The Art of Software Testing
# Contents

List of Figures and Tables ix  
Preface xi  
Introduction xiii  

**Chapter 1** A Self-Assessment Test 1  

**Chapter 2** The Psychology and Economics of Program Testing 5  
  The Psychology of Testing 5  
  The Economics of Testing 9  
    Black-Box Testing 9  
    White-Box Testing 11  
  Software Testing Principles 14  
  Summary 20  

**Chapter 3** Program Inspections, Walkthroughs, and Reviews 21  
  Inspections and Walkthroughs 22  
  Code Inspections 24  
  An Error Checklist for Inspections 27  
    Data Reference Errors 27  
    Data-Declaration Errors 29  
    Computation Errors 30  
    Comparison Errors 31  
    Control-Flow Errors 32  
    Interface Errors 34  
    Input/Output Errors 35  
  Other Checks 38
Walkthroughs 38
Desk Checking 40
Peer Ratings 40
Summary 42

Chapter 4 Test-Case Design 43
White-Box Testing 44
Logic-Coverage Testing 44
Equivalence Partitioning 52
An Example 56
Boundary-Value Analysis 59
Cause-Effect Graphing 65
Error Guessing 88
The Strategy 90

Chapter 5 Module (Unit) Testing 91
Test-Case Design 92
Incremental Testing 105
Top-down versus Bottom-up Testing 109
Top-down Testing 110
Bottom-up Testing 116
A Comparison 118
Performing the Test 120

Chapter 6 Higher-Order Testing 123
Function Testing 129
System Testing 130
Facility Testing 133
Volume Testing 133
Stress Testing 134
Usability Testing 135
Security Testing 137
Performance Testing 137
Storage Testing 138
Configuration Testing 138
Contents

Compatibility/Configuration/Conversion Testing 138
Installability Testing 139
Reliability Testing 139
Recovery Testing 141
Serviceability Testing 142
Documentation Testing 142
Procedure Testing 142
Performing the System Test 143
Acceptance Testing 144
Installation Testing 144
Test Planning and Control 145
Test Completion Criteria 148
The Independent Test Agency 155

Chapter 7 Debugging 157
Debugging by Brute Force 158
Debugging by Induction 160
Debugging by Deduction 164
Debugging by Backtracking 168
Debugging by Testing 169
Debugging Principles 170
  Error-Locating Principles 170
  Error-Repairing Techniques 171
Error Analysis 173

Chapter 8 Extreme Testing 177
Extreme Programming Basics 178
Extreme Testing: The Concepts 183
  Extreme Unit Testing 183
  Acceptance Testing 185
Extreme Testing Applied 186
  Test-Case Design 186
  Test Driver and Application 189
Summary 191
Chapter 9  Testing Internet Applications 193
  Basic E-commerce Architecture 194
  Testing Challenges 196
  Testing Strategies 200
    Presentation Layer Testing 202
    Business Layer Testing 205
    Data Layer Testing 208
Appendix A Sample Extreme Testing Application 213
Appendix B Prime Numbers Less Than 1,000 221
Glossary 223
Index 227
List of Figures and Tables

Figure 2.1  Control-Flow Graph of a Small Program 12
Table 2.1  Vital Program Testing Guidelines 15
Figure 2.2  The Surprising Errors Remaining/Errors Found Relationship 20
Table 3.1  Inspection Error Checklist Summary, Part I 36
Table 3.2  Inspection Error Checklist Summary, Part II 37
Figure 4.1  A Small Program to Be Tested 47
Figure 4.2  Machine Code for the Program in Figure 4.1 50
Figure 4.3  A Form for Enumerating Equivalence Classes 54
Table 4.1  Equivalence Classes 57
Figure 4.4  Input to the MTEST Program 62
Figure 4.5  Basic Cause-Effect Graph Symbols 68
Figure 4.6  Sample Cause-Effect Graph 69
Figure 4.7  Logic Diagram Equivalent to Figure 4.6 69
Figure 4.8  Constraint Symbols 70
Figure 4.9  Symbol for “Masks” Constraint 71
Figure 4.10 Sample Cause-Effect Graph with “Exclusive” Constraint 71
Figure 4.11 Syntax of the DISPLAY Command 72
Figure 4.12 Beginning of the Graph for the DISPLAY Command 76
Figure 4.13 Full Cause-Effect Graph without Constraints 77
Figure 4.14 Complete Cause-Effect Graph of the DISPLAY Command 78
Figure 4.15 Considerations Used When Tracing the Graph 80
Figure 4.16 Sample Graph to Illustrate the Tracing Considerations 81
Figure 4.17 First Half of the Resultant Decision Table 82
Figure 4.18 Second Half of the Resultant Decision Table 84
Figure 5.1  Input Tables to Module BONUS 93
Figure 5.2  Module BONUS 94
Table 5.1  Situations Corresponding to the Decision Outcomes 96
List of Figures and Tables

