PRASHANT JOSHI
3-JUNE-2004

SOFTWARE TESTING

A Craftsman's Approach

SECOND EDITION

Paul C. Jorgensen, Ph.D.
Department of Computer Science and Information Systems
Grand Valley State University
Allendale, Michigan
and
Software Paradigms
Rockford, Michigan

CRC PRESS
Boca Raton London New York Washington, D.C.

Library of Congress Cataloging-in-Publication Data

Jorgensen, Paul.
Software testing : a craftman's approach / Paul C. Jorgensen.—2nd ed.
p. cm.
Includes bibliographical references and index.
ISBN 0-8493-0809-7
1. Computer software—Testing. I. Title.

QA76.76.T48 J67 2002
005.1'4—dc21 2002022405

International Standard Book Number 0-8493-0809-7
Library of Congress Card Number 2002022405
Printed in the United States of America 1 2 3 4 5 6 7 8 9 0
Printed on acid-free paper

Dedication

To Carol, Kirsten, and Katia

Preface to the First Edition

We huddled around the door to the conference room, each taking a turn looking through the small window. Inside, a recently hired software designer had spread out source listings on the conference table, and carefully passed a crystal hanging from a long chain over the source code. Every so often, the designer marked a circle in red on the listing. Later, one of my colleagues asked the designer what he had been doing in the conference room. The nonchalant reply: "Finding the bugs in my program." This is a true story, it happened in the mid-1980s when people had high hopes for hidden powers in crystals.

In a sense, the goal of this book is to provide you with a better set of crystals. As the title suggests, I believe that software (and system) testing is a craft, and I think I have some mastery of that craft. Out of a score of years developing telephone switching systems, I spent about a third of that time on testing: defining testing methodologies and standards, coordinating system testing for a major international telephone toll switch, specifying and helping build two test execution tools (now we would call them CASE tools), and a fair amount of plain, hands-on testing. For the past seven years, I have been teaching software engineering at the university graduate level. My academic research centers on specification and testing. Adherents to the Oxford Method claim that you never really learn something until you have to teach it — I think they're right. The students in my graduate course on testing are all full-time employees in local industries. Believe me, they keep you honest. This book is an outgrowth of my lectures and projects in that class.

I think of myself as a software engineer, but when I compare the level of precision and depth of knowledge prevalent in my field to those of more traditional engineering disciplines, I am uncomfortable with the term. A colleague and I were returning to our project in Italy when Myers' book, *The Art of Software Testing,* first came out. On the way to the airport, we stopped by the MIT bookstore and bought one of the early copies. In the intervening 15 years, I believe we have moved from an art to a craft. I had originally planned to title this book "The Craft of Software Testing," but as I neared the final chapters, another book with that title appeared. Maybe that's confirmation that software testing is becoming a craft. There's still a way to go before it is a science.

Part of any craft is knowing the capabilities and limitations of both the tools and the medium. A good woodworker has a variety of tools and, depending on the item being made and the wood being used, knows which tool is the most appropriate. Of all the phases of the traditional Waterfall Model of the software development life cycle, testing is the most amenable to precise analysis. Elevating software testing to a craft requires that the testing craftsperson know the basic tools. To this end, Chapters 3 and 4 provide mathematical background that is used freely in the remainder of the text.

Mathematics is a descriptive device that helps us better understand software to be tested. Precise notation, by itself, is not enough. We must also have good technique and judgment to identify appropriate testing methods and to apply them well. These are the goals of Parts II and III, which deal with fundamental functional and structural testing techniques. These techniques are applied to the continuing examples, which are described in Chapter 2. In Part IV, we apply these techniques to the integration and system levels of testing, and to object-oriented testing. At these levels, we are more concerned with what to test than how to test it, so the discussion moves toward requirements specification. Part IV concludes with an examination of testing interactions in a software controlled system, with a short discussion of client–server systems.

It is ironic that a book on testing contains faults. Despite the conscientious efforts of reviewers and editors, I am confident that faults persist in the text. Those that remain are my responsibility.

In 1977, I attended a testing seminar given by Edward Miller, who has since become one of the luminaries in software testing circles. In that seminar, Miller went to great lengths to convince us that testing need not be bothersome drudgery, but can be a very creative, interesting part of software development. My goal for you, the reader of this book, is that you will become a testing craftsperson, and that you will be able to derive the sense of pride and pleasure that a true craftsperson realizes from a job well done.

Paul C. Jorgensen
Rockford, Michigan
January 1995

Preface

Seven years have passed since I wrote the preface to the first edition. Much has happened in that time, hence this new edition. The most significant change is the dominance of the unified modeling language (UML) as a standard for the specification and design of object-oriented software. The main changes in this edition are the five chapters in Part V that deal with testing object-oriented software. Nearly all the material in Part V is UML-based.

The second major change is that the Pascal examples of the first edition are replaced by a language-neutral pseudocode. Most of the examples have been elaborated, and they are supported by Visual Basic executable modules available on the CRC Press Web site (www.crcpress.com). Several new examples illustrate some of the issues of testing object-oriented software. Dozens of other changes have been made; the most important additions include an improved description of equivalence class testing, a continuing case study, and more details about integration testing.

I am flattered that the first edition is one of the primary references on software testing in the trial-use standard Software Engineering Body of Knowledge jointly produced by the ACM and IEEE Computer Society (www.swebok.org). This recognition makes the problem of uncorrected mistakes more of a burden. A reader in South Korea sent me a list of 38 errors in the first edition, and students in my graduate class on software testing have gleefully contributed others. There is a nice analogy with testing here: I have fixed all the known errors, and my editor tells me it is time to stop looking for others. If you find any, please let me know — they are my responsibility. My e-mail address is: jorgensp@gvsu.edu.

I need to thank Jerry Papke and Helena Redshaw at CRC Press for their patience. I also want to thank my friend and colleague, Prof. Roger Ferguson, for his continued help with the new material in Part V, especially the continuing object-oriented calendar example. In a sense, Roger has been a tester of numerous drafts of Chapters 16 through 20.

Paul C. Jorgensen
Rockford, Michigan
May 2002

The Author

Paul C. Jorgensen, Ph.D., spent 20 years of his first career developing, supporting, and testing telephone switching systems. Since 1986, he has been teaching graduate courses in software engineering, first at Arizona State University, and then at Grand Valley State University. His consulting practice, Software Paradigms, hibernates during the Michigan winter when he is teaching and returns to life (along with everything else in Michigan) during the summer months.

Living and working for 3 years in Italy made him a confirmed "Italophile." He frequently travels to Italy with his wife, Carol, and daughters, Kirsten and Katia. In the summer, Paul sails his "Rebel" as often as possible. He is a year-round weight lifter. His e-mail address is jorgensp@gvsu.edu.

Contents

PART II FUNCTIONAL TESTING

PART III STRUCTURAL TESTING

PART IV INTEGRATION AND SYSTEM TESTING

A MATHEMATICAL CONTEXT

1

Chapter 1

A Perspective on Testing

Why do we test? The two main reasons are: to make a judgment about quality or acceptability and to discover problems. We test because we know that we are fallible — this is especially true in the domain of software and software-controlled systems. The goal of this chapter is to create a perspective (or context) on software testing. We will operate within this context for the remainder of the text.

1.1 Basic Definitions

Much of testing literature is mired in confusing (and sometimes inconsistent) terminology, probably because testing technology has evolved over decades and via scores of writers. The terminology here (and throughout this book) is taken from standards developed by the Institute of Electronics and Electrical Engineers (IEEE) Computer Society. To get started, let us look at a useful progression of terms.

Error — people make errors. A good synonym is mistake. When people make mistakes while coding, we call these mistakes bugs. Errors tend to propagate; a requirements error may be magnified during design and amplified still more during coding.

Fault — a fault is the result of an error. It is more precise to say that a fault is the representation of an error, where representation is the mode of expression, such as narrative text, dataflow diagrams, hierarchy charts, source code, and so on. Defect is a good synonym for fault, as is bug. Faults can be elusive. When a designer makes an error of omission, the resulting fault is that something is missing that should be present in the representation. This suggests a useful refinement; to borrow from the church, we might speak of faults of commission and faults of omission. A fault of commission occurs when we enter something into a representation that is incorrect. Faults of omission occur when we fail to enter correct information. Of these two types, faults of omission are more difficult to detect and resolve.

Failure — a failure occurs when a fault executes. Two subtleties arise here: one is that failures only occur in an executable representation, which is usually

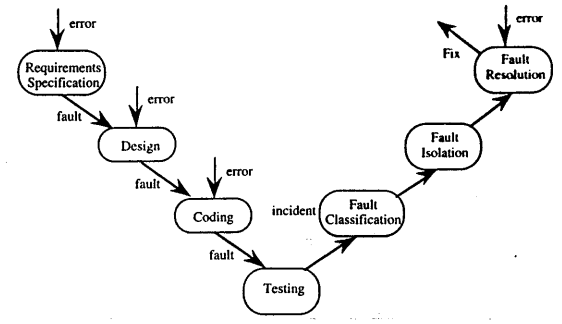

Figure 1.1 A testing life cycle.

taken to be source code, or more precisely, loaded object code; the second subtlety is that this definition relates failures only to faults of commission. How can we deal with failures that correspond to faults of omission? We can push this still further: What about faults that never happen to execute, or perhaps do not execute for a long time? The Michaelangelo virus is an example of such a fault. It does not execute until Michaelangelo's birthday, March 6. Reviews prevent many failures by finding faults; in fact, well-done reviews can find faults of omission.

Incident — when a failure occurs, it may or may not be readily apparent to the user (or customer or tester). An incident is the symptom associated with a failure that alerts the user to the occurrence of a failure.

Test — testing is obviously concerned with errors, faults, failures, and incidents. A test is the act of exercising software with test cases. A test has two distinct goals: to find failures or to demonstrate correct execution.

Test Case — test case has an identity and is associated with a program behavior. A test case also has a set of inputs and a list of expected outputs.

Figure 1.1 portrays a life cycle model for testing. Notice that, in the development phases, three opportunities arise for errors to be made, resulting in faults that propagate through the remainder of the development process. One prominent tester summarizes this life cycle as follows: the first three phases are Putting Bugs IN; the testing phase is Finding Bugs; and the last three phases are Getting Bugs OUT (Poston, 1990). The Fault Resolution step is another opportunity for errors (and new faults). When a fix causes formerly correct software to misbehave, the fix is deficient. We will revisit this when we discuss regression testing.

From this sequence of terms, we see that test cases occupy a central position in testing. The process of testing can be subdivided into separate steps: test planning, test case development, running test cases, and evaluating test results. The focus of this book is how to identify useful sets of test cases.

1.2 Test Cases

The essence of software testing is to determine a set of test cases for the item to be tested. Before going on, we need to clarify what information should be in a test case:

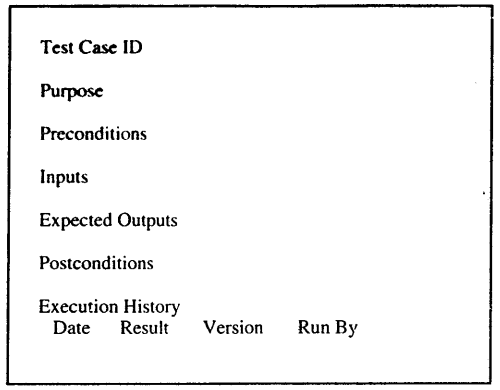

Figure 1.2 Typical test case information.

Inputs are really of two types: preconditions (circumstances that hold prior to test case execution) and the actual inputs that were identified by some testing method

Expected outputs, again of two types: postconditions and actual outputs

The output portion of a test case is frequently overlooked, which is unfortunate because this is often the hard part. Suppose, for example, you were testing software that determined an optimal route for an aircraft, given certain FAA air corridor constraints and the weather data for a flight day. How would you know what the optimal route really is? Various responses can address this problem. The academic response is to postulate the existence of an oracle who "knows all the answers." One industrial response to this problem is known as Reference Testing, where the system is tested in the presence of expert users. These experts make judgments as to whether outputs of an executed set of test case inputs are acceptable.

The act of testing entails establishing the necessary preconditions, providing the test case inputs, observing the outputs, and then comparing these with the expected outputs to determine whether the test passed.

The remaining information (Figure 1.2) in a well-developed test case primarily supports testing management. Test cases should have an identity and a reason for being (requirements tracing is a fine reason). It is also useful to record the execution history of a test case, including when and by whom it was run, the pass/fail result of each execution, and the version (of software) on which it was run. From all of this it becomes clear that test cases are valuable — at least as valuable as source code. Test cases need to be developed, reviewed, used, managed, and saved.

1.3 Insights from a Venn Diagram

Testing is fundamentally concerned with behavior, and behavior is orthogonal to the structural view common to software (and system) developers. A quick differentiation is that the structural view focuses on what it is and the behavioral view considers what it does. One of the continuing sources of difficulty for testers is

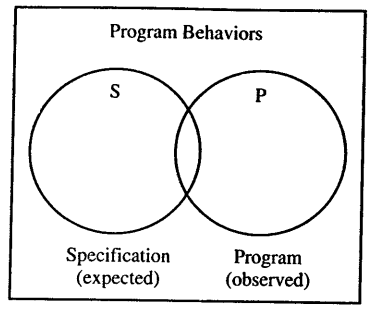

Figure 1.3 Specified and implemented program behaviors.

that the base documents are usually written by and for developers; the emphasis is therefore on structural, instead of behavioral, information. In this section, we develop a simple Venn diagram that clarifies several nagging questions about testing.

