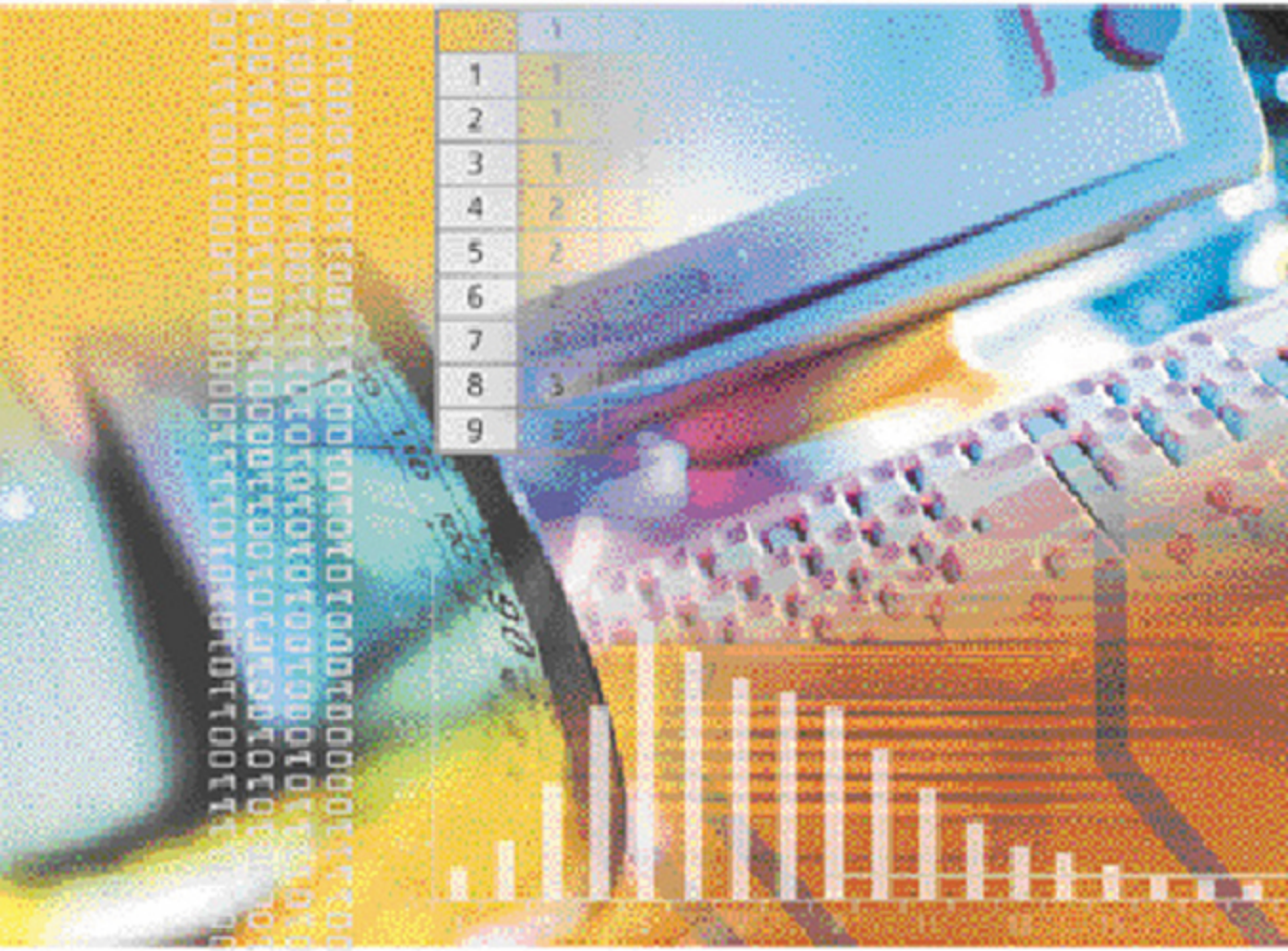
A Practitioner's Guide to Software Test Design



Lee Copeland

Dedication

To my wife Suzanne, and our wonderful children and grandchildren

Shawn and Martha
Andrew and Cassandra
David
Cathleen
Katelynn and Kiley
Melissa and Jay
Ross, Elizabeth, and Miranda
Brian and Heather
Cassidy and Caden
Thomas and Jeni
Carrie
Sundari
Rajan

and to Wayne, Jerry, Dani, Ron, and Rayanne for their encouragement over the years.

