeneous systems, software applications, and their architectural, design, and implementation components.

The pitfalls in this book primarily apply to large and medium-sized projects producing important systems and software applications that require at least a quasi-rigorous testing program and process. The pitfalls do not necessarily apply to very small and simple projects producing relatively trivial systems or software programs that (1) are neither business-, mission-, safety-, nor security-critical; (2) will be used only in-house with close collaboration between stakeholders and developers; (3) will be used once and not maintained; or (4) are prototypes that will not be placed into operation. Such systems often can be adequately tested in a highly informal and ad hoc manner. Some of the pitfalls apply only or primarily to testing systems having significant hardware, and these pitfalls therefore do not (primarily) apply to testing software-only applications.

Intended Audience

This book is written primarily for testers and their technical managers. It is intended to help you recognize and avoid potential testing-related pitfalls.

This book is also written for all stakeholders in system development and sustainment who need a better understanding of what can go wrong with testing, both when preparing for testing and during the actual testing. This includes customer and user representatives, project managers and technical leaders, requirements engineers, architects and designers, implementers, maintainers, and specialty engineers (such as configuration managers, quality engineers, reliability engineers, and human factors engineers).

Finally, this book is written for testing subject-matter experts, whether academics or consultants, who need a more organized and comprehensive understanding of what can go wrong with testing.

How to Use This Book and Its Contents

The primary goal of this book is to provide the information you need to

- Avoid falling into any of the commonly occurring testing pitfalls
- Recognize when you have already fallen into one or more testing pitfalls
- Escape from these pitfalls while minimizing the resulting negative consequences

This book provides detailed information on the commonly occurring testing pitfalls, and it can be used

- To improve understanding of and communication about commonly occurring testing pitfalls
- As training material for testers and the stakeholders of testing
- As checklists[1][1] when
  - Developing and reviewing an organizational or project testing process or strategy
  - Developing and reviewing test planning documentation, such as:
    - Test and Evaluation Master Plans (TEMPs), System Test Plans (STPs), or Test Strategy Documents (TSDs)
    - The testing sections of planning documents such as the System Engineering Management Plan (SEMP) and the System Development Plan (SDP)
    - Test planning presentations (for example, for training and status reporting)
    - Testing wikis, SharePoint sites, and Application Lifecycle Management (ALM) tool repositories
  - Evaluating the testing-related parts of contractor proposals
  - Evaluating test planning documentation, test descriptions, and test results (quality control)
  - Evaluating the actual as-performed testing process (quality assurance)[2]
  - Identifying testing risks and appropriate risk-mitigation approaches
- To categorize testing pitfalls for metrics collection, analysis, and reporting

1. Notes are identified by a bracketed number ([#]), and are located in Appendix C, Notes.

- To help identify testing areas potentially needing improvement, both during a project and at its conclusion, such as during project postmortems

Organization of This Book

This book is organized as follows:

- **Preface**

  This preface begins with a brief introduction to the book, followed by a description of the book's scope and its intended audience. Next, it offers brief advice on how best to use the information provided here. Finally, I acknowledge the book's many technical reviewers, without whom it would not be half as good.
- **Chapter 1: Foundational Concepts**

  The first chapter defines the most important concepts in this book: testing, defects, and testing pitfalls. It presents the system-engineering V Models that explain how different types of testing are associated with the project's work products. It addresses why testing is so important as well as explains why it has some significant limitations. Finally, it explains how the testing pitfalls are categorized and documented to make them easier to locate and to understand.
- **Chapter 2: Brief Overviews of the Testing Pitfalls**

  The second chapter identifies and summarizes ninety-two commonly occurring testing pitfalls. The purpose of Chapter 2 is to provide a *very* brief, high-level overview of each pitfall, making it easy for readers to search for and identify pitfalls relevant or specific to their situations.
- **Chapter 3: Detailed Descriptions of the Testing Pitfalls**

  The third chapter provides detailed descriptions of each of the commonly occurring testing pitfalls. Specifically, it documents each pitfall as follows: its name, description, applicability, characteristic symptoms, potential negative consequences, potential causes, and associated recommendations for avoiding the pitfall or limiting its consequences. Chapter 3 is intended to be used primarily as a handy reference once relevant pitfalls are identified via either the Contents section or Chapter 2. Thus, I suggest that you read this chapter as you would read the patterns in a patterns book: once through rapidly to get a basic understand of the pitfalls, then examine the detailed specifications of individual pitfalls on an as-needed basis.
- **Chapter 4: Conclusion**

  The fourth and final chapter provides a holistic summary of the pitfalls before concluding with a brief look at potential future research that might make this categorization of testing pitfalls even more useful.