Consider a universe of program behaviors. (Notice that we are forcing attention on the essence of testing.) Given a program and its specification, consider the set S of specified behaviors, and the set P of programmed behaviors. Figure 1.3 shows the relationship among our universe of discourse as well as the specified and programmed behaviors. Of all the possible program behaviors, the specified ones are in the circle labeled S; and all those behaviors actually programmed (note the slight difference between P and U, the universe) are in P. With this diagram, we can see more clearly the problems that confront a tester. What if certain specified behaviors have not been programmed? In our earlier terminology, these are faults of omission. Similarly, what if certain programmed (implemented) behaviors have not been specified? These correspond to faults of commission and to errors that occurred after the specification was complete. The intersection of S and P (the football-shaped region) is the "correct" portion, that is, behaviors that are both specified and implemented. A very good view of testing is that it is the determination of the extent of program behavior that is both specified and implemented. (As an aside, note that "correctness" only has meaning with respect to a specification and an implementation. It is a relative term, not an absolute.)

The new circle in Figure 1.4 is for test cases. Notice the slight discrepancy with our universe of discourse and the set of program behaviors. Because a test case causes a program behavior, the mathematicians might forgive us. Now, consider the relationships among the sets S, P, and T. There may be specified behaviors that are not tested (regions 2 and 5), specified behaviors that are tested (regions 1 and 4), and test cases that correspond to unspecified behaviors (regions 3 and 7).

Similarly, there may be programmed behaviors that are not tested (regions 2 and 6), programmed behaviors that are tested (regions 1 and 3), and test cases that correspond to unprogrammed behaviors (regions 4 and 7). Each of these regions is important. If specified behaviors exist for which no test cases are available, the testing is necessarily incomplete. If certain test cases correspond to unspecified behaviors, two possibilities arise: either such a test case is unwarranted, or the specification is deficient. (In my experience, good testers often postulate test cases of this latter type. This is a fine reason to have good testers participate in specification and design reviews.)

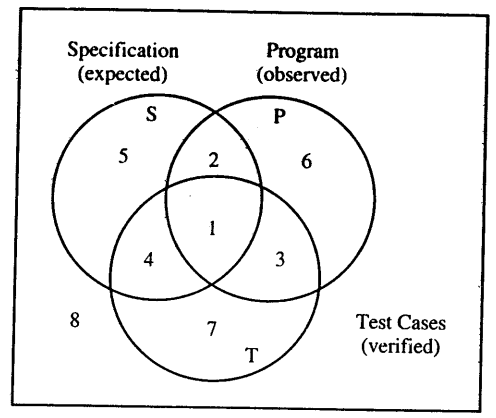

Figure 1.4 Specified, implemented, and tested behaviors.

We are already at a point where we can see some possibilities for testing as a craft: what can a tester do to make the region where these sets all intersect (region 1) as large as possible? Another approach is to ask how the test cases in the set T are identified. The short answer is that test cases are identified by a testing method. This framework gives us a way to compare the effectiveness of diverse testing methods, as we shall see in Chapters 8 and 11.

1.4 Identifying Test Cases

Two fundamental approaches are used to identify test cases, known as functional and structural testing. Each of these approaches has several distinct test case identification methods, more commonly called testing methods.

1.4.1 Functional Testing

Functional testing is based on the view that any program can be considered to be a function that maps values from its input domain to values in its output range. (Function, domain, and range are defined in Chapter 3.) This notion is commonly used in engineering, when systems are considered to be black boxes. This leads to the term black box testing, in which the content (implementation) of a black box is not known, and the function of the black box is understood completely in terms of its inputs and outputs (see Figure 1.5). In *Zen and the Art of Motorcycle Maintenance,* Pirsig refers to this as "romantic" comprehension (Pirsig, 1973). Many times, we operate very effectively with black box knowledge; in fact, this is central to object orientation. As an example, most people successfully operate automobiles with only black box knowledge.

With the functional approach to test case identification, the only information used is the specification of the software. Functional test cases have two distinct advantages: (1) they are independent of how the software is implemented, so if

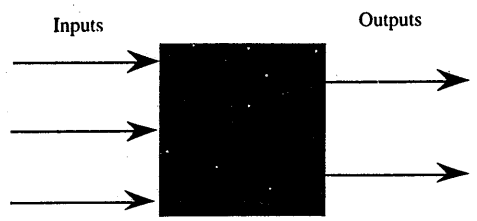

Figure 1.5 An engineer's black box.

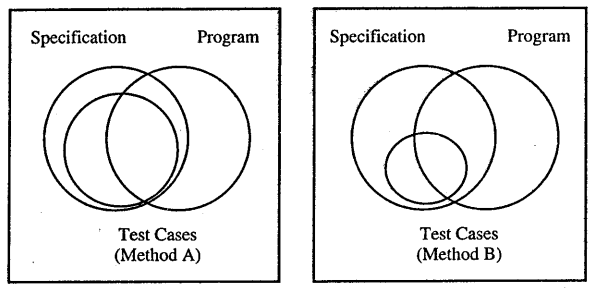

Figure 1.6 Comparing functional test case identification methods.

the implementation changes, the test cases are still useful; and (2) test case development can occur in parallel with the implementation, thereby reducing overall project development interval. On the negative side, functional test cases frequently suffer from two problems: significant redundancies may exist among test cases, compounded by the possibility of gaps of untested software.

Figure 1.6 shows the results of test cases identified by two functional methods. Method A identifies a larger set of test cases than does Method B. Notice that, for both methods, the set of test cases is completely contained within the set of specified behavior. Because functional methods are based on the specified behavior, it is hard to imagine these methods identifying behaviors that are not specified. In Chapter 8, we will see direct comparisons of test cases generated by various functional methods for the examples defined in Chapter 2.

In Part II, we will examine the mainline approaches to functional testing, including boundary value analysis, robustness testing, worst-case analysis, special value testing, input (domain) equivalence classes, output (range) equivalence classes, and decision table-based testing. The common thread running through these techniques is that all are based on definitional information of the item tested. The mathematical background presented in Chapter 3 applies primarily to the functional approaches.

1.4.2 Structural Testing

Structural testing is the other fundamental approach to test case identification. To contrast it with functional testing, it is sometimes called white box (or even clear box) testing. The clear box metaphor is probably more appropriate, because the

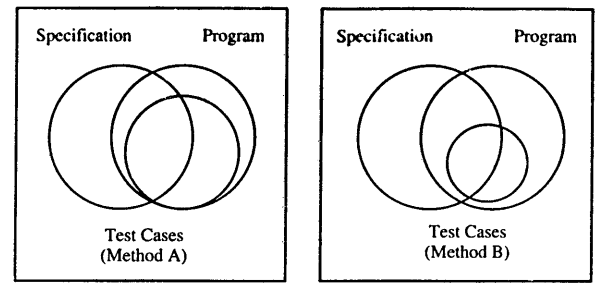

Figure 1.7 Comparing structural test case identification methods.

essential difference is that the implementation (of the black box) is known and used to identify test cases. The ability to "see inside" the black box allows the tester to identify test cases based on how the function is actually implemented.

Structural testing has been the subject of some fairly strong theory. To really understand structural testing, familiarity with the concepts of linear graph theory (Chapter 4) is essential. With these concepts, the tester can rigorously describe exactly what is tested. Because of its strong theoretical basis, structural testing lends itself to the definition and use of test coverage metrics. Test coverage metrics provide a way to explicitly state the extent to which a software item has been tested, and this in turn makes testing management more meaningful.

Figure 1.7 shows the results of test cases identified by two structural methods. As before, Method A identifies a larger set of test cases than does Method B. Is a larger set of test cases necessarily better? This is an excellent question, and structural testing provides important ways to develop an answer. Notice that, for both methods, the set of test cases is completely contained within the set of programmed behavior. Because structural methods are based on the program, it is hard to imagine these methods identifying behaviors that are not programmed. It is easy to imagine, however, that a set of structural test cases is relatively small with respect to the full set of programmed behaviors. At the end of Part III, we will see direct comparisons of test cases generated by various structural methods.

1.4.3 The Functional versus Structural Debate

Given two fundamentally different approaches to test case identification, it is natural to question which is better. If you read much of the literature, you will find strong adherents to either choice. Referring to structural testing, Robert Poston writes, "This tool has been wasting tester's time since the 1970s ... [it] does not support good software testing practice and should not be in the tester's toolkit" (Poston, 1991). In defense of structural testing, Edward Miller writes, "Branch coverage [a structural test coverage metric], if attained at the 85% or better level, tends to identify twice the number of defects that would have been found by 'intuitive' [functional] testing" (Miller, 1991).

The Venn diagrams presented earlier yield a strong resolution to this debate. Recall that the goal of both approaches is to identify test cases. Functional testing uses only the specification to identify test cases, while structural testing uses the

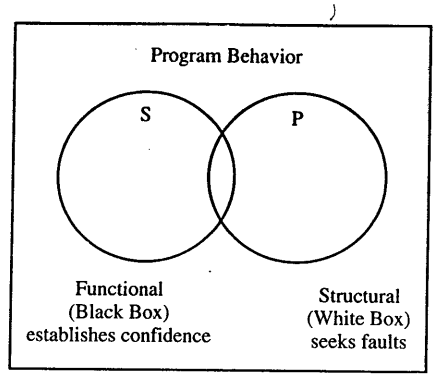

Figure 1.8 Sources of test cases.

program source code (implementation) as the basis of test case identification. Our earlier discussion forces the conclusion that neither approach alone is sufficient. Consider program behaviors: if all specified behaviors have not been implemented, structural test cases will never be able to recognize this. Conversely, if the program implements behaviors that have not been specified, this will never be revealed by functional test cases. (A virus is a good example of such unspecified behavior.) The quick answer is that both approaches are needed; the testing craftsperson's answer is that a judicious combination will provide the confidence of functional testing and the measurement of structured testing. Earlier, we asserted that functional testing often suffers from twin problems of redundancies and gaps. When functional test cases are executed in combination with structural test coverage metrics, both of these problems can be recognized and resolved (see Figure 1.8).

The Venn diagram view of testing provides one final insight. What is the relationship between the set T of test cases and the sets S and P of specified and implemented behaviors? Clearly, the test cases in T are determined by the test case identification method used. A very good question to ask is how appropriate (or effective) is this method? To close a loop from an earlier discussion, recall the causal trail from error to fault, failure, and incident. If we know what kind of errors we are prone to make, and if we know what kinds of faults are likely to reside in the software to be tested, we can use this to employ more appropriate test case identification methods. This is the point at which testing really becomes a craft.

1.5 Error and Fault Taxonomies

Our definitions of error and fault hinge on the distinction between process and product: process refers to how we do something, and product is the end result of a process. The point at which testing and Software Quality Assurance (SQA) meet is that SQA typically tries to improve the product by improving the process. In that sense, testing is clearly more product oriented. SQA is more concerned with reducing errors endemic in the development process, while testing is more concerned with discovering faults in a product. Both disciplines benefit from a clearer definition of types of faults. Faults can be classified in several ways: the

1. Mild	Misspelled word
2. Moderate	Misleading or redundant information
3. Annoying	Truncated names, bill for $0.00
4. Disturbing	Some transaction(s) not processed
5. Serious	Lose a transaction
6. Very serious	Incorrect transaction execution
7. Extreme	Frequent "very serious" errors
8. Intolerable	Database corruption
9. Catastrophic	System shutdown
10. Infectious	Shutdown that spreads to others

Figure 1.9 Faults classified by severity.

development phase in which the corresponding error occurred, the consequences of corresponding failures, difficulty to resolve, risk of no resolution, and so on. My favorite is based on anomaly occurrence: one time only, intermittent, recurring, or repeatable. Figure 1.9 contains a fault taxonomy (Beizer, 1984) that distinguishes faults by the severity of their consequences.

For a comprehensive treatment of types of faults, see the IEEE Standard Classification for Software Anomalies (IEEE, 1993). (A software anomaly is defined in that document as "a departure from the expected," which is pretty close to our definition.) The IEEE standard defines a detailed anomaly resolution process built around four phases (another life cycle): recognition, investigation, action, and disposition. Some of the more useful anomalies are given in Tables 1.1 through 1.5; most of these are from the IEEE standard, but I have added some of my favorites.

Table 1.1 Input/Output Faults

Type	*Instances*
Input	Correct input not accepted
	Incorrect input accepted
	Description wrong or missing
	Parameters wrong or missing
Output	Wrong format
	Wrong result
	Correct result at wrong time (too early, too late)
	Incomplete or missing result
	Spurious result
	Spelling/grammar
	Cosmetic

Table 1.2 Logic Faults

Missing case(s)
Duplicate case(s)
Extreme condition neglected
Misinterpretation
Missing condition
Extraneous condition(s)
Test of wrong variable
Incorrect loop iteration
Wrong operator (e.g., < instead of ≤)

Table 1.3 Computation Faults

Incorrect algorithm
Missing computation
Incorrect operand
Incorrect operation
Parenthesis error
Insufficient precision (round-off, truncation)
Wrong built-in function

Table 1.4 Interface Faults

Incorrect interrupt handling
I/O timing
Call to wrong procedure
Call to nonexistent procedure
Parameter mismatch (type, number)
Incompatible types
Superfluous inclusion

Table 1.5 Data Faults

Incorrect initialization
Incorrect storage/access
Wrong flag/index value
Incorrect packing/unpacking
Wrong variable used
Wrong data reference
Scaling or units error
Incorrect data dimension
Incorrect subscript
Incorrect type
Incorrect data scope
Sensor data out of limits
Off by one
Inconsistent data

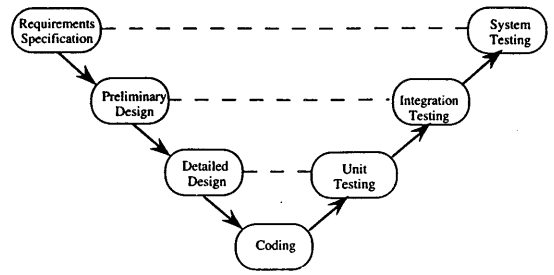

Figure 1.10 Levels of abstraction and testing in the Waterfall Model.