Figure 5.3  Test Cases to Satisfy the Decision-Coverage Criterion 97
Table 5.2  Situations Corresponding to the Condition Outcomes 98
Figure 5.4  Test Cases to Satisfy the Condition-Coverage Criterion 99
Figure 5.5  Test Cases to Satisfy the Multicondition-Coverage Criterion 100
Figure 5.6  Supplemental Boundary-Value-Analysis Test Cases for BONUS 104
Figure 5.7  Sample Six-Module Program 106
Figure 5.8  Sample 12-Module Program 111
Figure 5.9  Second Step in the Top-down Test 113
Figure 5.10  Intermediate State in the Top-down Test 115
Figure 5.11  Intermediate State in the Bottom-up Test 118
Table 5.3  Comparison of Top-down and Bottom-up Testing 119
Figure 6.1  The Software Development Process 124
Figure 6.2  The Development Process with Intermediate Verification Steps 126
Figure 6.3  The Correspondence between Development and Testing Processes 127
Figure 6.4  The System Test 131
Table 6.1  Hours per Year for Various Uptime Requirements 141
Table 6.2  Hypothetical Estimate of When the Errors Might Be Found 151
Figure 6.5  Estimating Completion by Plotting Errors Detected by Unit Time 153
Figure 6.6  Postmortem Study of the Testing Processes of a Large Project 154
Figure 7.1  The Inductive Debugging Process 161
Figure 7.2  A Method for Structuring the Clues 162
Figure 7.3  An Example of Clue Structuring 163
Figure 7.4  The Deductive Debugging Process 165
Figure 7.5  Test Case Results from the DISPLAY Command 166
Table 8.1  The 12 Practices of Extreme Programming 180
Table 8.2  Test Case Descriptions for check4Prime.java 187
Table 8.3  Test Driver Methods 189
Figure 9.1  Typical Architecture of an E-Commerce Site 195
Table 9.1  Examples of Presentation, Business, and Data Tier Testing 199
Figure 9.2  Detailed View of Internet Application Architecture 202
Table 9.2  Items to Test in Each Tier 203
In 1979, Glenford Myers published a book that turned out to be a classic. Myers’s original *The Art of Software Testing* stood the test of time, 25 years on the publisher’s list of available books. This fact alone is a testament to the solid, basic, and valuable nature of his work.

During that same time, the current authors of the updated version of this book published collectively more than 120 books, most of them on computer software topics. Some of these books sold very well, going through multiple versions. (Corey Sandler’s *Fix Your Own PC* is in its seventh edition as this book is written, and Tom Badgett’s books on Microsoft PowerPoint and other Office titles went through four or more editions, for example.) Nevertheless, none of the current authors’ books remained current more than a few years.

What is the difference? These newer books cover more transient topics: operating systems, applications software, security, communications technology, and hardware configurations. Rapid changes in computer hardware and software technology during the 1980s and 1990s necessitated frequent changes and updates to these topics.

During that period dozens—perhaps even hundreds—of books also were published about software testing. They, too, took a more transient approach to the topic.

Myers’s *The Art of Software Testing*, on the other hand, gave the industry a long-lasting, foundational guide to one of the most important computer topics: How do you ensure that all of the software you produce does what it was designed to do and, just as important, does not do what it isn’t supposed to do?
The version you are reading today retains that same foundational philosophy. We have updated the examples to include more current programming languages, and we have addressed topics that no one knew about when Myers wrote the first edition: Web programming, e-commerce, and Extreme Programming and Testing.

But we didn’t forget that a new classic must be true to its roots, so this version also offers you a software testing philosophy that is all Glenford Myers, a philosophy and a process that work across current and unforeseeable future hardware and software platforms. Hopefully this, too, is a book that will span a generation of software designers and developers.
At the time this book was first published, in 1979, it was a well-known rule of thumb that in a typical programming project approximately 50 percent of the elapsed time and more than 50 percent of the total cost were expended in testing the program or system being developed.

Today, a quarter of the century later, the same is still true. There are new development systems, languages with built-in tools, and programmers who are used to developing more on the fly. But testing still plays an important part in any software development project.

Given these facts, you might expect that by this time program testing would have been refined into an exact science. This is far from true. In fact, less seems to be known about software testing than about any other aspect of software development. Furthermore, testing has been an out-of-vogue subject—it was true when this book was first published and, unfortunately, it is still true today. Today there are more books and articles about software testing, meaning that, at least, the topic has more visibility than it did when this book was first published. But testing remains among the “dark arts” of software development.