Table Of Contents

Preface

A Practitioner's Guide To Software Test Design contains today's important current test design approaches in one unique book. Until now, software testers had to search through a number of books, periodicals, and web sites to locate this vital information.

The book focuses only on software test design, not related subjects such as test planning, test management, test team development, etc. While those are important in software testing, they have often overshadowed what testers really need—the more practical aspects of testing, specifically test case design. Other excellent books can guide you through the overall process of software testing. One of my favorites is *Systematic Software Testing*.

A Practitioner's Guide To Software Test Design illustrates each test design approach through detailed examples and step-by-step instructions. These lead the reader to a clear understanding of each test design technique.

Importance of Test Design

"The act of careful, complete, systematic, test design will catch as many bugs as the act of testing. … Personally, I believe that it's far more effective."

- Boris Beizer

Today's Testing Challenges

For any system of interesting size it is impossible to test all the different logic paths and all the different input data combinations. Of the infinite number of choices, each one of which is worthy of some level of testing, testers can only choose a very small subset because of resource constraints. The purpose of this book is to help the tester wisely choose such subsets; to choose those tests that are most likely to discover defects

It is vital to choose test cases wisely. Missing a defect can result in significant losses to our organization if a defective system is placed into production.

A Practitioner's Guide To Software Test Design describes a set of key test design strategies that improve both the efficiency and effectiveness of software test professionals.

Structure And Approach

A Practitioner's Guide To Software Test Design explains the most important test design techniques in use today. Some of these techniques are classics and well known throughout the testing community. Some have been around for a while but are not well-known among test engineers. Still others are not widely known, but should be because of their effectiveness. This book brings together all these techniques into one volume, helping the test designer become more efficient and effective in testing.

Each test design technique is approached from a practical, rather than a theoretical basis. Each test design technique is first introduced through a simple example, then explained in detail. When possible, additional examples of its use are presented. The types of problems on which the approach can be used, along with its limitations, are described. Each test design technique chapter ends with a summary of its key points, along with exercises the reader can use for practice, and references for further reading. Testers can use the techniques presented immediately on their projects.

Each test design approach is described in a self-contained chapter. Because the chapters are focused, concise, and independent they can be read "out of order." Testers can read the chapters that are most relevant to their work at the moment.

A Note From The Author

I love a good double integral sign

$$\iint$$

as much as the next tester but we're going to concentrate on the practical, not the theoretical.

Audience

This book was written specifically for :

- Software Test Engineers who have the primary responsibility for test case design. This book details the most efficient and effective methods for creating test cases.

- Software Developers who, with the advent of Extreme Programming and other agile development methods, are being expected to do more and better testing of the software they write. Many developers have not been exposed to the design techniques described in this book.

- Test and Development Managers who must understand., at least in principle, the work their staff performs. Not only does this book provides an overview of important test design methods, it will assist managers in estimating the effort, time, and cost of good testing.

- Instructors and professors who are searching for an excellent reference for a course in software test design techniques.

Acknowledgements

The following reviewers have provided invaluable assistance in the writing of this book: Chuck Allison, Dale Perry, Danny Faught, Dorothy Graham, Geoff Quentin, James Bach, Jon Hagar, Paul Gerrard, Rex Black, Rick Craig, Sid Snook, and Wayne Middleton. In addition, Robert Coutre was invaluable in

the editing process. My sincere thanks to each of you. Any faults in this book should be attributed directly to them. (Just kidding!)

Some Final Comments

This book contains a number of references to Web sites. These references were correct when the manuscript was submitted to the publisher. Unfortunately, they may have become broken. Such is the Web. Try Google™ to locate them.

It has become standard practice for authors to include a pithy quotation on the title page of each chapter. Unfortunately, the practice has become so prevalent that all the good quotations have been used. Just for fun, I have chosen instead to include on each chapter title page a winning entry from the 2003 Bulwer-Lytton Fiction Contest (www.bulwer-lytton.com). Since 1982, the English Department at San Jose State University has sponsored this event, a competition that challenges writers to compose the opening sentence to the worst of all possible novels. It was inspired by Edward George Bulwer-Lytton who began his novel *Paul Clifford* with:

> "It was a dark and stormy night; the rain fell in torrents—except at occasional intervals, when it was checked by a violent gust of wind which swept up the streets (for it is in London that our scene lies), rattling along the housetops, and fiercely agitating the scanty flame of the lamps that struggled against the darkness."