1.6 Levels of Testing

Thus far, we have said nothing about one of the key concepts of testing — levels of abstraction. Levels of testing echo the levels of abstraction found in the Waterfall Model of the software development life cycle. Although this model has its drawbacks, it is useful for testing as a means of identifying distinct levels of testing and for clarifying the objectives that pertain to each level. A diagrammatic variation of the Waterfall Model is given in Figure 1.10; this variation emphasizes the correspondence between testing and design levels. Notice that, especially in terms of functional testing, the three levels of definition (specification, preliminary design, and detailed design) correspond directly to three levels of testing — unit, integration, and system testing.

A practical relationship exists between levels of testing versus functional and structural testing. Most practitioners agree that structural testing is most appropriate at the unit level, while functional testing is most appropriate at the system level. This is generally true, but it is also a likely consequence of the base information produced during the requirements specification, preliminary design, and detailed design phases. The constructs defined for structural testing make the most sense at the unit level, and similar constructs are only now becoming available for the integration and system levels of testing. We develop such structures in Part IV to support structural testing at the integration and system levels for both traditional and object-oriented software.

References

Beizer, Boris, *Software System Testing and Quality Assurance,* Van Nostrand Reinhold, New York, 1984.

IEEE Computer Society, *IEEE Standard Glossary of Software Engineering Terminology,* 1983, ANSI/IEEE Std 729-1983.

IEEE Computer Society, *IEEE Standard Classification for Software Anomalies,* 1993, IEEE Std 1044-1993.

Miller, Edward F. Jr., Automated software testing: A technical perspective, *American Programmer,* April 1991, 4:4, 38–43.

Pirsig, Robert M., *Zen and the Art of Motorcycle Maintenance,* Bantam Books, New York, 1973.

Poston, Robert M., *T: Automated Software Testing Workshop*, Programming Environments, Inc., Tinton Falls, NJ, 1990.
Poston, Robert M., A complete toolkit for the software tester, *American Programmer*, April 1991, 4:4, 28–37. Reprinted in CrossTalk, a USAF publication.

Exercises

1. Make a Venn Diagram that reflects a part of the following statement: " ... we have left undone that which we ought to have done, and we have done that which we ought not to have done ..."
2. Describe each of the eight regions in Figure 1.4. Can you recall examples of these in software you have written?
3. One of the folk tales of software lore describes a disgruntled employee who writes a payroll program. The program contains logic that checks for the employee's identification number before producing paychecks. If the employee is ever terminated, the program creates havoc. Discuss this situation in terms of the error, fault, and failure pattern, and decide which form of testing would be appropriate.

Chapter 2

Examples

Three examples will be used throughout Parts II and III to illustrate the various unit testing methods. They are: the triangle problem (a venerable example in testing circles); a logically complex function, NextDate; and an example that typifies MIS testing, known here as the commission problem. Taken together, these examples raise most of the issues that testing craftspersons will encounter at the unit level. The discussion of integration and system testing in Part IV uses three other examples: a simplified version of an automated teller machine (ATM), known here as the simple ATM system (SATM); the currency converter, an event-driven application typical of graphical user interface (GUI) applications; and the windshield wiper control device from the Saturn automobile. Finally, an object-oriented version of NextDate is provided, called o-oCalendar, which is used to illustrate aspects of testing object-oriented software in Part V.

For the purposes of structural testing, pseudocode implementations of the three unit-level examples are given in this chapter. System-level descriptions of the SATM system, the currency cconverter, and the Saturn windshield wiper system are given in Part IV. These applications are described both traditionally (with E/R diagrams, data flow diagrams, and finite state machines) and with the de facto object-oriented standard, the universal modeling language (UML) in Part V.

2.1 Generalized Pseudocode

Generalized pseudocode provides a "language neutral" way to express program source code. It has constructs at two levels: unit and program components. Units can be interpreted either as traditional components (procedures and functions) or as object-oriented components (classes and objects). This definition is somewhat informal; terms such as expression, variable list, and field desription are used with no formal definition. Items in angle brackets indicate language elements that can be used at the identified positions. Part of the value of any pseudocode is the supression of unwanted detail; here, we illustrate this by allowing natural language phrases in place of more formal, complex conditions. (See Table 2.1.)

Table 2.1 Generalized Pseudocode

Language Element	*Generalized Pseudocode Construct*
Comment	' <text>
Data structure declaration	Type <type name> <list of field descriptions> End <type name>
Data declaration	Dim <variable> As <type>
Assignment statement	<variable> = <expression>
Input	Input (<variable list>)
Output	Output (<variable list>)
Simple condition	<expression> <relational operator> <expression>
Compound condition	<simple condition> <logical connective> <Simple condition>
Sequence	statements in sequential order

Simple selection	If <condition> Then
	<then clause>
	EndIf

Selection	If <condition> Then <then clause> Else <else clause> EndIf
Multiple selection	Case <variable> Of Case 1: <predicate> <Case clause> ... Case n: <predicate> <Case clause> EndCase
Counter-controlled repetition	For <counter> = <start> To <end> <loop body> EndFor
Pretest repetition	Do While <condition> <loop body> EndWhile
Posttest repetition	Do <loop body> Until <condition>
Procedure definition (similarly for functions and o-o methods)	<procedure name> (Input: <variable list>; Output: <variable list>) <body> End <procedure name>
Interunit communication	Call <procedure name> (<variable list>; <variable list>)
Class/Object definition	<name> (<attribute list>; <method list>, <body> End <name>
Interunit communication	msg <destination object name>.<method name> (<variable list>)
Object creation	Instantiate <class name>.<object name> (attribute values)

Table 2.1 Generalized Pseudocode *(Continued)*

Language Element	*Generalized Pseudocode Construct*
Object destruction	Delete <class name>.<object name>
Program	Program <program name> <unit list> End<program name>

2.2 The Triangle Problem

The triangle problem is the most widely used example in software testing literature. Some of the more notable entries in three decades of testing literature are Gruenberger (1973); Brown (1975); Myers (1979); Pressman (1982) and its 2nd, 3rd, 4th, and 5th editions; Clarke (1983); Clarke (1984); Chellappa (1987); and Hetzel (1988). There are others, but this list makes the point.

2.2.1 Problem Statements

Simple version: The triangle program accepts three integers, a, b, and c as input. These are taken to be sides of a triangle. The output of the program is the type of triangle determined by the three sides: Equilateral, Isosceles, Scalene, or NotATriangle. Sometimes this problem is extended to include right triangles as a fifth type; we will use this extension in some of the exercises.

We can improve this definition by providing more detail. Doing so, the problem statement follows next.

Improved version: The triangle program accepts three integers, a, b, and c, as input. These are taken to be sides of a triangle. The integers a, b, and c must satisfy the following conditions:

c1. $1 \leq a \leq 200$
c2. $1 \leq b \leq 200$
c3. $1 \leq c \leq 200$
c4. $a < b + c$
c5. $b < a + c$
c6. $c < a + b$

The output of the program is the type of triangle determined by the three sides: Equilateral, Isosceles, Scalene, or NotATriangle. If an input value fails any of these conditions, the program notes this with an output message, for example, "Value of b is not in the range of permitted values." If values of a, b, and c satisfy conditions c1, c2, and c3, one of four mutually exclusive outputs is given:

1. If all three sides are equal, the program output is Equilateral.
2. If exactly one pair of sides is equal, the program output is Isosceles.
3. If no pair of sides is equal, the program output is Scalene.
4. If any of conditions c4, c5, and c6 fails, the program output is NotATriangle.

2.2.2 Discussion

Perhaps one of the reasons for the longevity of this example is that it contains clear but complex logic. It also typifies some of the incomplete definitions that

impair communication among customers, developers, and testers. The first specification presumes the developers know some details about triangles, particularly the Triangle Property: the sum of any pair of sides must be strictly greater than the third side. The upper limit of 200 is both arbitrary and convenient; it will be used when we develop boundary value test cases in Chapter 5.

2.2.3 Traditional Implementation

The "traditional" implementation of this grandfather of all examples has a rather FORTRAN-like style. The flowchart for this implementation appears in Figure 2.1. The flowchart box numbers correspond to comment numbers in the (FORTRAN-like) pseudocode program given next. (These numbers correspond exactly to those in Pressman [1982].) I do not really like this implementation very much, so a more structured implementation is given in Section 2.2.4.

The variable match is used to record equality among pairs of the sides. A classical intricacy of the FORTRAN style is connected with the variable match: notice that all three tests for the triangle property do not occur. If two sides are

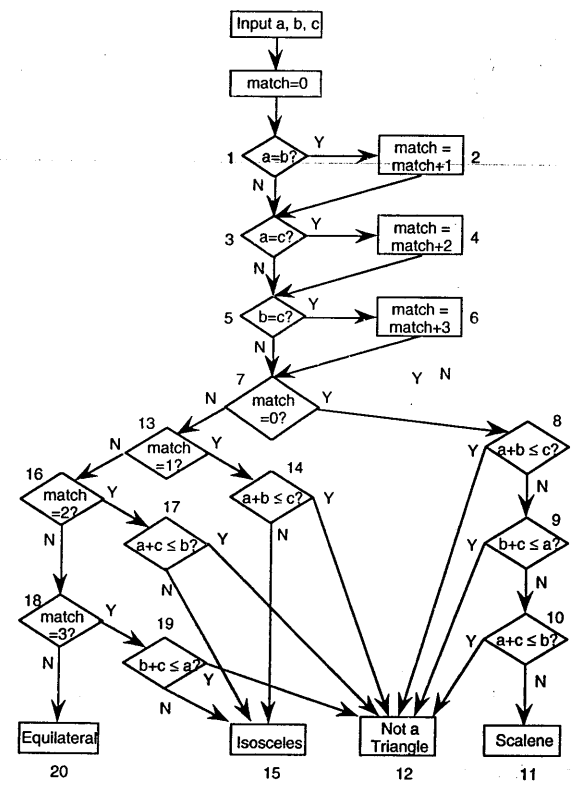

Figure 2.1 Flowchart for the traditional triangle program implementation.

equal, say a and c, it is only necessary to compare a + c with b. (Because b must be greater than zero, a + b must be greater than c, because c equals a.) This observation clearly reduces the number of comparisons that must be made. The efficiency of this version is obtained at the expense of clarity (and ease of testing). We will find this version useful later in Part III when we discuss infeasible program execution paths. That is the only reason for perpetuating this version.

```
Program triangle1 'Fortran-like version
'
Dim a,b,c,match As INTEGER
'
Output("Enter 3 integers which are sides of a triangle")
Input(a,b,c)
Output("Side A is ",a)
Output("Side B is ",b)
Output("Side C is ",c)
match = 0
If a = b                                                    '(1)
   Then match = match + 1                                   '(2)
EndIf
If a = c                                                    '(3)
   Then match = match + 2                                   '(4)
EndIf
If b = c                                                    '(5)
   Then match = match + 3                                   '(6)
EndIf
If match = 0                                                '(7)
   Then If (a+b)<=c                                         '(8)
           Then Output("NotATriangle")                      '(12.1)
           Else If (b+c)<=a                                 '(9)
                   Then Output("NotATriangle")              '(12.2)
                   Else If (a+c)<=b                         '(10)
                           Then Output("NotATriangle")      '(12.3)
                           Else Output ("Scalene")          '(11)
                        EndIf
                EndIf
        EndIf
   Else If match=1                                          '(13)
           Then If (a+c)<=b                                 '(14)
                   Then Output("NotATriangle")              '(12.4)
                   Else Output ("Isosceles")                '(15.1)
                EndIf
           Else If match=2                                  '(16)
                   Then If (a+c)<=b
                           Then Output("NotATriangle")      '(12.5)
                           Else Output ("Isosceles")        '(15.2)
                        EndIf
                   Else If match=3                          '(18)
                           Then If (b+c)<=a                 '(19)
                                   Then Output("NotATriangle") '(12.6)
                                   Else Output ("Isosceles")   '(15.3)
                                EndIf
                           Else Output ("Equilateral")      '(20)
                                EndIf
                        EndIf
        EndIf
EndIf
'
End Triangle1
```

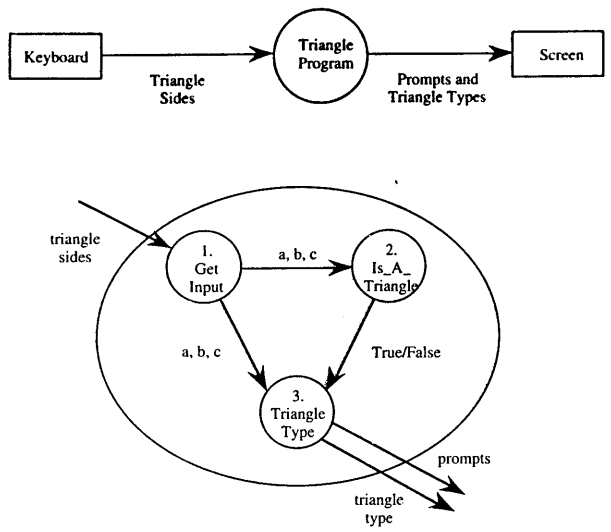

Figure 2.2 Dataflow diagram for a structured triangle program implementation.