- **Appendixes**
  The appendixes start with a glossary of terms and a list of acronyms. In order to keep the descriptions of the individual pitfalls reasonably short—especially for the experienced tester, who will recognize the majority of these pitfalls—Appendix C provides extensive notes for those who might desire a little extra information. The notes are identified by bracketed numbers [#] throughout the text. The book's relatively short list of references comes next. The final appendix is a checklist that can be used when planning testing and assessing testing programs and organizations.

Acknowledgments

I am grateful for the interest of the more than 350 testers, testing subject-matter experts (SMEs), and academics from forty-six countries who volunteered to review various draft versions of this book. I am extremely grateful to the following individuals who provided excellent review comments and recommendations, often several times, while reviewing different drafts of the manuscript:

Dimpy Adhikary, Amagi Media Labs, India
Vince Alcalde, Independent Consultant, Australia
Stephanie Benedetti, AIPSO, US
Laxmi Bhat, Minerva Networks, US
Robert V. Binder, System Verification Associates, US
Peter Bolin, Revolution IT Pty Ltd, Australia
Michael Bolton, DevelopSense, Canada
Paul Carvalho, Software Testing and Quality Services, Canada
Alexandru Cosma, Frequentis, Romania
John Dannenberg, Compuware Corporation, US
Jorge Alberto De Flon, Servicio de Administración Tributara (SAT), Mexico
George Despotou, University of York, UK
Lee Eldridge, Independent Consultant, Australia
Eliazar Elisha, University of Liverpool, UK
Robin Goldsmith, Go Pro Management Inc., US
Jon L. Gross, Software Engineering Institute, US
Paolo Guolo, Private Consultant, Italy
Kobi Halperin, Ceragon Networks, Israel
Sam Harbaugh, Integrated Software Inc., US
John Hawrylak, Software Engineering Institute, US
M. E. Hom, Compass360 Consulting, US
Thanh Cong Huynh, LogiGear, Vietnam
Capers Jones, Namcook Analytics, US

Ronald Kohl, Independent Consultant, US
Wido Kunde, Baker Hughes, Germany
Seth Landsman, The MITRE Corporation, US
Philippe Lebacq, Toyota Motors Europe, Belgium
Stephen Masters, Software Engineering Institute, US
Ken Nidiffer, Software Engineering Institute, US
Anne Nieberding, Independent Consultant, US
William Novak, Software Engineering Institute, US
Mahesh Palan, Calypso Technology, US
Dan Pautler, Elekta, US
David C. Peterson, Protengent, US
Mark Powel, Attwater Consulting, US
James Redpath, Sogeti, US
Sudip Saha, Navigators Software, India
Alejandro Salado, Kayser—Threde GmbH, Germany
David Schultz, NASA, US
Matt Sheranko, Knowledge Code, US
Oleg Spozito, Independent Consultant, Canada
Barry Stanly, Independent Consultant, US
Amit Wertheimer, EMC Corporation, RSA, Israel
Lou Wheatcraft, Requirements Experts, US
Kerry Wilkerson, Private Consultant, US
Allison Yeager, Blackbaud, US
Thomas Zalewski, Texas State Government, US

Although the vast majority of the comments and recommendations from each of the reviewers made it into this book in one form or another, that does not mean that every reviewer would necessarily agree with everything in it. Further, I naturally take responsibility for any errors that slipped past my diligent reviewers and made it into the final book.

I would like to thank John Foreman, an SEI fellow and member of the management team, who provided the funding and time I needed to finalize the manuscript for publication.

Finally, I would like to thank the acquisition and production teams at Addison-Wesley for their strong support in publishing this book. Deserving special mention are Bernard Goodwin, my acquisitions editor, and Vicki Rowland, who copyedited the manuscript and created a very high level of consistency of both content and format. Both worked closely with me through several iterations of the book's cover and contents, thereby making this the most enjoyable of my books to bring to completion.