My appreciation to Dr. Scott Rice of San Jose State University for permission to use these exemplary illustrations of bad writing. Hopefully, nothing in this book will win this prestigious award.

Acknowledgements

The photo of James Bach is used by permission.
The photo of Alistair Cockburn is used by permission.
The caricature of Mick Jagger is owned and copyrighted by Martin O'Loughlin and used by permission.
Clipart copyright by Corel Corporation and used under a licensing agreement.

References

Craig, Rick D. and Stefan P. Jaskiel (2002). *Systematic Software Testing*. Artech House Publishers.

Chapter 1 –
The Testing Process

"The flock of geese flew overhead in a 'V' formation - not in an old-fashioned-looking Times New Roman kind of a 'V', branched out slightly at the two opposite arms at the top of the 'V', nor in a more modern-looking, straight and crisp, linear Arial sort of 'V' (although since they were flying, Arial might have been appropriate), but in a slightly asymmetric, tilting off-to-one-side sort of italicized Courier New-like 'V' - and LaFonte knew that he was just the type of man to know the difference."[*]

— John Dotson

[*] If you think this quotation has nothing to do with software testing you are correct. For an explanation please read "Some Final Comments" in the Preface.

Testing

What is testing? While many definitions have been written, at its core testing is the process of comparing "what is" with "what ought to be." A more formal definition is given in the IEEE Standard 610.12-1990, "IEEE Standard Glossary of Software Engineering Terminology" which defines "testing" as:

> "The process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component."

The "specified conditions" referred to in this definition are embodied in test cases, the subject of this book.

Key Point

At its core, testing is the process of comparing "what is" with "what ought to be."

Rick Craig and Stefan Jaskiel propose an expanded definition of software testing in their book, *Systematic Software Testing*.

> "Testing is a concurrent lifecycle process of engineering, using and maintaining testware in order to measure and improve the quality of the software being tested."

This view includes the planning, analysis, and design that leads to the creation of test cases in addition to the IEEE's focus on test execution.

Different organizations and different individuals have varied views of the purpose of software testing. Boris Beizer describes five levels of testing maturity. (He called them phases but today we know the "correct" term is "levels" and there are always five of them.)

Level 0 – "There's no difference between testing and debugging. Other than in support of debugging, testing has no purpose." Defects may be stumbled upon but there is no formalized effort to find them.

Level 1 – "The purpose of testing is to show that software works." This approach, which starts with the premise that the software is (basically) correct, may blind us to discovering defects. Glenford Myers wrote that those performing the testing may subconsciously select test cases that should not fail. They will not create the "diabolical" tests needed to find deeply hidden defects.

Level 2 – "The purpose of testing is to show the software doesn't work." This is a very different mindset. It assumes the software doesn't work and challenges the tester to find its defects. With this approach we will consciously select test cases that evaluate the system in its nooks and crannies, at its boundaries, and near its edges, using diabolically constructed test cases.

Level 3 – "The purpose of testing is not to prove anything, but to reduce the perceived risk of not working to an acceptable value." While we can prove a system incorrect with only one test case, it is impossible to ever prove it correct. To do so would require us to test every possible valid combination of input data and every possible invalid combination of input data. Our goals are to understand the quality of the software in terms of its defects; to furnish the programmers with information about the software's deficiencies and to provide management with an evaluation of the negative impact on our organization if we shipped this system to customers in its present state.

Level 4 – "Testing is not an act. It is a mental discipline that results in low-risk software without much testing effort." At this maturity level we focus on making software more testable from its inception. This includes reviews and inspections of its requirements, design, and code. In addition, it means writing code that incorporates facilities the tester can easily use to interrogate it while it is executing. Further, it means writing code that is self-diagnosing, that reports errors rather than requiring testers to discover them.

Current Challenges

When I ask my students about the challenges they face in testing they typically reply:

- Not enough time to test properly
- Too many combinations of inputs to test
- Not enough time to test well
- Difficulty in determining the expected results of each test
- Non-existent or rapidly changing requirements
- Not enough time to test thoroughly
- No training in testing processes
- No tool support
- Management that either doesn't understand testing or (apparently) doesn't care about quality
- Not enough time

This book does not contain "magic pixie dust" that you can use to create additional time, better requirements, or more enlightened management. It does, however, contain techniques that will make you more efficient and effective in your testing by helping you choose and construct test

cases that will find substantially more defects than you have in the past while using fewer resources.