Notice that six ways are used to reach the NotATriangle box (12.1–12.6), and three ways are used to reach the Isosceles box (15.1–15.3).

2.2.4 Structured Implementation

Figure 2.2 is a dataflow diagram description of the triangle program. We could implement it as a main program with the three indicated procedures. We will use this example later for unit testing, therefore, the three procedures have been merged into one pseudocode program. Comment lines relate sections of the code to the decomposition given in Figure 2.2.

```
Program triangle2 'Structured programming version of simpler specification
'
Dim a,b,c As Integer
Dim IsATriangle As Boolean
'
'Step 1: Get Input
Output("Enter 3 integers which are sides of a triangle")
Input(a,b,c)
Output("Side A is ",a)
Output("Side B is ",b)
Output("Side C is ",c)
'
'Step 2: Is A Triangle?
If (a < b + c) AND (b < a + c) AND (c < a + b)
   Then IsATriangle = True
   Else IsATriangle = False
EndIf
'
```



```
'Step 3: Determine Triangle Type
If IsATriangle
   Then   If (a = b) AND (b = c)
             Then Output ("Equilateral")
             Else   If (a ≠ b) AND (a ≠ c) AND (b ≠ c)
                       Then   Output ("Scalene")
                       Else   Output ("Isosceles")
                    EndIf
          EndIf
   Else   Output("Not a Triangle")
EndIf
'
End triangle2


Program triangle3 'Structured programming version of improved specification
'
Dim a,b,c As Integer
Dim c1, c2, c3, IsATriangle As Boolean
'
'Step 1: Get Input
Do
   Output("Enter 3 integers which are sides of a triangle")
   Input(a,b,c)
   c1 = (1 <= a) AND (a <= 200)
   c2 = (1 <= b) AND (b <= 200)
   c3 = (1 <= c) AND (c <= 200)
   If NOT(c1)
      Then   Output("Value of a is not in the range of permitted values")
   EndIf
   If NOT(c2)
      Then   Output("Value of b is not in the range of permitted values")
   EndIf
   If NOT(c3)
      Then   Output("Value of c is not in the range of permitted values")
   EndIf
Until c1 AND c2 AND c3
Output("Side A is ",a)
Output("Side B is ",b)
Output("Side C is ",c)
'
'Step 2: Is A Triangle?
If (a < (b + c)) AND (b < (a + c)) AND (c < (a + b))
   Then IsATriangle = True
   Else IsATriangle = False
EndIf
'
'Step 3: Determine Triangle Type
If IsATriangle
   Then   If (a = b) AND (b = c)
             Then Output ("Equilateral")
             Else   If (a ≠ b) AND (a ≠ c) AND (b ≠ c)
                       Then   Output ("Scalene")
                       Else   Output ("Isosceles")
                    EndIf
          EndIf
   Else   Output("Not a Triangle")
EndIf
'
End triangle3
```

2.3 The NextDate Function

The complexity in the triangle program is due to relationships between inputs and correct outputs. We will use the NextDate function to illustrate a different kind of complexity — logical relationships among the input variables.

2.3.1 Problem Statements

NextDate is a function of three variables: month, date, and year. It returns the date of the day after the input date. The month, date, and year variables have integer values subject to these conditions:

c1. $1 \leq month \leq 12$
c2. $1 \leq day \leq 31$
c3. $1812 \leq year \leq 2012$

As we did with the triangle program, we can make our specification more specific. This entails defining responses for invalid values of the input values for the day, month, and year. We can also define responses for invalid logical combinations, such as June 31 of any year. If any of conditions c1, c2, or c3 fails, NextDate produces an output indicating the corresponding variable has an out-of-range value — for example, "Value of month not in the range 1..12". Because numerous invalid day–month–year combinations exist, NextDate collapses these into one message: "Invalid Input Date."

2.3.2 Discussion

Two sources of complexity exist in the NextDate function: the complexity of the input domain discussed previously, and the rule that determines when a year is a leap year. A year is 365.2422 days long, therefore, leap years are used for the "extra day" problem. If we declared a leap year every fourth year, a slight error would occur. The Gregorian Calendar (after Pope Gregory) resolves this by adjusting leap years on century years. Thus, a year is a leap year if it is divisible by 4, unless it is a century year. Century years are leap years only if they are multiples of 400 (Inglis, 1961); so 1992, 1996, and 2000 are leap years, while the year 1900 is not a leap year. The NextDate function also illustrates a sidelight of software testing. Many times, we find examples of Zipf's Law, which states that 80% of the activity occurs in 20% of the space. Notice how much of the source code is devoted to leap year considerations. In the second implementation, notice how much code is devoted to input value validation.

2.3.3 Implementation

```
Program NextDate1        'Simple version
'
Dim tomorrowDay,tomorrowMonth,tomorrowYear As Integer
Dim day,month,year As Integer
'
```



```
Output ("Enter today's date in the form MM DD YYYY")
Input (month,day,year)
Case month Of
Case 1: month Is 1,3,5,7,8, Or 10: '31 day months (except Dec.)
   If day < 31
      Then tomorrowDay = day + 1
      Else
         tomorrowDay = 1
         tomorrowMonth = month + 1
   EndIf
Case 2: month Is 4,6,9, Or 11 '30 day months
   If day < 30
      Then tomorrowDay = day + 1
      Else
         tomorrowDay = 1
         tomorrowMonth = month + 1
   EndIf
Case 3: month Is 12: 'December
   If day < 31
      Then tomorrowDay = day + 1
      Else
         tomorrowDay = 1
         tomorrowMonth = 1
         If year = 2012
            Then Output ("2012 is over")
            Else tomorrow.year = year + 1
   EndIf
Case 4: month is 2: 'February
   If day < 28
      Then tomorrowDay = day + 1
      Else
         If day = 28
            Then
               If ((year is a leap year)
                  Then tomorrowDay = 29 'leap year
                  Else          'not a leap year
                     tomorrowDay = 1
                     tomorrowMonth = 3
               EndIf
            Else    If day = 29
                        Then tomorrowDay = 1
                           tomorrowMonth = 3
                        Else     Output("Cannot have Feb.", day)
                     EndIf
         EndIf
   EndIf
EndCase
Output ("Tomorrow's date is", tomorrowMonth, tomorrowDay, tomorrowYear)
'
End NextDate


Program NextDate2        Improved version
'
Dim tomorrowDay,tomorrowMonth,tomorrowYear As Integer
Dim day,month,year As Integer
Dim c1, c2, c3 As Boolean
'
Do
   Output ("Enter today's date in the form MM DD YYYY")
```

```
    Input (month,day,year)
    c1 = (1 <= day) AND (day <= 31)
    c2 = (1 <= month) AND (month <= 12)
    c3 = (1812 <= year) AND (year <= 2012)
    If NOT(c1)
        Then    Output("Value of day not in the range 1..31")
    EndIf
    If NOT(c2)
        Then    Output("Value of month not in the range 1..12")
    EndIf
    If NOT(c3)
        Then    Output("Value of year not in the range 1812..2012")
    EndIf
Until c1 AND c2 AND c2

Case month Of
Case 1: month Is 1,3,5,7,8, Or 10: '31 day months (except Dec.)
    If day < 31
        Then tomorrowDay = day + 1
        Else
            tomorrowDay = 1
            tomorrowMonth = month + 1
    EndIf
Case 2: month Is 4,6,9, Or 11 '30 day months
    If day < 30
        Then tomorrowDay = day + 1
        Else
            If day = 30
                Then    tomorrowDay = 1
                        tomorrowMonth = month + 1
                Else    Output("Invalid Input Date")
            EndIf
    EndIf
Case 3: month Is 12: 'December
    If day < 31
        Then tomorrowDay = day + 1
        Else
            tomorrowDay = 1
            tomorrowMonth = 1
            If year = 2012
                Then Output ("Invalid Input Date")
                Else tomorrow.year = year + 1
    EndIf
Case 4: month is 2: 'February
    If day < 28
        Then tomorrowDay = day + 1
        Else
            If day = 28
                Then
                    If (year is a leap year)
                        Then tomorrowDay = 29 'leap day
                        Else          'not a leap year
                            tomorrowDay = 1
                            tomorrowMonth = 3
                    EndIf
                Else
                    If day = 29
                        Then
                            If (year is a leap year)
                                Then    tomorrowDay = 1
                                        tomorrowMonth = 3
                                Else
                                    If day > 29
                                        Then    Output("Invalid Input Date")
                                    EndIf
                                EndIf
                        EndIf
                EndIf
    EndIf
EndCase
Output ("Tomorrow's date is", tomorrowMonth, tomorrowDay, tomorrowYear)
'
End NextDate2
```


2.4 The Commission Problem

Our third example is more typical of commercial computing. It contains a mix of computation and decision making, so it leads to interesting testing questions.

2.4.1 Problem Statement

A rifle salesperson in the former Arizona Territory sold rifle locks, stocks, and barrels made by a gunsmith in Missouri. Locks cost $45, stocks cost $30, and barrels cost $25. The salesperson had to sell at least one complete rifle per month, and production limits were such that the most the salesperson could sell in a month was 70 locks, 80 stocks, and 90 barrels. After each town visit, the salesperson sent a telegram to the Missouri gunsmith with the number of locks, stocks, and barrels sold in that town. At the end of a month, the salesperson sent a very short telegram showing –1 locks sold. The gunsmith then knew the sales for the month were complete and computed the salesperson's commission as follows: 10% on sales up to (and including) $1000, 15% on the next $800, and 20% on any sales in excess of $1800. The commission program produced a monthly sales report that gave the total number of locks, stocks, and barrels sold, the salesperson's total dollar sales, and, finally, the commission.

2.4.2 Discussion

This example is somewhat contrived to make the arithmetic quickly visible to the reader. It might be more realistic to consider some other additive function of several variables, such as various calculations found in filling out a U.S. 1040 income tax form. (We will stay with rifles.) This problem separates into three distinct pieces: the input data portion, in which we could deal with input data validation (as we did for the triangle and NextDate programs), the sales calculation, and the commission calculation portion. This time, we will omit the input data validation portion. We will replicate the telegram convention with a sentinel-controlled While loop that is typical of MIS data gathering applications.

2.4.3 Implementation

```
Program Commission (INPUT,OUTPUT)
'
Dim locks, stocks, barrels As Integer
Dim lockPrice, stockPrice, barrelPrice As Real
Dim totalLocks,totalStocks,totalBarrels As Integer
Dim lockSales, stockSales, barrelSales As Real
Dim sales,commission : REAL
'
lockPrice  = 45.0
stockPrice  = 30.0
barrelPrice = 25.0
totalLocks = 0
totalStocks = 0
totalBarrels = 0
'
Input(locks)
While NOT(locks = -1)     'Input device uses -1 to indicate end of data
    Input(stocks, barrels)
    totalLocks = totalLocks + locks
    totalStocks = totalStocks + stocks
    totalBarrels = totalBarrels + barrels
    Input(locks)
EndWhile
'
Output("Locks sold: ", totalLocks)
Output("Stocks sold: ", totalStocks)
Output("Barrels sold: ", totalBarrels)
'
lockSales = lockPrice*totalLocks
stockSales = stockPrice*totalStocks
barrelSales = barrelPrice * totalBarrels
sales = lockSales + stockSales + barrelSales
Output("Total sales: ", sales)
'
If (sales > 1800.0)
    Then
        commission = 0.10 * 1000.0
        commission = commission + 0.15 * 800.0
        commission = commission + 0.20*(sales-1800.0)
    Else If (sales > 1000.0)
                Then
                    commission = 0.10 * 1000.0
                    commission = commission + 0.15*(sales-1000.0)
                Else commission = 0.10 * sales
            EndIf
EndIf
Output("Commission is $",commission)
'
End Commission
```

2.5 The SATM System

To better discuss the issues of integration and system testing, we need an example with larger scope. The automated teller machine described here is a refinement of that in Topper (1993); it contains an interesting variety of functionality and interactions that typify the client side of client–server systems.

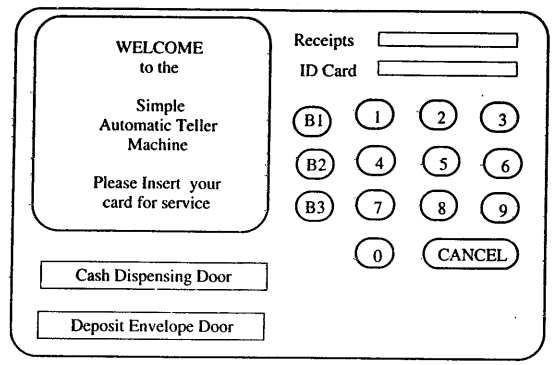

Figure 2.3 The SATM terminal.