ABOUT THE AUTHOR

Donald G. Firesmith, a senior member of the technical staff in the Software Solutions Division at the Software Engineering Institute (SEI), helps the US Department of Defense and other governmental agencies acquire large, complex, software-reliant systems. He is an internationally recognized subject matter expert who has published seven software and systems engineering books in the areas of requirements engineering, architecture engineering, situational method engineering, testing, and object-oriented development. With thirty-five years of industry experience, he has also published dozens of technical articles, spoken at numerous international conferences, and has been the program chair or on the program committee of several software conferences. He has taught several hundred courses in commercial and governmental settings (both civilian and military, as well as numerous tutorials at international conferences. Copies of his numerous papers and presentations are downloadable from his personal website: http://donald.firesmith.net.

This page intentionally left blank

CHAPTER 1

OVERVIEW

1.1 What Is Testing?

Testing is the activity of executing a system, subsystem, or component under specific preconditions (for example, pretest mode, states, stored data, and external conditions) with specific inputs so that its actual behavior (outputs and postconditions) can be compared with its required or expected behavior.

Testing differs from other verification and validation methods (for example, analysis, demonstration, and inspection) in that it is a dynamic, as opposed to a static, analysis method that involves the actual execution of the thing being tested.

Testing has the following goals:

- Primary goal:
  - Enable the system under test (SUT) to be improved by:
    - "Breaking" it (that is, by causing faults and failures)
    - Exposing its defects so that they can be fixed
- Secondary goals:
  - Provide adequate confidence based on sufficient objective evidence regarding the SUT's:
    - Quality

      A system's quality is not just its lack of defects or its correctness (in terms of meeting its requirements). A system must also have the necessary levels of relevant quality characteristics and attributes; for example, availability, capacity, extensibility, maintainability, performance, portability, reliability, robustness, safety, security, and usability.
    - Fitness for purpose
    - Readiness for shipping, deployment, or being placed into operation

1.2 Testing and the V Models

Figure 1.1 illustrates a common way of modeling system engineering: the traditional V Model of system engineering activities.[1] On the left side of the V are the analysis activities that decompose the users' problem into small, manageable pieces. Similarly, the right side of the V shows the synthesis activities that aggregate (and test) these pieces into the system that solves the users' problem.

While useful, the traditional V model does not really represent system engineering from the tester's viewpoint. The next three figures show three increasingly detailed V models that better capture the testing-specific aspects of system engineering.

Figure 1.2 illustrates a V model oriented around work products rather than activities. Specifically, these are the major executable work products because testing involves the execution of work products. In this case, the left side of the V illustrates the analysis of ever more detailed executable models, whereas the right side of the V illustrates the corresponding incremental and iterative synthesis of the actual system. This V model shows the executable things that are tested rather than the general system engineering activities that generate them.

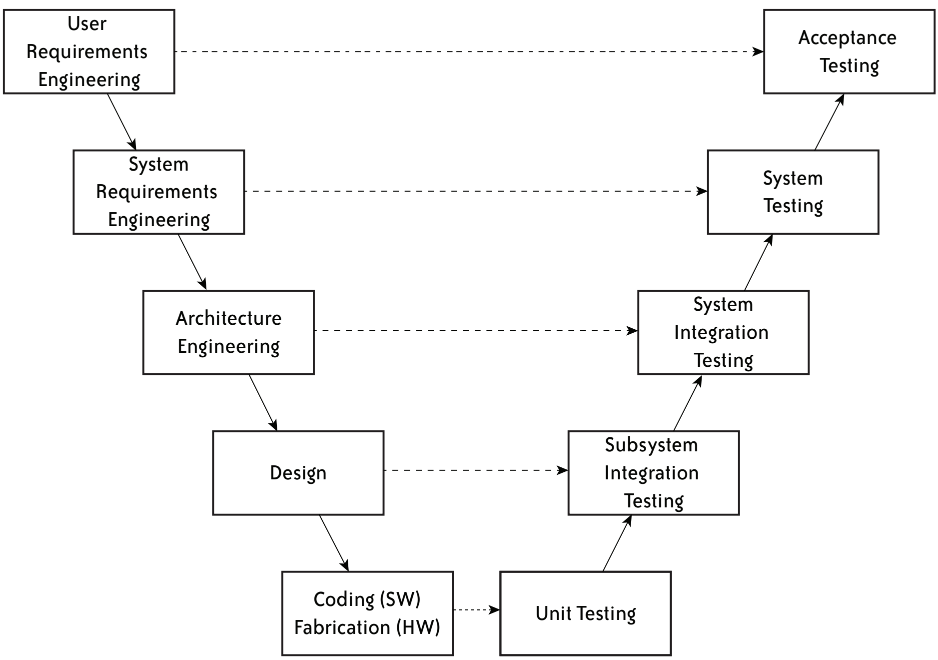

FIGURE 1.1 Traditional Single V model of system engineering activities

1. V stands for both validation and verification.

Figure 1.3 illustrates the Double-V model, which adds the corresponding tests to the Single V Model [Feiler 2012]. The key ideas to take away from this model are:

- Every executable work product should be tested. Testing need not, and in fact should not, be restricted to the implemented system and its parts. It is also important to test any executable requirements, architecture, and design. In this way, associated defects are found and fixed before they can migrate to the actual system and its parts. This typically involves testing executable requirements, architecture, or design models of the system under test (SUT) that are implemented in modeling languages (typically state-based and sufficiently formal) such as SpecTRM-RL, Architecture Analysis and Design Language (AADL), and Program Design Language (PDL); simulations of the SUT; or executable prototypes of the SUT.
- Tests should be created and performed as the corresponding work products are created. The short arrows with two arrowheads are used to show that (1) the executable work products can be developed first and used to drive the

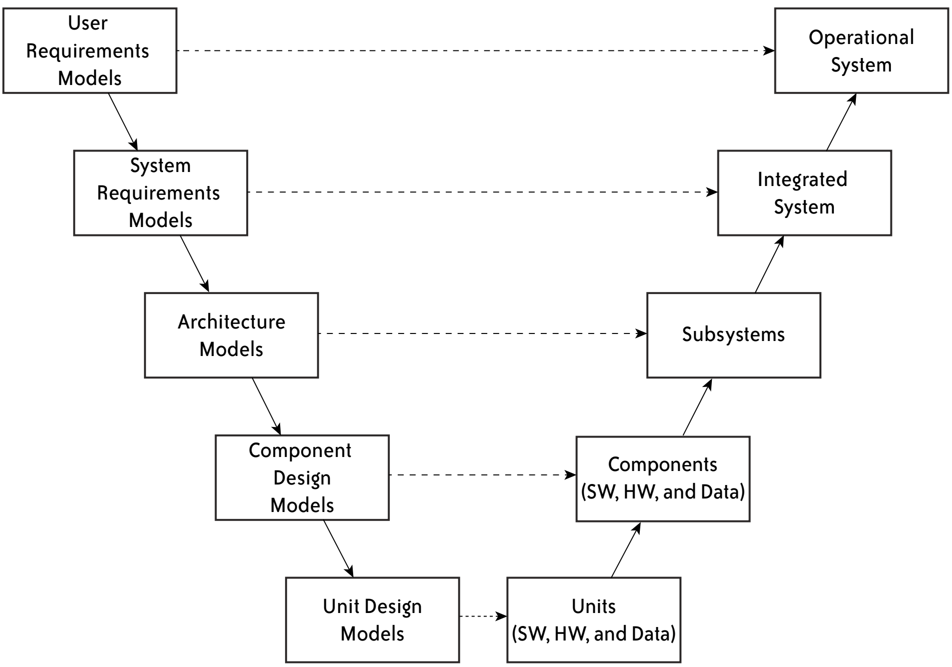

FIGURE 1.2 The Single V model of testable work products

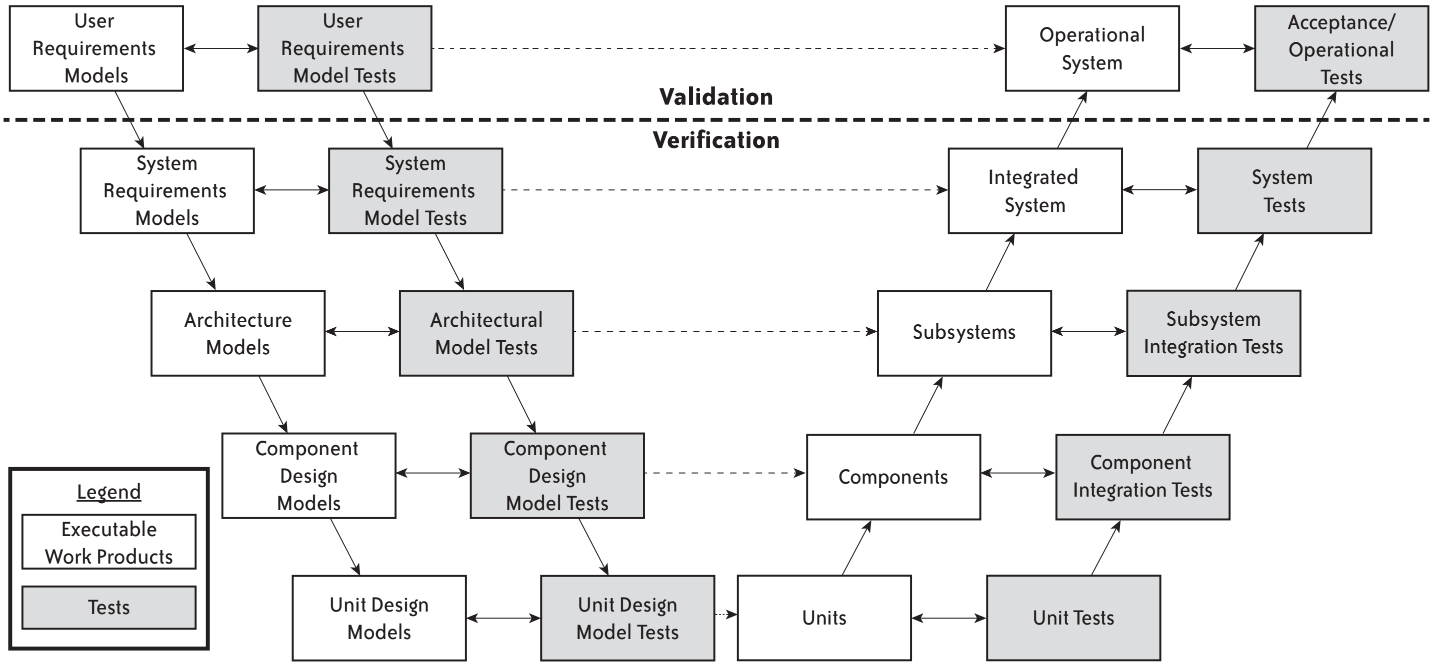

FIGURE 1.3 The Double V model of testable work products and corresponding tests

creation of the tests or (2) Test Driven Development (TDD) can be used, in which case the tests are developed before the work product they test.

- The top row of the model uses testing to *validate* that the system meets the needs of its stakeholders (that is, that the correct system is built). Conversely, the bottom four rows of the model use testing to *verify* that the system is built correctly (that is, architecture conforms to requirements, design conforms to architecture, implementation conforms to design, and so on).