This would be more than enough reason to update this book on the art of software testing, but there are additional motivations. At various times, we have heard professors and teaching assistants say, “Our students graduate and move into industry without any substantial knowledge of how to go about testing a program. Moreover, we rarely have any advice to provide in our introductory courses on how a student should go about testing and debugging his or her exercises.”
So, the purpose of this updated edition of *The Art of Software Testing* is the same as it was in 1979: to fill these knowledge gaps for the professional programmer and the student of computer science. As the title implies, the book is a practical, rather than theoretical, discussion of the subject, complete with updated language and process discussions. Although it is possible to discuss program testing in a theoretical vein, this book is intended to be a practical, “both feet on the ground” handbook. Hence, many subjects related to program testing, such as the idea of mathematically proving the correctness of a program, were purposefully excluded.

Chapter 1 is a short self-assessment test that every reader should take before reading further. It turns out that the most important practical information that you must understand about program testing is a set of philosophical and economic issues; these are discussed in Chapter 2. Chapter 3 discusses the important concept of non-computer-based code walk-throughs or inspections. Rather than focus attention on the procedural or managerial aspects of this concept, as most discussions do, Chapter 3 discusses it from a technical, how-to-find-errors point of view.

The knowledgeable reader will realize that the most important component in the bag of tricks of a program tester is the knowledge of how to write effective test cases; this is the subject of Chapter 4. Chapters 5 and 6 discuss, respectively, the testing of individual modules or subroutines and the testing of larger entities, with Chapter 7 presenting some practical advice on program debugging. Chapter 8 discusses the concepts of extreme programming and extreme testing, while Chapter 9 shows how to use other features of program testing detailed elsewhere in this book with Web programming, including e-commerce systems.

This book has three major audiences. Although we hope that not everything in this book will be new information to the professional programmer, it should add to the professional’s knowledge of testing techniques. If the material allows you to detect just one more bug in one program, the price of the book will have been recovered many times over. The second audience is the project manager, since the book contains new, practical information on the management of the
testing process. The third audience is the programming or computer science student; the goal here is to expose the student to the problems of program testing and to provide a set of effective techniques. It is suggested that the book be used as a supplement in programming courses such that the student is exposed to the subject of software testing at an early time in his or her education.

Glenford J. Myers
Tom Badgett
Todd M. Thomas
Corey Sandler
The Art of Software Testing
CHAPTER 1

A Self-Assessment Test

Since this book was first published 25 years ago, software testing has become both easier and more difficult than ever.

Software testing is more difficult because of the vast array of programming languages, operating systems, and hardware platforms that have evolved. And, while relatively few people used computers in the 1970s, today virtually anyone in business or education could hardly complete a day’s work without using a computer. Furthermore, the machines themselves are hundreds of times more powerful than those early devices.

Therefore, the software we write today potentially touches millions of people, enabling them to do their jobs effectively and efficiently—or causing them untold frustration and the cost of lost work or lost business. This is not to say that software is more important today than it was when the first edition of this book was published, but it is safe to say that computers—and the software that drives them—certainly affect more people and more businesses today.

Software testing is easier, in some ways, because the array of software and operating systems is much more sophisticated than ever, providing intrinsic well-tested routines that can be incorporated into applications without the need for a programmer to develop them from scratch. Graphical user interfaces (GUIs), for example, can be built from a development language’s libraries, and, since they are pre-programmed objects that have been debugged and tested previously, the need for testing them as part of a custom application is much reduced.

Software testing is a process, or a series of processes, designed to make sure computer code does what it was designed to do and that it
does not do anything unintended. Software should be predictable and consistent, offering no surprises to users. In this book we will look at many approaches to achieving this goal.

Now, before we start the book, we’d like you to take a short exam. We want you to write a set of test cases—specific sets of data—to properly test a relatively simple program. Create a set of test data for the program—data the program must handle correctly to be considered a successful program. Here’s a description of the program:

The program reads three integer values from an input dialog. The three values represent the lengths of the sides of a triangle. The program displays a message that states whether the triangle is scalene, isosceles, or equilateral.

Remember that a scalene triangle is one where no two sides are equal, whereas an isosceles triangle has two equal sides, and an equilateral triangle has three sides of equal length. Moreover, the angles opposite the equal sides in an isosceles triangle also are equal (it also follows that the sides opposite equal angles in a triangle are equal), and all angles in an equilateral triangle are equal.

Evaluate your set of test cases by using it to answer the following questions. Give yourself one point for each “yes” answer.