2.5.1 Problem Statement

The SATM system communicates with bank customers via the 15 screens shown in Figure 2.4. Using a terminal with features as shown in Figure 2.3, SATM customers can select any of three transaction types: deposits, withdrawals, and balance inquiries. These transactions can be done on two types of accounts: checking and savings.

When a bank customer arrives at an SATM station, screen 1 is displayed. The bank customer accesses the SATM system with a plastic card encoded with a personal account number (PAN), which is a key to an internal customer account file, containing, among other things, the customer's name and account information. If the customer's PAN matches the information in the customer account file, the system presents screen 2 to the customer. If the customer's PAN is not found, screen 4 is displayed, and the card is kept.

At screen 2, the customer is prompted to enter his or her personal identification number (PIN). If the PIN is correct (i.e., matches the information in the customer account file), the system displays screen 5; otherwise, screen 3 is displayed. The customer has three chances to get the PIN correct; after three failures, screen 4 is displayed, and the card is kept.

On entry to screen 5, the system adds two pieces of information to the customer's account file: the current date and an increment to the number of ATM sessions. The customer selects the desired transaction from the options shown on screen 5; then the system immediately displays screen 6, where the customer chooses the account to which the selected transaction will be applied.

If balance is requested, the system checks the local ATM file for any unposted transactions and reconciles these with the beginning balance for that day from the customer account file. Screen 14 is then displayed.

If deposit is requested, the status of the deposit envelope slot is determined from a field in the terminal control file. If no problem is known, the system displays screen 7 to get the transaction amount. If a problem occurs with the deposit envelope slot, the system displays screen 12. Once the deposit amount has been entered, the system displays screen 13, accepts the deposit envelope,

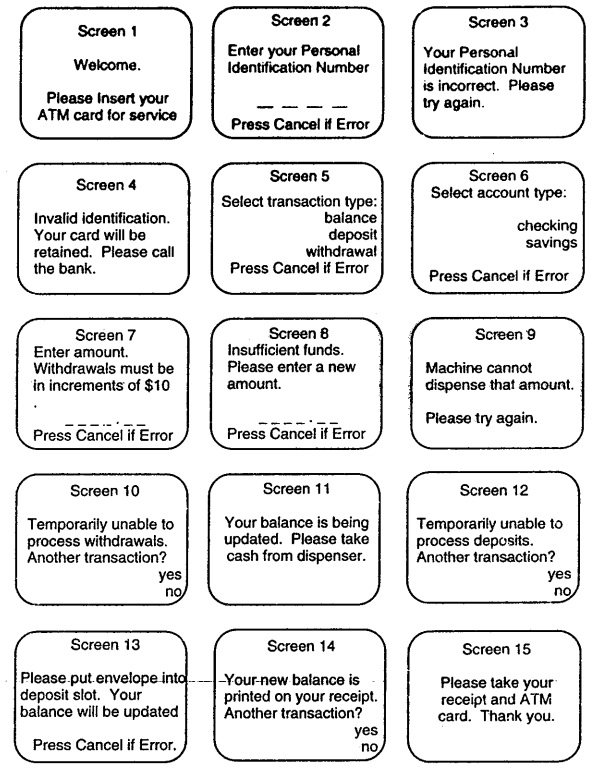

Figure 2.4 SATM screens.

and processes the deposit. The deposit amount is entered as an unposted amount in the local ATM file, and the count of deposits per month is incremented. Both of these (and other information) are processed by the master ATM (centralized) system once a day. The system then displays screen 14.

If withdrawal is requested, the system checks the status (jammed or free) of the withdrawal chute in the terminal control file. If jammed, screen 10 is displayed; otherwise, screen 7 is displayed so the customer can enter the withdrawal amount. Once the withdrawal amount is entered, the system checks the terminal status file to see if it has enough money to dispense. If it does not, screen 9 is displayed; otherwise the withdrawal is processed. The system checks the customer balance (as described in the balance request transaction); if the funds are insufficient, screen 8 is displayed. If the account balance is sufficient, screen 11 is displayed and the money is dispensed. The withdrawal amount is written to the unposted local ATM file, and the count of withdrawals per month is incremented. The balance is printed on the transaction receipt as it is for a balance request transaction. After the cash has been removed, the system displays screen 14.

When the No button is pressed in screens 10, 12, or 14, the system presents screen 15 and returns the customer's ATM card. Once the card is removed from

the card slot, screen 1 is displayed. When the Yes button is pressed in screens 10, 12, or 14, the system presents screen 5 so the customer can select additional transactions.

2.5.2 Discussion

A surprising amount of information is "buried" in the system description just given. For instance, if you read it closely, you can infer that the terminal only contains $10 bills (see screen 7). This textual definition is probably more precise than what is usually encountered in practice. The example is deliberately simple (hence the name).

A plethora of questions could be resolved by a list of assumptions. For example, is there a borrowing limit? What keeps a customer from taking out more than his actual balance if he goes to several ATM terminals? A lot of start-up questions are used: how much cash is initially in the machine? How are new customers added to the system? These and other real-world refinements are eliminated to maintain simplicity.

2.6 The Currency Converter

The currency conversion program is another event-driven program that emphasizes code associated with a graphical user interface (GUI). A sample GUI built with Visual Basic is shown in Figure 2.5.

The application converts U.S. dollars to any of four currencies: Brazilian reals, Canadian dollars, European Union euros, and Japanese yen. Currency selection is governed by the radio buttons (Visual Basic option buttons), which are mutually exclusive. When a country is selected, the system responds by completing the label; for example, "Equivalent in ..." becomes "Equivalent in Canadian dollars" if the Canada button is clicked. Also, a small Canadian flag appears next to the

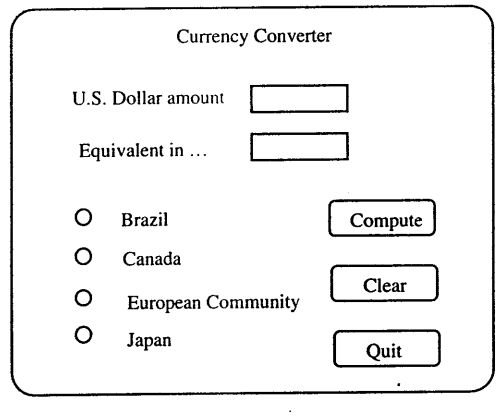

Figure 2.5 Currency converter GUI.

output position for the equivalent currency amount. Either before or after currency selection, the user inputs an amount in U.S. dollars. Once both tasks are accomplished, the user can click on the Compute button, the Clear button, or the Quit button. Clicking on the Compute button results in the conversion of the U.S. dollar amount to the equivalent amount in the selected currency. Clicking on the Clear button resets the currency selection, the U.S. Dollar amount, and the equivalent currency amount and the associated label. Clicking on the Quit button ends the application. This example nicely illustrates a description with UML and an object-oriented implementation; we will revisit it in Part V.

2.7 Saturn Windshield Wiper Controller

The windshield wiper on some Saturn automobiles is controlled by a lever with a dial. The lever has four positions: OFF, INT (for intermittent), LOW, and HIGH; and the dial has three positions, numbered simply 1, 2, and 3. The dial positions indicate three intermittent speeds, and the dial position is relevant only when the lever is at the INT position. The decision table below shows the windshield wiper speeds (in wipes per minute) for the lever and dial positions.

c1. Lever	OFF	INT	INT	INT	LOW	HIGH
c2. Dial	n/a	1	2	3	n/a	n/a
a1. Wiper	0	4	6	12	30	60

We will use this example in our discussion of interaction testing (see Chapter 15).

References

Brown, J.R. and Lipov, M., Testing for software reliability, *Proceedings of the International Symposium on Reliable Software,* Los Angeles, April 1975, pp. 518–527.

Chellappa, Mallika, Nontraversible Paths in a Program, *IEEE Transactions on Software Engineering,* Vol. SE-13, No. 6, June 1987, pp. 751–756.

Clarke, Lori A. and Richardson, Debra J., The application of error sensitive strategies to debugging, *ACM SIGSOFT Software Engineering Notes,* Vol. 8, No. 4, August 1983.

Clarke, Lori A. and Richardson, Debra J., A reply to Foster's comment on "The Application of Error Sensitive Strategies to Debugging," *ACM SIGSOFT Software Engineering Notes,* Vol. 9, No. 1, January 1984.

Gruenberger, F., Program testing, the historical perspective, in *Program Test Methods,* William C. Hetzel, Ed., Prentice-Hall, New York, 1973, pp. 11–14.

Hetzel, Bill, *The Complete Guide to Software Testing,* 2nd ed., QED Information Sciences, Inc., Wellesley, MA, 1988.

Inglis, Stuart J., *Planets, Stars, and Galaxies,* 4th Ed., John Wiley & Sons, New York, 1961.

Myers, Glenford J., *The Art of Software Testing,* Wiley Interscience, New York, 1979.

Pressman, Roger S., *Software Engineering: A Practitioner's Approach,* McGraw-Hill, New York, 1982.

Topper, Andrew et al., *Structured Methods: Merging Models, Techniques, and CASE,* McGraw-Hill, New York, 1993.

Exercises

1. Revisit the traditional triangle program flowchart in Figure 2.1. Can the variable match ever have the value of 4? Of 5? Is it ever possible to "execute" the following sequence of numbered boxes: 1, 2, 5, 6 ?
2. Recall the discussion from Chapter 1 about the relationship between the specification and the implementation of a program. If you study the implementation of NextDate carefully, you will see a problem. Look at the CASE clause for 30-day months (4, 6, 9, 11). There is no special action for day = 31. Discuss whether this implementation is correct. Repeat this discussion for the treatment of values of day 29 in the CASE clause for February.
3. In Chapter 1, we mentioned that part of a test case is the expected output. What would you use as the expected output for a NextDate test case of June 31, 1812? Why?
4. One common addition to the triangle problem is to check for right triangles. Three sides constitute a right triangle if the Pythagorean relationship is satisfied: $c^2 = a^2 + b^2$. This change makes it convenient to require that the sides be presented in increasing order, i.e., a <= b <= c. Extend the Triangle3 program to include the right triangle feature. We will use this extension in the exercise sections in Parts II and III.
5. What will the Triangle2 program do for the sides -3, -3, 5? Discuss this in terms of the considerations we made in Chapter 1.
6. The function YesterDate is the inverse of NextDate. Given a month, day, year, YesterDate returns the date of the day before. Develop a program in your favorite language (or our generalized pseudocode) for YesterDate. We will also use this as a continuing exercise.
7. Part of the art of GUI design is to prevent user input errors. Event-driven applications are particularly vulnerable to input errors because events can occur in any order. As the given pseudocode definition stands, a user could enter a U.S. dollar amount and then click on the compute button without selecting a country. Similarly, a user could select a country and then click on the compute button without inputting a dollar amount. In an object-oriented application, we can control this by being careful about when we instantiate objects. Revise the GUI class pseudocode to prevent these two errors.
8. The CRC Press Web site (http://www.crcpress.com) contains some software supplements for this book. There is a series of exercises that I use in my graduate class in software testing; the first part of a continuing exercise is to use the naive.exe (runs on a PC with Windows) program to test the Triangle, NextDate, and Commission problems. The Visual Basic program is self-explanatory. As a start to becoming a testing craftsperson, use naive.exe to test our three examples in an intuitive (hence "naive") way. There are faults inserted into each program. If (when) you find failures, try to hypothesize the underlying fault. Keep your results for comparison to ideas in Chapters 5, 6, and 9.

Chapter 3

Discrete Math for Testers

More than any other life cycle activity, testing lends itself to mathematical description and analysis. In this chapter and in the next, testers will find the mathematics they need. Following the craftsperson metaphor, the mathematical topics presented here are tools; a testing craftsperson should know how to use them well. With these tools, a tester gains rigor, precision, and efficiency — all of which improve testing. The "for testers" part of the chapter title is important: this chapter is written for testers who either have a sketchy math background or who have forgotten some of the basics. Serious mathematicians (or maybe just those who take themselves seriously) will likely be annoyed by the informal discussion here. If you are already comfortable with the topics in this chapter, skip to the next chapter and start right in on graph theory.

In general, discrete mathematics is more applicable to functional testing, while graph theory pertains more to structural testing. "Discrete" raises a question: What might be indiscrete about mathematics? The mathematical antonym is continuous, as in calculus, which software developers (and testers) seldom use. Discrete math includes set theory, functions, relations, propositional logic, and probability theory, each of which is discussed here.

3.1 Set Theory

How embarrassing to admit, after all the lofty expiation of rigor and precision, that no explicit definition of a set exists. This is really a nuisance because set theory is central to these two chapters on math. At this point, mathematicians make an important distinction: naive versus axiomatic set theory. In naive set theory, a set is recognized as a primitive term, much like point and line are primitive concepts in geometry. Here are some synonyms for "set": collection, group, bunch — you get the idea. The important thing about a set is that it lets us refer to several things as a group, or a whole. For example, we might wish

to refer to the set of months that have exactly 30 days (we need this set when we test the NextDate function from Chapter 2). In set theory notation, we write:

$$M1 = \{April, June, September, November\}$$

and we read this notation as "M1 is the set whose elements are the months April, June, September, November."

3.1.1 Set Membership

The items in a set are called elements or members of the set, and this relationship is denoted by the symbol $\in$. Thus we could write April $\in$ M1. When something is not a member of a set, we use the symbol $\notin$, so we might write December $\notin$ M1.