- Finally, in practice, the two sides of the bottom row typically are combined so that the unit design models are incorporated into the units and so that the programming language is used as a program design language (PDL). Similarly, the unit design model tests are incorporated into the unit tests so that the same unit tests verify both the unit design and its implementation.

Figure 1.4 documents the Triple-V model, in which additional verification activities have been added to verify that the testing activities were performed properly. This provides evidence that testing is sufficiently complete and that it will not produce numerous false-positive and false-negative results.

Although the V models appear to show a sequential waterfall development cycle, they also can be used to illustrate an evolutionary (that is, incremental, iterative, and concurrent) development cycle that incorporates many small, potentially overlapping V models. However, when applying a V model to the agile development of a large, complex system, there are some potential complications that require more than a simple collection of small V models, such as:

- The architecturally significant requirements and the associated architecture need to be firmed up as rapidly as is practical because all subsequent increments depend on the architecture, which is difficult and expensive to modify once the initial increment(s) have been based on it.
- Multiple, cross-functional agile teams will be working on different components and subsystems simultaneously, so their increments must be coordinated across teams to produce consistent, testable components and subsystems that can be integrated and released.

Finally, it is interesting to note that these V models are applicable not just to the system under development but also to the development of the system's test environments or test beds and its test laboratories or facilities.