Test Cases

To be most effective and efficient, test cases must be designed, not just slapped together. The word "design" has a number of definitions:

Key Point

To be most effective and efficient, test cases must be designed, not just slapped together.

1. To conceive or fashion in the mind; invent: *design a good reason to attend the STAR testing conference.* To formulate a plan for; devise: *design a marketing strategy for the new product.*
2. To plan out in systematic, usually documented form: *design a building; design a test case.*.
3. To create or contrive for a particular purpose or effect: *a game designed to appeal to all ages.*
4. To have as a goal or purpose; intend.
5. To create or execute in an artistic or highly skilled manner.

Each of these definitions applies to good test case design. Regarding test case design, Roger Pressman wrote:

> "The design of tests for software and other engineering products can be as challenging as the initial design of the product itself. Yet … software engineers often treat testing as an afterthought, developing test cases that 'feel right' but have little assurance of being complete. Recalling the objectives of testing, we must design tests that have the highest likelihood of finding the most errors with a minimum amount of time and effort."

Well designed test cases are composed of three parts:

- Inputs
- Outputs
- Order of execution

Key Point

Test cases consist of inputs, outputs, and order of execution.

Inputs

Inputs are commonly thought of as data entered at a keyboard. While that is a significant source of system input. data can come from other sources—data from interfacing systems, data from interfacing devices, data read from files or databases, the state the system is in when the data arrives, and the environment within which the system executes.

Outputs

Outputs have this same variety. Often outputs are thought of as just the data displayed on a computer screen. In addition, data can be sent to interfacing systems and to external devices. Data can be written to files or databases. The state or the environment may be modified by the system's execution.

All of these relevant inputs and outputs are important components of a test case. In test case design, determining the expected outputs is the function of an "oracle."

An oracle is any program, process, or data that provides the test designer with the expected result of a test. Beizer lists four types of oracles:

- Kiddie Oracles – Just run the program and see what comes out. If it looks about right, it must be right.

- Regression Test Suites – Run the program and compare the output to the results of the same tests run against a previous version of the program.

- Validated Data – Run the program and compare the results against a standard such as a table, formula, or other accepted definition of valid output.

- Purchased Test Suites – Run the program against a standardized test suite that has been previously created and validated. Programs like compilers, web browsers, and SQL (Structured Query Language) processors are often tested against such suites.

- Existing Program – Run the program and compare the output to another version of the program.

Order of Execution

There are two styles of test case design regarding order of test execution.

- Cascading test cases – Test cases may build on each other. For example, the first test case exercises a particular feature of the software and then leaves the system in a state such that the second test case can be executed. In testing a database consider these test cases:

    1. Create a record
    2. Read the record
    3. Update the record
    4. Read the record
    5. Delete the record
    6. Read the deleted record

Each of these tests builds on the previous tests. The advantage is that each test case is typically smaller and simpler. The disadvantage is that if one test fails, the subsequent tests may be invalid.

- Independent test cases – Each test case is entirely self contained. Tests do not build on each other or require that other tests have successfully executed. The advantage is that any number of tests can be executed in any order. The disadvantage is that each test tends to be larger and more complex and thus more difficult to design, create, and maintain.

Types Of Testing

Testing is often divided into black box testing and white box testing.

Black box testing is a strategy in which testing is based solely on the requirements and specifications. Unlike its complement, white box testing, black box testing requires no knowledge of the internal paths, structure or implementation of the software under test.

White box testing is a strategy in which testing is based on the internal paths, structure, and implementation of the software under test. Unlike its complement, black box testing, white box testing generally requires detailed programming skills.

An additional type of testing is called gray box testing. In this approach we peak into the "box" under test just long enough to understand how it has been implemented. Then we close up the box and use our knowledge to choose more effective black box tests.

Testing Levels

Typically testing, and therefore test case design, is performed at four different levels:

Key Point

The classical testing levels are unit, integration, system, and acceptance.

- Unit Testing – A unit is the "smallest" piece of software that a developer creates. They are typically the work of one programmer and are stored in a single disk file. Different programming languages have different units: In C++ and Java the unit is the class. In C the unit is the function. In less structured languages like Basic and COBOL the unit may be the entire program.