3.1.2 Set Definition

A set is defined in three ways: by simply listing its elements, by giving a decision rule, or by constructing a set from other sets. The listing option works well for sets with only a few elements as well as for sets in which the elements obey an obvious pattern. We might define the set of years in the NextDate program as follows:

$$Y = \{1812, 1813, 1814, \ldots, 2011, 2012\}$$

When we define a set by listing its elements, the order of the elements is irrelevant. We will see why when we discuss set equality. The decision rule approach is more complicated, and this complexity carries both advantages and penalties. We could define the years for NextDate as:

$$Y = \{year : 1812 \leq year \leq 2012\}$$

which reads "Y is the set of all years such that (the colon is "such that") the years are between 1812 and 2012 inclusive." When a decision rule is used to define a set, the rule must be unambiguous. Given any possible value of year, we can therefore determine whether or not that year is in our set Y.

The advantage of defining sets with decision rules is that the unambiguity requirement forces clarity. Experienced testers have encountered "untestable requirements." Many times, the reason that such requirements cannot be tested boils down to an ambiguous decision rule. In our triangle program, for example, suppose we defined a set:

$$N = \{t : t \text{ is a nearly equilateral triangle}\}$$

We might say that the triangle with sides (500, 500, 501) is an element of N, but how would we treat the triangles with sides (50, 50, 51) or (5, 5, 6)?

A second advantage of defining sets with decision rules is that we might be interested in sets where the elements are difficult to list. In the commission problem, for example, we might be interested in the set:

$$S = \{sales : \text{the 15\% commission rate applies to the sale}\}$$

We cannot easily write down the elements of this set; but given a particular value for sale, we can easily apply the decision rule.

The main disadvantage of decision rules is that they can become logically complex, particularly when they are expressed with the predicate calculus quantifiers $\exists$ ("there exists") and $\forall$ ("for all"). If everyone understands this notation, the precision is helpful; too often, customers are overwhelmed by statements with these quantifiers. A second problem with decision rules has to do with self-reference. This is interesting, but it really has very little application for testers. The problem arises when a decision rule refers to itself, which is a circularity. As an example, the Barber of Seville "is the man who shaves everyone who does not shave himself."

3.1.3 The Empty Set

The empty set, denoted by the symbol $\varnothing$, occupies a special place in set theory. The empty set contains no elements. At this point, mathematicians will digress to prove a lot of facts about empty sets:

- The empty set is unique; that is, there cannot be two empty sets (we will take their word for it).
- $\varnothing$, $\{\varnothing\}$, $\{\{\varnothing\}\}$, are all different sets (we will not need this).

It is useful to note that, when a set is defined by a decision rule that is always false, the set is empty. For instance,

$$\varnothing = \{year : 2012 \leq year \leq 1812\}$$

3.1.4 Venn Diagrams

Sets are commonly pictured by Venn diagrams — as in Chapter 1, when we discussed sets of specified and programmed behaviors. In a Venn diagram, a set is depicted as a circle; points in the interior of the circle correspond to elements of the set. Then, we might draw our set M1 of 30-day months as in Figure 3.1.

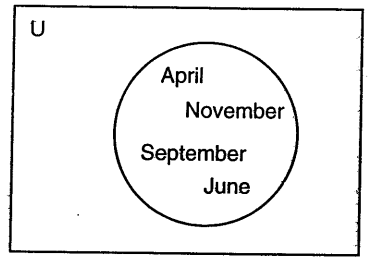

Figure 3.1 Venn diagram of the set of 30-day months.

Venn diagrams communicate various set relationships in an intuitive way, but some picky questions arise. What about finite versus infinite sets? Both can be drawn as Venn diagrams; in the case of finite sets, we cannot assume that every interior point corresponds to a set element. We do not need to worry about this, but it is helpful to know the limitations. Sometimes, we will find it helpful to label specific elements.

Another sticking point has to do with the empty set. How do we show that a set, or maybe a portion of a set, is empty? The common answer is to shade empty regions, but this is often contradicted by other uses in which shading is used to highlight regions of interest. The best practice is to provide a legend that clarifies the intended meaning of shaded areas.

It is often helpful to think of all the sets in a discussion as being subsets of some larger set, known as the universe of discourse. We did this in Chapter 1 when we chose the set of all program behaviors as our universe of discourse. The universe of discourse can usually be guessed from given sets. In Figure 3.1, most people would take the universe of discourse to be the set of all months in a year. Testers should be aware that assumed universes of discourse are often sources of confusion. As such, they constitute a subtle point of miscommunication between customers and developers.

3.1.5 Set Operations

Much of the expressive power of set theory comes from basic operations on sets: union, intersection, and complement. Other handy operations are used: relative complement, symmetric difference, and Cartesian product. Each of these is defined next. In each of these definitions, we begin with two sets, A and B, contained in some universe of discourse U. The definitions use logical connectives from the propositional calculus: and ($\wedge$), or ($\vee$), exclusive–or ($\oplus$), and not ($\neg$).

Definition

Given sets A and B,

Their union is the set $A \cup B = \{x : x \in A \vee x \in B\}$
Their intersection is the set $A \cap B = \{x : x \in A \wedge x \in B\}$
The complement of A is the set $A' = \{x : x \notin A\}$
The relative complement of B with respect to A is the set $A - B = \{x : x \in A \wedge x \notin B\}$
The symmetric difference of A and B is the set $A \oplus B = \{x : x \in A \oplus x \in B\}$

Venn diagrams for these sets are shown in Figure 3.2.

The intuitive expressive power of Venn diagrams is very useful for describing relationships among test cases and among items to be tested. Looking at the Venn diagrams in Figure 3.2, we might guess that:

$$A \oplus B = (A \cup B) - (A \cap B)$$

This is the case, and we could prove it with propositional logic.

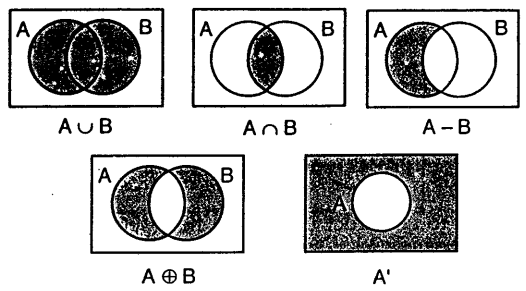

Figure 3.2 Venn diagrams of basic sets.

Venn diagrams are used elsewhere in software development: together with directed graphs, they are the basis of the StateCharts notations, which are among the most rigorous specification techniques supported by CASE technology. StateCharts are also the control notation chosen for the UML, the Universal Modeling Language from Rational Corp. and the Object Management Group.

The Cartesian product (also known as the cross product) of two sets is more complex; it depends on the notion of ordered pairs, which are two element sets in which the order of the elements is important. The usual notation for unordered and ordered pairs is:

unordered pair: (a, b)
ordered pair: < a, b >

The difference is that, for $a \neq b$,

(a, b) = (b, a), but
< a, b > $\neq$ < b, a >

This distinction is important to the material in Chapter 4; as we shall see, the fundamental difference between ordinary and directed graphs is exactly the difference between unordered and ordered pairs.

Definition

The Cartesian product of two sets A and B is the set

$$A \times B = \{\langle x, y\rangle : x \in A \wedge x \in B\}$$

Venn diagrams do not show Cartesian products, so we will look at a short example. The Cartesian product of the sets A = {1, 2, 3} and B = {w, x, y, z} is the set:

$$A \times B = \{\langle 1, w\rangle, \langle 1, x\rangle, \langle 1, y\rangle, \langle 1, z\rangle, \langle 2, w\rangle, \langle 2, x\rangle, \langle 2, y\rangle, \langle 2, z\rangle, \langle 3, w\rangle, \langle 3, x\rangle, \langle 3, y\rangle, \langle 3, z\rangle\}$$

The Cartesian product has an intuitive connection with arithmetic. The cardinality of a set A is the number of elements in A and is denoted by $|A|$. (Some authors prefer Card(A).) For sets A and B, $|A \times B| = |A| \times |B|$. When we study

functional testing in Chapter 5, we will use the Cartesian product to describe test cases for programs with several input variables. The multiplicative property of the Cartesian product means that this form of testing generates a very large number of test cases.

3.1.6 Set Relations

We use set operations to construct interesting new sets from existing sets. When we do, we often would like to know something about the way the new and the old sets are related. Given two sets, A and B, we define three fundamental set relationships:

Definition

A is a subset of B, written $A \subseteq B$, if and only if (iff) $a \in A \Rightarrow a \in B$
A is a proper subset of B, written $A \subset B$, iff $A \subseteq B \wedge B - A \neq \varnothing$
A and B are equal sets, written $A = B$, iff $A \subseteq B \wedge B \subseteq A$

In plain English, set A is a subset of set B if every element of A is also an element of B. In order to be a proper subset of B, A must be a subset of B and there must be some element in B that is not an element of A. Finally, the sets A and B are equal if each is a subset of the other.

3.1.7 Set Partitions

A partition of a set is a very special situation that is extremely important for testers. Partitions have several analogs in everyday life: we might put up partitions to separate an office area into individual offices; we also encounter political partitions when a state is divided up into legislative districts. In both of these, notice that the sense of "partition" is to divide up a whole into pieces such that everything is in some piece, and nothing is left out. More formally:

Definition

Given a set B, and a set of subsets A1, A2, ..., An of B, the subsets are a partition of B iff

$A1 \cup A2 \cup \ldots \cup An = B$, and
$i \neq j \Rightarrow Ai \cap Aj = \varnothing$

Because a partition is a set of subsets, we frequently refer to individual subsets as elements of the partition.

The two parts of this definition are important for testers. The first part guarantees that every element of B is in some subset, while the second part guarantees that no element of B is in two of the subsets. This corresponds well with the legislative districts example: everyone is represented by some legislator,

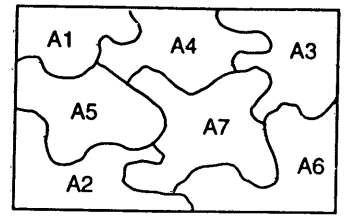

Figure 3.3 Venn diagram of a partition.

and nobody is represented by two legislators. A picture puzzle is another good example of a partition, in fact, Venn diagrams of partitions are often drawn like puzzles, as in Figure 3.3.

Partitions are helpful to testers because the two definitional properties yield important assurances: completeness (everything is somewhere) and nonredundancy. When we study functional testing, we shall see that its inherent weakness is the vulnerability to both gaps and redundancies: some things may remain untested, while others are tested repeatedly. One of the difficulties of functional testing centers on finding an appropriate partition. In the Triangle Program, for example, the universe of discourse is the set of all triplets of positive integers. (Note that this is actually a Cartesian product of the set of positive integers with itself three times.) We might partition this universe three ways:

1. Into triangles and nontriangles
2. Into equilateral, isosceles, scalene, and nontriangles
3. Into equilateral, isosceles, scalene, right, and nontriangles

At first these partitions seem okay, but there is a problem with the last partition. The sets of scalene and right triangles are not disjoint (the triangle with sides 3,4,5 is a right triangle that is scalene.)

3.1.8 Set Identities

Set operations and relations, when taken together, yield an important class of set identities that can be used to algebraically simplify complex set expressions. Math students usually have to derive all these; we will just list them and (occasionally) use them.

Name	*Expression*
Identity Laws	$A \cup \varnothing = A$
	$A \cap U = A$
Domination Laws	$A \cup U = U$
	$A \cap \varnothing = \varnothing$
Idempotent Laws	$A \cup A = A$
	$A \cap A = A$
Complementation Laws	$(A')' = A$

Name	Expression
Commutative Laws	$A \cup B = B \cup A$ $A \cap B = B \cap A$
Associative Laws	$A \cup (B \cup C) = (A \cup B) \cup C$ $A \cap (B \cap C) = (A \cap B) \cap C$
Distributive Laws	$A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$ $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$
DeMorgan's Laws	$(A \cup B)' = A' \cap B'$ $(A \cap B)' = A' \cup B'$

3.2 Functions

Functions are a central notion to software development and testing. The whole functional decomposition paradigm, for example, implicitly uses the mathematical notion of a function. We make this notion explicit here because all functional testing is based on it.

Informally, a function associates elements of sets. In the NextDate program, for example, the function of a given date is the date of the following day, and in the triangle problem, the function of three input integers is the kind of triangle formed by sides with those lengths. In the commission problem, the salesperson's commission is a function of sales, which in turn is a function of the number of locks, stocks, and barrels sold. Functions in the ATM system are much more complex; not surprisingly, this will add complexity to the testing.

Any program can be thought of as a function that associates its outputs with its inputs. In the mathematical formulation of a function, the inputs are the domain and the outputs are the range of the function.

Definition

Given sets A and B, a function f is a subset of $A \times B$ such that, for $a_i, a_j \in A$, $b_i, b_j \in B$, and $f(a_i) = b_i$, $f(a_j) = b_j$, $b_i \neq b_j \Rightarrow a_i \neq a_j$.

Formal definitions like this one are notoriously terse, so let us take a closer look. The inputs to the function f are elements of the set A, and the outputs of f are elements of B. What the definition says is that the function f is "well behaved" in the sense that an element in A is never associated with more than one element of B. (If this could happen, how would we ever test such a function? This is an example of nondeterminism.)