1.3 What Is a Defect?

A system *defect* (informally known as a bug) is a flaw or weakness in the system or one of its components that could cause it to behave in an unintended, unwanted manner or to exhibit an unintended, unwanted property. Defects are related to, but are different from:

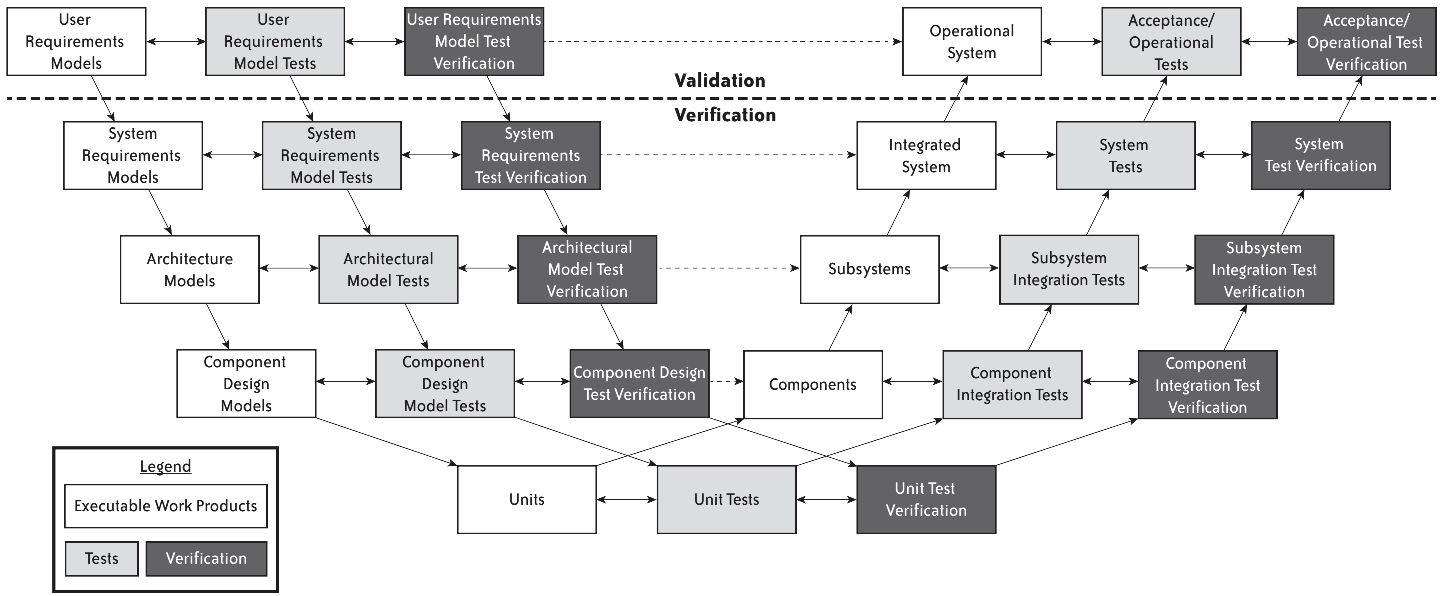

FIGURE 1.4 The Triple V model of work products, tests, and test verification

- **Errors** Human mistakes that cause the defect (for example, making a programming mistake or inputting incorrect data)
- **Faults** Incorrect conditions that are system-internal and not directly visible from outside the system's boundary (for example, the system stores incorrect data or is in an incorrect mode or state)
- **Failures** Events or conditions in which the system visibly behaves incorrectly or has incorrect properties (that is, one or more of its behaviors or properties are different from what its stakeholders can reasonably expect)

Common examples of defects include the following flaws or weaknesses:

- Defects can cause the SUT to violate specified (or unspecified) requirements, including:
  - Functional requirements
  - Data requirements
  - Interface requirements
  - Quality requirements
  - Architecture, design, implementation, and configuration constraints
- Defects can also result when the SUT conforms to incorrect or unnecessary requirements.
- Defects can cause the SUT to:
  - Fail to behave as it should
  - Be missing characteristics that it should have
  - Behave as it should not behave
  - Have characteristics that it should not have
- Defects can cause the SUT to be inconsistent with its architecture or design.
- Defects can result from incorrect or inappropriate architecture, design, implementation, or configuration decisions.
- Defects can violate design guidelines or coding standards.
- Defects can be safety or security vulnerabilities (for example, using inherently unsafe language features or failure to verify input data).

1.4 Why Is Testing Critical?

A National Institute of Standards & Technology (NIST) report [NIST 2002] states that inadequate testing methods and tools cost the US economy between $22.2 billion and $59.5 billion annually, with roughly half of these costs borne by software developers, in the form of extra testing, and half by software users, in the form of failure avoidance and mitigation efforts. The same study notes that between 25% and 90% of software development budgets are often spent on testing.