- Integration Testing – In integration we assemble units together into subsystems and finally into systems. It is possible for units to function perfectly in isolation but to fail when integrated together. A classic example is this C program and its subsidiary function:

```
// main program
void oops int;
main () {
  oops(42);  // call the oops function passing an integer
}

// function oops
#include <stdio.h>
void oops (double x){  // expects a double length integer,
                       // not a regular integer
 printf ("%f\n",x);    // will always print 0
                       // no matter what has been passed
  return;
}
```

If these units were tested individually, each would appear to function correctly. In this case, the defect only appears when the two units are integrated together. The

main program passes an integer to function oops but oops expects a double length integer and trouble ensues. It is vital to perform integration testing as the integration process proceeds.

- System Testing – A system consists of all of the software (and possibly hardware, user manuals, training materials, etc.) that make up the product delivered to the customer. System testing focuses on defects that arise at this highest level of integration. Typically system testing includes many types of testing: functionality, usability, security, internationalization and localization, reliability and availability, capacity, performance, backup and recovery, portability, and many more. This book deals only with functionality testing. While the other types of testing are important, they are beyond the scope of this volume.

- Acceptance Testing – Acceptance testing is defined as that testing, which when completed successfully, will result in the customer accepting the software and giving us their money. From the customer's point of view, they would generally like the most exhaustive acceptance testing possible (equivalent to the level of system testing). From the vendor's point of view, we would generally like the minimum level of testing possible that would result in money changing hands. Typical strategic questions that should be addressed before acceptance testing are: Who defines the level of the acceptance testing? Who creates the test scripts? Who executes the tests What is the pass/fail criteria for the acceptance test? When and how do we get paid?

Not all systems are amenable to using these levels. These levels assume that there is a significant period of time between developing units and integrating them into subsystems and then

into systems. In Web development it is often possible to go from concept to code to production in a matter of hours. In that case, the unit-integration-system levels don't make much sense. Many Web testers use an alternate set of levels:

- Code quality
- Functionality
- Usability
- Performance
- Security

The Impossibility Of Testing Everything

In his monumental book *Testing Object Oriented Systems*, Robert Binder provides an excellent example of the impossibility of testing "everything." Consider the following program:

```
int blech (int j) {
  j = j - 1;            // should be j = j + 1
  j = j / 30000;
  return j;
}
```

Note that the second line is incorrect! The function blech accepts an integer j, subtracts one from it, divides it by 30000 (integer division, whole numbers, no remainder) and returns the value just computed. If integers are implemented using 16 bits on this computer executing this software, the lowest possible input value is –32768 and the highest is 32767. Thus there are 65,536 possible inputs into this tiny program. (Your organization's programs are probably larger.) Will you have the time (and the stamina) to create 65,536 test cases? Of course not. So which input values do we choose? Consider the following input values and their ability to detect this defect.

Input (j)	Expected Result	Actual Result
1	0	0
42	0	0
40000	1	1
-64000	-2	-2

Oops! Note that none of the test cases chosen have detected this defect. In fact only 4 of the possible 65,546 input values will find this defect. What is the chance that you will choose all four? What is the chance you will choose one of the four? What is the chance you will win the Powerball™ lottery? Is your answer the same to each of these three questions?

Summary

- Testing is a concurrent lifecycle process of engineering, using and maintaining testware in order to measure and improve the quality of the software being tested. (Craig and Jaskiel)

- The design of tests for software and other engineering products can be as challenging as the initial design of the product itself. Yet … software engineers often treat testing as an afterthought, developing test cases that 'feel right' but have little assurance of being complete. Recalling the objectives of testing, we must design tests that have the highest likelihood of finding the most errors with a minimum amount of time and effort. (Pressman)

- Black box testing is a strategy in which testing is based solely on the requirements and specifications. White box testing is a strategy in which testing is based on the internal paths, structure, and implementation of the software under test.

- Typically testing, and therefore test case design, is performed at four different levels: Unit, Integration, System, and Acceptance.

Exercise

1. Which four inputs to the blech routine will find the hidden defect? How did you determine them? What does this suggest to you as an approach to finding other defects?

References

Beizer, Boris (1990). *Software Testing Techniques* (Second Edition). Van Nostrand Reinhold.

Binder, Robert V. (2000). *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley.

Craig, Rick D. and Stefan P. Jaskiel (2002). *Systematic Software Testing*. Artech House Publishers.

IEEE Standard 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology, 1991.

Meyers, Glenford (1979). *The Art of Software Testing*. John Wiley & Sons.

Pressman, Roger S. (1982). *Software Engineering: A Practitioner's Approach* (Fourth Edition). McGraw-Hill.