1. Do you have a test case that represents a valid scalene triangle? (Note that test cases such as 1,2,3 and 2,5,10 do not warrant a “yes” answer because there does not exist a triangle having these dimensions.)
2. Do you have a test case that represents a valid equilateral triangle?
3. Do you have a test case that represents a valid isosceles triangle? (Note that a test case representing 2,2,4 would not count because it is not a valid triangle.)
4. Do you have at least three test cases that represent valid isosceles triangles such that you have tried all three permutations of two equal sides (such as, 3,3,4; 3,4,3; and 4,3,3)?
5. Do you have a test case in which one side has a zero value?
6. Do you have a test case in which one side has a negative value?
7. Do you have a test case with three integers greater than zero such that the sum of two of the numbers is equal to the third? (That is, if the program said that 1,2,3 represents a scalene triangle, it would contain a bug.)
8. Do you have at least three test cases in category 7 such that you have tried all three permutations where the length of one side is equal to the sum of the lengths of the other two sides (for example, 1,2,3; 1,3,2; and 3,1,2)?
9. Do you have a test case with three integers greater than zero such that the sum of two of the numbers is less than the third (such as 1,2,4 or 12,15,30)?
10. Do you have at least three test cases in category 9 such that you have tried all three permutations (for example, 1,2,4; 1,4,2; and 4,1,2)?
11. Do you have a test case in which all sides are zero (0,0,0)?
12. Do you have at least one test case specifying noninteger values (such as 2.5,3.5,5.5)?
13. Do you have at least one test case specifying the wrong number of values (two rather than three integers, for example)?
14. For each test case did you specify the expected output from the program in addition to the input values?

Of course, a set of test cases that satisfies these conditions does not guarantee that all possible errors would be found, but since questions 1 through 13 represent errors that actually have occurred in different versions of this program, an adequate test of this program should expose at least these errors.

Now, before you become concerned about your own score, consider this: In our experience, highly qualified professional programmers score, on the average, only 7.8 out of a possible 14. If you’ve done better, congratulations; if not, we’ll try to help.

The point of the exercise is to illustrate that the testing of even a trivial program such as this is not an easy task. And if this is true, consider the difficulty of testing a 100,000-statement air traffic control
system, a compiler, or even a mundane payroll program. Testing also becomes more difficult with the object-oriented languages such as Java and C++. For example, your test cases for applications built with these languages must expose errors associated with object instantiation and memory management.

It might seem, from working with this example, that thoroughly testing a complex, real-world program would be impossible. Not so! Although the task can be daunting, adequate program testing is a very necessary—and achievable—part of software development, as you will learn in this book.
Software testing is a technical task, but it also involves some important considerations of economics and human psychology.

In an ideal world, we would want to test every possible permutation of a program. In most cases, however, this simply is not possible. Even a seemingly simple program can have hundreds or thousands of possible input and output combinations. Creating test cases for all of these possibilities is impractical. Complete testing of a complex application would take too long and require too many human resources to be economically feasible.

In addition, the software tester needs the proper attitude (perhaps “vision” is a better word) to successfully test a software application. In some cases, the tester’s attitude may be more important than the actual process itself. Therefore, we will start our discussion of software testing with these issues before we delve into the more technical nature of the topic.

The Psychology of Testing

One of the primary causes of poor program testing is the fact that most programmers begin with a false definition of the term. They might say:

• “Testing is the process of demonstrating that errors are not present.”
or

• “The purpose of testing is to show that a program performs its intended functions correctly.”

or

• “Testing is the process of establishing confidence that a program does what it is supposed to do.”

These definitions are upside-down.

When you test a program, you want to add some value to it. Adding value through testing means raising the quality or reliability of the program. Raising the reliability of the program means finding and removing errors.

Therefore, don’t test a program to show that it works; rather, you should start with the assumption that the program contains errors (a valid assumption for almost any program) and then test the program to find as many of the errors as possible.

Thus, a more appropriate definition is this:

*Testing is the process of executing a program with the intent of finding errors.*

Although this may sound like a game of subtle semantics, it’s really an important distinction. Understanding the true definition of software testing can make a profound difference in the success of your efforts.

Human beings tend to be highly goal-oriented, and establishing the proper goal has an important psychological effect. If our goal is to demonstrate that a program has no errors, then we will subconsciously be steered toward this goal; that is, we tend to select test data that have a low probability of causing the program to fail. On the other hand, if our goal is to demonstrate that a program has errors, our test data will have a higher probability of finding errors. The latter approach will add more value to the program than the former.

This definition of testing has myriad implications, many of which are scattered throughout this book. For instance, it implies that testing is a destructive process, even a sadistic process, which explains