Testing is currently the most important of the standard verification and validation methods used during system development and maintenance. This is not because testing is necessarily the most effective and efficient way to verify that the system behaves as it should; it is not. (See Table 1.1, below.) Rather, it is because far more effort, funding, and time are expended on testing than on all other types of verification put together.

According to Capers Jones, most forms of testing find only about 35% of the code defects [Jones 2013b]. Similarly, on average, individual programmers find less than half the defects in their own software.

For example, Capers Jones analyzed data regarding defect identification effectiveness from projects that were completed in early 2013 and produced the results summarized in Table 1.1 [Jones 2013a]. Thus, the use of requirements inspections identified 87% of requirements defects and 25.6% of all defects in the software and its documentation. Similarly, static analysis of the code identified 87% of the code defects and 33.2% of all defects. Finally, a project that used all of these static verification methods identified 95% of all defects.

As Table 1.2 shows, static verification methods are cumulatively more effective at identifying defects except, surprisingly, documentation defects.

TABLE 1.1 Average Percentage of Defects Found as a Function of Static Verification Method and Defect Type

Verification Method	*Defect Type (Location)*					*Total Effectiveness*
	Requirements	*Architecture*	*Design*	*Code*	*Documentation*	
Requirements Inspection	87%	5%	10%	5%	8.5%	25.6%
Architecture Inspection	10%	85%	10%	2.5%	12%	14.9%
Design Inspection	14%	10%	87%	7%	16%	37.3%
Code Inspection	15%	12.5%	20%	85%	10%	70.1%
Static Analysis	2%	2%	7%	87%	3%	33.2%
IV&V	12%	10%	23%	7%	18%	16.5%
SQA Review	17%	10%	17%	12%	12.4%	28.1%
Total	95.2%	92.7%	96.1%	99.1%	58.8%	95.0%

Source: Jones 2013a

TABLE 1.2 Cumulative Effectiveness at Finding Defects by Static Verification Methods, Testing, and Both

Verification Method	*Defect Type (Location)*					*Total Effectiveness*
	Requirements	*Architecture*	*Design*	*Code*	*Documentation*	
Static	95.2%	92.7%	96.1%	99.1%	58.8%	95.0%
Testing	72.3%	74.0%	87.6%	93.4%	95.5%	85.7%
Total	98.11%	98.68%	99.52%	99.94%	98.13%	99.27%

Source: Jones 2013a

1.5 The Limitations of Testing

In spite of its critical nature, testing has a number of pitfalls that make it far less effective and efficient than it should be. Testing is relatively ineffective in the sense that a significant number of residual defects remain in a completed system when it is placed into operation. Testing is also relatively inefficient when you consider the large amount of effort, funding, and time that is currently spent to find defects.

According to Capers Jones, most types of testing find only about 35% of the software defects [Jones 2013]. This is consistent with the following, more detailed analysis of defect detection rates as a function of test type and test capabilities, as shown in Table 1.3 [McConnell 2004].

As Table 1.4 shows, no single type of testing is very effective at uncovering defects, regardless of defect type. Even when all of these testing methods are used on an average project, they still only identify four out of five of the code defects.

TABLE 1.3 Defect Detection Rate

	Defect Detection Rates		
Test Type	*Lowest*	*Mode*	*Highest*
Unit Test	15%	30%	50%
Component Test	20%	30%	35%
Integration Test	25%	35%	40%
System Test	25%	40%	55%
Regression Test	15%	25%	30%
Low-volume Beta Test	25%	35%	40%
High-volume Beta Test	60%	75%	85%

Source: McConnell 2004

TABLE 1.4 Defect Detection Rate

Static Verification	*Project Defect Detection Rate*		
	Worst	*Average*	*Best*
Desk Checking	23%	25%	27%
Static Analysis	0%	55%	55%
Inspection	0%	0%	93%
Static Subtotal	19%	64%	98%

Testing	*Project Defect Detection Rate*		
	Worst	*Average*	*Best*
Unit Test	28%	30%	32%
Function Test	31%	33%	35%
Regression Test	10%	12%	14%
Component Test	28%	30%	32%
Performance Test	6%	10%	14%
System Test	32%	34%	36%
Acceptance Test	13%	15%	17%
Testing Subtotal	72%	81%	87%
Cumulative Total	81.1%	95.6%	99.96%

Source: Jones 2013b

1.6 What Is a Testing Pitfall?

A testing pitfall is any decision, mindset, action, or failure to act that unnecessarily and, potentially unexpectedly, causes testing to be less effective, less efficient, or more frustrating to perform. Basically, a testing pitfall is a commonly occurring way to screw up testing, and projects fall into pitfalls when testers, managers, requirements engineers, and other testing stakeholders make testing-related mistakes that can have unintended negative consequences.