3.2.1 Domain and Range

In the definition just given, the set A is the domain of the function f, and the set B is the range. Because input and output have a "natural" order, it is an easy step to say that a function f is really a set of ordered pairs in which the first element is from the domain and the second element is from the range. Here are two common notations for function:

$$f : A \rightarrow B$$
$$f \subseteq A \times B$$

We have not put any restrictions on the sets A and B in this definition. We could have A = B, and either A or B could be a Cartesian product of other sets.

3.2.2 Function Types

Functions are further described by particulars of the mapping. In the definition below, we start with a function $f : A \rightarrow B$, and we define the set:

$$f(A) = \{b_i \in B : b_i = f(a_i) \text{ for some } a_i \in A\}$$

This set is sometimes called the image of A under f.

Definition

f is a function from A onto B iff f(A) = B
f is a function from A into B iff $f(A) \subset B$ (note the proper subset here!)
f is a one-to-one function from A to B iff, for all $a_i, a_j \in A$, $a_i \neq a_j \Rightarrow f(a_i) \neq f(a_j)$
f is a many-to-one function from A to B iff, there exists $a_i, a_j \in A$, $a_i \neq a_j$ such that $f(a_i) = f(a_j)$.

Back to plain English, if f is a function from A onto B, we know that every element of B is associated with some element of A. If f is a function from A into B, we know that there is at least one element of B that is not associated with an element of A. One-to-one functions guarantee a form of uniqueness: distinct domain elements are never mapped to the same range element. (Notice this is the inverse of the "well-behaved" attribute described earlier.) If a function is not one-to-one, it is many-to-one; that is, more than one domain element can be mapped to the same range element. In these terms, the "well-behaved" requirement prohibits functions from being one-to-many. Testers familiar with relational databases will recognize that all these possibilities (one-to-one, one-to-many, many-to-one, and many-to-many) are allowed for relations.

Referring again to our testing examples, suppose we take A, B, and C to be sets of dates for the NextDate program, where:

$A = \{\text{date} : 1 \text{ January } 1812 \leq \text{date} \leq 31 \text{ December } 2012\}$
$B = \{\text{date} : 2 \text{ January } 1812 \leq \text{date} \leq 1 \text{ January } 2013\}$
$C = A \cup B$

Now, NextDate : $A \rightarrow B$ is a one-to-one, onto function, and NextDate : $A \rightarrow C$ is a one-to-one, into function.

It makes no sense for NextDate to be many-to-one, but it is easy to see how the triangle problem can be many-to-one. When a function is one-to-one and onto, such as NextDate : $A \rightarrow B$ previously, each element of the domain corresponds to exactly one element of the range; conversely, each element of the

range corresponds to exactly one element of the domain. When this happens, it is always possible to find an inverse function (see the YesterDate problem in Chapter 2) that is one-to-one from the range back to the domain.

All this is important for testing. The into versus onto distinction has implications for domain- and range-based functional testing, and one-to-one functions require much more testing than many-to-one functions.

3.2.3 Function Composition

Suppose we have sets and functions such that the range of one is the domain of the next:

$$f : A \rightarrow B$$
$$g : B \rightarrow C$$
$$h : C \rightarrow D$$

When this occurs, we can compose the functions. To do this, let us refer to specific elements of the domain and range sets $a \in A$, $b \in B$, $c \in C$, $d \in D$, and suppose that $f(a) = b$, $g(b) = c$, and $h(c) = d$. Now the composition of functions g and f is:

$$\begin{aligned} h \circ g \circ f(a) &= h(g(f(a))) \\ &= h(g(b)) \\ &= h(c) \\ &= d \end{aligned}$$

Function composition is a very common practice in software development; it is inherent in the process of defining procedures and subroutines. We have an example of it in the commission program, in which:

$$f_1(\text{locks, stocks, barrels}) = \text{sales}$$
$$f_2(\text{sales}) = \text{commission}$$

Composed chains of functions can be problematic for testers, particularly when the range of one function is a proper subset of the domain of the "next" function in the chain. Figure 3.4 shows how this can happen in a program defined by a dataflow diagram.

In the causal flow, the composition $g \cdot f(a)$ (which we know to be g(b), which yields c) is a rather assemblyline-like process. In the noncausal flow, the possibility of more than one source of b values for the datastore B raises two problems for testers. Multiple sources of b values might raise problems of domain/range compatibility; and even if this is not a problem, there might be timing anomalies with respect to b values. (What if g used an "old" b value?)

A special case of composition can be used, which helps testers in a curious way. Recall we discussed how one-to-one onto functions always have an inverse function. It turns out that this inverse function is unique and is guaranteed to exist (again, the math folks would prove this). If f is a one-to-one function from A onto

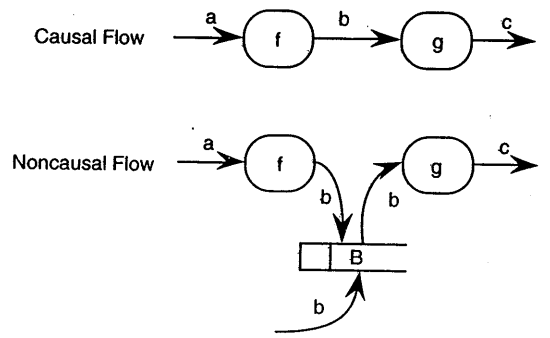

Figure 3.4 Causal and noncausal flows in a data flow diagram.

B, we denote its unique inverse by f^{-1}. It turns out that for $a \in A$ and $b \in B$, $f^{-1} \cdot f(a) = a$ and $f \cdot f^{-1}(b) = b$. The NextDate and YesterDate programs are such inverses. The way this helps testers is that, for a given function, its inverse acts as a "cross-check," and this can often expedite the identification of functional test cases.

3.3 Relations

Functions are a special case of a relation: both are subsets of some Cartesian product, but in the case of functions, we have the "well-behaved" requirement that says that a domain element cannot be associated with more than one range element. This is borne out in everyday usage: when we say something "is a function" of something else, our intent is that there is a deterministic relationship present. Not all relationships are strictly functional. Consider the mapping between a set of patients and a set of physicians. One patient may be treated by several physicians, and one physician may treat several patients — a many-to-many mapping.

3.3.1 Relations among Sets

Definition

Given two sets A and B, a relation R is a subset of the Cartesian product $A \times B$.

Two notations are popular; when we wish to speck about the entire relation, we usually just write $R \subseteq A \times B$; for specific elements $a_i \in A$, $b_i \in B$, we write $a_i \; R \; b_i$. Most math texts omit treatment of relations; we are interested in them because they are essential to both data modeling and object-oriented analysis.

Next, we have to explain an overloaded term — cardinality. Recall that, as it applies to sets, cardinality refers to the number of elements in a set. Because a relation is also a set, we might expect that the cardinality of a relation refers to how many ordered pairs are in the set $R \subseteq A \times B$. Unfortunately, this is not the case.

Definition

Given two sets A and B, a relation $R \subseteq A \times B$, the cardinality of relation R is:

One-to-one iff R is a one-to-one function from A to B
Many-to-one iff R is a many-to-one function from A to B
One-to-many iff at least one element $a \in A$ is in two ordered pairs in R, that is $(a,b_i) \in R$ and $(a, b_j) \in R$
Many-to-many iff at least one element $a \in A$ is in two ordered pairs in R, that is $(a, b_i) \in R$ and $(a, b_j) \in R$. and at least one element $b \in B$ is in two ordered pairs in R, that is $(a_i, b) \in R$ and $(a_j, b) \in R$.

The distinction between functions into and onto their range has an analog in relations — the notion of participation.

Definition

Given two sets A and B, a relation $R \subseteq A \times B$, the participation of relation R is:

Total iff every element of A is in some ordered pair in R
Partial iff some element of A is not in some ordered pair in R
Onto iff every element of B is in some ordered pair in R
Into iff some element of B is not in some ordered pair in R

In plain English, a relation is total if it applies to every element of A, and partial if it does not apply to every element. Another term for this distinction is mandatory versus optional participation. Similarly, a relation is onto if it applies to every element of B, and into if it does not. The parallelism between total/partial and onto/into is curious and deserves special mention here. From the standpoint of relational database theory, no reason exists for this; in fact, a compelling reason exists to avoid this distinction. Data modeling is essentially declarative, while process modeling is essentially imperative. The parallel sets of terms force a direction on relations, when in fact no need exists for the directionality. Part of this is a likely holdover from the fact that Cartesian products consist of ordered pairs, which clearly have a first and second element.

So far, we have only considered relations between two sets. Extending relations to three or more sets is more complicated than simply the Cartesian product. Suppose, for example, we had three sets, A, B, and C, and a relation $R \subseteq A \times B \times C$. Do we intend the relation to be strictly among three elements, or is it between one element and an ordered pair (there would be three possibilities here)? This line of thinking also needs to be applied to the definitions of cardinality and participation. It is straightforward for participation, but cardinality is essentially a binary property. (Suppose, for example the relation is one-to-one from A to B and is many-to-one from A to C.) We discussed a three-way relation in Chapter 1, when we examined the relationships among specified, implemented, and tested program behaviors. We would like to have some form of totality between test cases and specification–implementation pairs; we will revisit this when we study functional and structural testing.

Testers need to be concerned with the definitions of relations because they bear directly on software properties to be tested. The onto/into distinction, for example, bears directly on what we will call output-based functional testing. The mandatory–optional distinction is the essence of exception handling, which also has implications for testers.

3.3.2 Relations on a Single Set

Two important mathematical relations are used, both of which are defined on a single set: ordering relations and equivalence relations. Both are defined with respect to specific properties of relations.

Let A be a set, and let $R \subseteq A \times A$ be a relation defined on A, with $\langle a, a\rangle$, $\langle a, b\rangle$, $\langle b, a\rangle$, $\langle b, c\rangle$, $\langle a, c\rangle \in R$. Relations have four special attributes:

Definition

A relation $R \subseteq A \times A$ is:

Reflexive iff for all $a \in A$, $\langle a, a\rangle \in R$
Symmetric iff $\langle a, b\rangle \in R \Rightarrow \langle b, a\rangle \in R$
Antisymmetric $\langle a, b\rangle, \langle b, a\rangle \in R \Rightarrow a = b$
Transitive iff $\langle a, b\rangle, \langle b, c\rangle \in R \Rightarrow \langle a, c\rangle \in R$

Family relationships are nice examples of these properties. You might want to think about the following relationships and decide for yourself which attributes apply: brother of, sibling of, and ancestor of. Now we can define the two important relations.

Definition

A relation $R \subseteq A \times A$ is an ordering relation if R is reflexive, antisymmetric, and transitive.

Ordering relations have a sense of direction; some common ordering relations are Older than, $\geq$, $\Rightarrow$, and Ancestor of. (The reflexive part usually requires some fudging — we really should say Not Younger Than and Not a Descendant of.) Ordering relations are a common occurrence in software: data access techniques, hashing codes, tree structures, and arrays are all situations in which ordering relations are used.

The power set of a given set is the set of all subsets of the given set. The power set of the set A is denoted P(A). The subset relation $\subseteq$ is an ordering relation on P(A), because it is reflexive (any set is trivially a subset of itself), it is antisymmetric (the definition of set equality), and it is transitive.

Definition

A relation $R \subseteq A \times A$ is an equivalence relation if R is reflexive, symmetric, and transitive.

Mathematics is full of equivalence relations: equality and congruence are two quick examples. A very important connection exists between equivalence relations and partitions of a set. Suppose we have some partition A_1, A_2, ..., An of a set B, and we say that two elements, b_1 and b_2 of B, are related (i.e., b_1 R b_2) if b_1 and b_2 are in the same partition element. This relation is reflexive (any element is in its own partition), it is symmetric (if b_1 and b_2 are in a partition element, then b_2 and b_1 are), and it is transitive (if b_1 and b_2 are in the same set, and if b_2 and b_3 are in the same set, then b_1 and b_3 are in the same set). The relation in defined from the partition is called the equivalence relation induced by the partition. The converse process works in the same way. If we start with an equivalence relation defined on a set, we can define subsets according to elements that are related to each other. This turns out to be a partition, and is called the partition induced by the equivalence relation. The sets in this partition are known as equivalence classes. The end result is that partitions and equivalence relations are interchangeable, and this becomes a powerful concept for testers. Recall that the two properties of a partition are notions of completeness and nonredundancy. When translated into testing situations, these notions allow testers to make powerful, absolute statements about the extent to which a software item has been tested. In addition, great efficiency follows from testing just one element of an equivalence class and assuming that the remaining elements will behave similarly.

3.4 Propositional Logic

We have already been using propositional logic notation; if you were perplexed by this usage definition before, you are not alone. Set theory and propositional logic have a chicken-and-egg relationship — it is hard to decide which should be discussed first. Just as sets are taken as primitive terms and are therefore not defined, we take propositions to be primitive terms. A proposition is a sentence that is either true or false, and we call these the truth values of the proposition. Furthermore, propositions are unambiguous: given a proposition, it is always possible to tell whether it is true or false. The sentence "Mathematics is difficult" would not qualify as a proposition because of the ambiguity. We usually denote propositions with lower-case letters p, q, and r. Propositional logic has operations, expressions, and identities that are very similar to (in fact, they are isomorphic) set theory.