In a sense, the description of a testing pitfall constitutes a testing antipattern. However, the term pitfall was specifically chosen to evoke the image of a hidden or not easily identified trap for the unwary or uninitiated. As with any trap, it is better to avoid a testing pitfall than it is to have to dig one's self and one's project out of it after having fallen in.

1.7 Categorizing Pitfalls

Many testing pitfalls can occur during the development or maintenance of software-reliant systems and software applications. While no project is likely to be so poorly managed and executed as to experience the majority of these pitfalls, most projects will suffer several of them. Similarly, although these testing pitfalls do not guarantee failure, they definitely pose serious risks that need to be managed.

This book documents 92 pitfalls that have been observed to commonly occur during testing. These pitfalls are categorized as follows:

- **General Testing Pitfalls**
  - Test Planning and Scheduling Pitfalls
  - Stakeholder Involvement and Commitment Pitfalls
  - Management-Related Testing Pitfalls
  - Staffing Pitfalls
  - Test Process Pitfalls
  - Test Tools and Environments Pitfalls
  - Test Communication Pitfalls
  - Requirements-Related Testing Pitfalls

- **Test-Type-Specific Pitfalls**
  - Unit Testing Pitfalls
  - Integration Testing Pitfalls
  - Specialty Engineering Testing Pitfalls
  - System Testing Pitfalls
  - System of Systems (SoS) Testing Pitfalls
  - Regression Testing Pitfalls

Although each of these testing pitfalls has been observed on multiple projects, it is entirely possible that you might have testing pitfalls that are not addressed by this document. Please notify me of any new testing pitfalls you stumble across or any additional recommended changes to the current pitfalls so that I can incorporate them into future editions of this book.

1.8 Pitfall Specifications

Chapter 2 gives high-level descriptions of the different pitfalls, while Chapter 3 documents each testing pitfall with the following detailed information:

- **Title** A short, descriptive name of the pitfall
- **Description** A brief definition of the pitfall
- **Potential Applicability** The context in which the pitfall may be applicable
- **Characteristic Symptoms (or, How You Will Know)** Symptoms that indicate the possible existence of the pitfall
- **Potential Negative Consequences (Why You Should Care)** Potential negative consequences to expect if the pitfall is not avoided or mitigated[3]
- **Potential Causes** Potential root and proximate causes of the pitfall[4]
- **Recommendations (What You Should Do)** Recommended actions (prepare, enable, perform, and verify) to take to avoid or mitigate the pitfall[5]
- **Related Pitfalls** A list of other related testing pitfalls

A few words on word choice and grammar are probably appropriate before you start reading about the individual pitfalls:

- **Potential Applicability** You may fall into these pitfalls on your project, but then again you may not. Some pitfalls will be more probable and therefore more relevant than others. Of course, if you have already fallen into a given pitfall, it ceases to be potentially applicable and is now absolutely applicable. Because potential applicability currently exists, it is described in the present tense.
- **Characteristic Symptoms** You may have observed these symptoms in the past, and you may well be observing them now. They may even be waiting for you in the future. To save me from having to write all three tenses and, more importantly, to save you from having to read them all, I have listed all symptoms in present tense.
- **Potential Negative Consequences** Once again, you may have suffered these consequences in the past, or they may be happening now. These consequences might still be in the future and avoidable (or subject to mitigation) if you follow the appropriate recommendations now. These consequences are also listed in the present tense.

Note that sometimes the first symptom(s) of a pitfall are the negative consequence(s) you are suffering from because you fell into it. Therefore, it is not always obvious whether something should be listed under symptoms, consequences, or both. To avoid listing the same negative event or situation twice for the same pitfall, I have endeavored to include it only once under the most obvious heading.

- **Potential Causes** Finally, the causes may also lie in your past, your present, or your future. However, they seem to sound best when written in the past tense, for they must by their very nature precede the pitfall's symptoms and consequences.

This page intentionally left blank

INDEX

J

L

U