3.4.1 Logical Operators

Logical operators (also known as logical connectives or operations) are defined in terms of their effect on the truth values of the propositions to which they are applied. This is easy, only two values are used: T (for true) and F (for false). Arithmetic operators could also be defined this way (in fact, that is how they are taught to children), but the tables become too large. The three basic logical operators are and ($\wedge$) or ($\vee$), and not ($\neg$); these are sometimes called conjunction, disjunction, and negation. Negation is the only unary (one operand) logical operator; the others are all binary.

p	q	$p \wedge q$	$p \vee q$	$\neg p$
T	T	T	T	F
T	F	F	T	F
F	T	F	T	T
F	F	F	F	T

Conjunction and disjunction are familiar in everyday life: a conjunction is true only when all components are true, and a disjunction is true if at least one component is true. Negations also behave as we expect. Two other common connectives are used: exclusive–or ($\oplus$) and IF–THEN ($\rightarrow$). They are defined as follows:

p	q	$p \oplus q$	$p \rightarrow q$
T	T	F	T
T	F	T	F
F	T	T	T
F	F	F	T

An exclusive–or is true only when one of the propositions is true, while a disjunction (or inclusive–or) is true also when both propositions are true. The IF–THEN connective usually causes the most difficulty. The easy view is that this is just a definition; but because the other connectives all transfer nicely to natural language, we have similar expectations for IF–THEN. The quick answer is that the IF–THEN connective is closely related to the process of deduction: in a valid deductive syllogism, we can say "if premises, then conclusion" and the IF–THEN statement will be a tautology.

3.4.2 Logical Expressions

We use logical operators to build logical expressions in exactly the same way that we use arithmetic operators to build algebraic expressions. We can specify the order in which operators are applied with the usual conventions on parentheses, or we can employ a precedence order (negation first, then conjunction followed by disjunction). Given a logical expression, we can always find its truth table by "building up" to it following the order determined by the parentheses. For example, the expression $\neg((p \rightarrow q) \wedge (q \rightarrow p))$ has the following truth table:

p	q	$p \rightarrow q$	$q \rightarrow p$	$(p \rightarrow q) \wedge ((q \rightarrow p)$	$\neg((p \rightarrow q) \wedge ((q \rightarrow p))$
T	T	T	T	T	F
T	F	F	T	F	T
F	T	T	F	F	T
F	F	T	T	T	F

3.4.3 Logical Equivalence

The notions of arithmetic equality and identical sets have analogs in propositional logic. Notice that the expressions $\neg((p \rightarrow q) \wedge (q \rightarrow p))$ and $p \oplus q$ have identical

truth tables. This means that, no matter what truth values are given to the base propositions p and q, these expressions will always have the same truth value. This property can be defined in several ways; we use the simplest.

Definition

Two propositions p and q are logically equivalent (denoted $p \Leftrightarrow q$) iff their truth tables are identical.

By the way, the curious "iff" abbreviation we have been using for "if and only if" is sometimes called the bi-conditional, so the proposition p iff q is really $(p \rightarrow q) \wedge (q \rightarrow p)$, which is denoted $p \leftrightarrow q$.

Definition

A proposition that is always true is a tautology; a proposition that is always false is a contradiction.

In order to be a tautology or a contradiction, a proposition must contain at least one connective and two or more primitive propositions. We sometimes denote a tautology as a proposition T, and a contradiction as a proposition F. We can now state several laws that are direct analogs of the ones we had for sets.

Law	*Expression*
Identity	$p \wedge T \Leftrightarrow p$
	$p \vee F \Leftrightarrow p$
Domination	$p \vee T \Leftrightarrow T$
	$p \wedge F \Leftrightarrow F$
Idempotent	$p \wedge p \Leftrightarrow p$
	$p \vee p \Leftrightarrow p$
Complementation	$\neg(\neg p) \Leftrightarrow p$
Commutative	$p \wedge q \Leftrightarrow q \wedge p$
	$p \vee q \Leftrightarrow q \vee p$
Associative	$p \wedge (q \wedge r) \Leftrightarrow (p \wedge q) \wedge r$
	$p \vee (q/r) \Leftrightarrow (p \vee q) \vee r$
Distributive	$p \wedge (q \vee r) \Leftrightarrow (p \wedge q)/(p \wedge r)$
	$p \vee (q \wedge r) \Leftrightarrow (p \vee q) \wedge (p \vee r)$
DeMorgan's	$\neg(p \wedge q) \Leftrightarrow \neg p \vee \neg q$
	$\neg(p \vee q) \Leftrightarrow \neg p \wedge \neg q$

3.5 Probability Theory

We will have two occasions to use probability theory in our study of software testing: one deals with the probability that a particular path of statements executes, and the other generalizes this to a popular industrial concept called an operational profile (see Chapter 14). Because of this limited use, we will only cover the rudiments here.

As with both set theory and propositional logic, we start out with a primitive concept — the probability of an event. Here is the definition provided by a classic textbook (Rosen, 1991):

> The probability of an event E, which is a subset of a finite sample space S of equally likely outcomes, is $p(E) = |E|/|S|$.

This definition hinges on the idea of an experiment that results in an outcome, the sample space is the set of all possible outcomes, and an event is a subset of outcomes. This definition is circular: What are "equally likely" outcomes? We assume these have equal probabilities, but then probability is defined in terms of itself.

The French mathematician Laplace had a reasonable working definition of probability two centuries ago. To paraphrase it, the probability that something occurs is the number of favorable ways it can occur divided by the total number of ways (favorable and unfavorable). Laplace's definition works well when we are concerned with drawing colored marbles out of a bag (probability folks are unusually concerned with their marbles; maybe there's a lesson here), but it does not extend well to situations in which it is hard to enumerate the various possibilities.

We will use our (refurbished) capabilities in set theory and propositional logic to arrive at a more cohesive formulation. As testers, we will be concerned with things that happen; we will call these events and say that the set of all events is our universe of discourse. Next, we will devise propositions about events, such that the propositions refer to elements in the universe of discourse. Now, for some universe U and some proposition p about elements of U, we make a definition:

Definition

The truth set T of a proposition p, written T(p), is the set of all elements in the universe U for which p is true.

Propositions are either true or false, therefore, a proposition p divides the universe of discourse into two sets, T(p) and (T(p))', where $T(p) \cup (T(p))' = U$. Notice that (T(p))' is the same as $T(\neg p)$. Truth sets facilitate a clear mapping among set theory, propositional logic, and probability theory.

Definition

The probability that a proposition p is true, denoted Pr(p), is $|T(p)|/|U|$.

With this definition, Laplace's "number of favorable ways" becomes the cardinality of the truth set T(p), and the total number of ways becomes the cardinality of the universe of discourse. This forces one more connection: because the truth set of a tautology is the universe of discourse, and the truth set of a contradiction is the empty set, the probabilities of $\emptyset$ and U are, respectively, 0 and 1.

The NextDate problem is a good source of examples. Consider the month variable and the proposition:

p(m) : m is a 30-day month

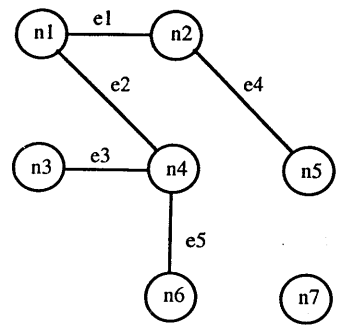

Figure 4.1 A graph with seven nodes and five edges.

Nodes are sometimes called vertices; edges are sometimes called arcs; and we sometimes call nodes the endpoints of an arc. The common visual form of a graph shows nodes as circles and edges as lines connecting pairs of nodes, as in Figure 4.1. We will use this figure as a continuing example, so take a minute to become familiar with it.

In the graph in Figure 4.1 the node and edge sets are

$$V = \{n_1, n_2, n_3, n_4, n_5, n_6, n_7\}$$
$$E = \{e_1, e_2, e_3, e_4, e_5\}$$
$$= \{(n_1, n_2), (n_1, n_4), (n_3, n_4), (n_2, n_5), (n_4, n_6)\}$$

To define a particular graph, we must first define a set of nodes and then define a set of edges between pairs of nodes. We usually think of nodes as program statements, and we have various kinds of edges, representing, for instance, flow of control or define/use relationships.

4.1.1 Degree of a Node

Definition

The degree of a node in a graph is the number of edges that have that node as an endpoint. We write deg(n) for the degree of node n.

We might say that the degree of a node indicates its "popularity" in a graph. In fact, social scientists use graphs to describe social interactions, in which nodes are people, edges often refer to things like "friendship," "communicates with," and so on. If we make a graph in which objects are nodes and edges are messages, the degree of a node (object) indicates the extent of integration testing that is appropriate for the object.

The degrees of the nodes in Figure 4.1 are:

$$\deg(n_1) = 2$$
$$\deg(n_2) = 2$$

$$\deg(n_3) = 1$$
$$\deg(n_4) = 3$$
$$\deg(n_5) = 1$$
$$\deg(n_6) = 1$$
$$\deg(n_7) = 0$$

4.1.2 Incidence Matrices

Graphs need not be represented pictorially — they can be fully represented in an incidence matrix. This concept becomes very useful for testers, so we will formalize it here. When graphs are given a specific interpretation, the incidence matrix always provides useful information for the new interpretation.

Definition

The incidence matrix of a graph G = (V, E) with m nodes and n edges is an m × n matrix, where the element in row i, column j is a 1 if and only if node i is an endpoint of edge j; otherwise, the element is 0.

The incidence matrix of the graph in Figure 4.1 is:

	e_1	e_2	e_3	e_4	e_5
n_1	1	1	0	0	0
n_2	1	0	0	1	0
n_3	0	0	1	0	0
n_4	0	1	1	0	1
n_5	0	0	0	1	0
n_6	0	0	0	0	1
n_7	0	0	0	0	0

We can make some observations about a graph by examining its incidence matrix. First, notice that the sum of the entries in any column is 2. That is because every edge has exactly two endpoints. If a column sum in an incidence matrix is ever something other than 2, there is a mistake somewhere. Thus, forming column sums is a form of integrity checking similar in spirit to that of parity checks. Next, we see that the row sum is the degree of the node. When the degree of a node is zero, as it is for node n_7, we say the node is isolated. (This might correspond to unreachable code, or to objects that are included but never used.)

4.1.3 Adjacency Matrices

The adjacency matrix of a graph is a useful supplement to the incidence matrix. Because adjacency matrices deal with connections, they are the basis of many later graph theory concepts.

Definition

The adjacency matrix of a graph G = (V, E) with m nodes is an m × m matrix, where the element in row i, column j is a 1 if and only if an edge exists between node i and node j; otherwise, the element is 0.

The adjacency matrix is symmetric (element i,j always equals element j,i), and a row sum is the degree of the node (as it was in the incidence matrix).

The adjacency matrix of the graph in Figure 4.1 is:

	n_1	n_2	n_3	n_4	n_5	n_6	n_7
n_1	0	1	0	1	0	0	0
n_2	1	0	0	0	1	0	0
n_3	0	0	0	1	0	0	0
n_4	1	0	1	0	0	1	0
n_5	0	1	0	0	0	0	0
n_6	0	0	0	1	0	0	0
n_7	0	0	0	0	0	0	0

4.1.4 Paths

As a preview of how we will use graph theory, the structural approaches to testing (see Part III) all center on types of paths in a program. Here, we define (interpretation-free) paths in a graph.

Definition

A path is a sequence of edges such that, for any adjacent pair of edges e_i, e_j in the sequence, the edges share a common (node) endpoint.

Paths can be described either as sequences of edges or as sequences of nodes; the node sequence choice is more common.

Some paths in the graph in Figure 4.1:

Path	*Node Sequence*	*Edge Sequence*
Between n_1 and n_5	n_1, n_2, n_5	e_1, e_4
Between n_6 and n_5	n_6, n_4, n_1, n_2, n_5	e_5, e_2, e_1, e_4
Between n_3 and n_2	n_3, n_4, n_1, n_2	e_3, e_2, e_1
Between n_1 and n_5	n_1, n_2, n_5	e_1, e_4

Paths can be generated directly from the adjacency matrix of a graph using a binary form of matrix multiplication and addition. In our continuing example, edge e_1 is between nodes n_1 and n_2, and edge e_4 is between nodes n_2 and n_5. In the product of the adjacency matrix with itself, the element in position (1, 2) forms a product with the element in position (2, 5), yielding an element in position

(1, 5), which corresponds to the two-edge path between n_1 and n_5. If we multiplied the product matrix by the original adjacency matrix again, we would get all three edge paths, and so on. At this point, the pure math folks go into a long digression to determine the length of the longest path in a graph; we will not bother. Instead, we focus our interest on the fact that paths connect "distant" parts of a graph.

The graph in Figure 4.1 predisposes a problem. It is not completely general, because it does not show all the situations that might occur in a graph. In particular, no paths exist in which a node occurs twice in the path. If it did, the path would be a loop (or circuit). We could create a circuit by adding an edge between nodes n_3 and n_6.

4.1.5 Connectedness

Paths let us speak about nodes that are connected; this leads to a powerful simplification device that is very important for testers.

Definition

Nodes n_i and n_j are connected if and only if they are in the same path.

"Connectedness" is an equivalence relation (see Chapter 3) on the node set of a graph. To see this, we can check the three defining properties of equivalence relations:

1. Connectedness is reflexive, because every node is obviously in a path of length 0 with itself.
2. Connectedness is symmetric, because if nodes n_i and n_j are in a path, then nodes n_j and n_i are in the same path.
3. Connectedness is transitive (see the discussion of adjacency matrix multiplication for paths of length 2).

Equivalence relations induce a partition (see Chapter 3 if you need a reminder), therefore, we are guaranteed that connectedness defines a partition on the node set of a graph. This permits the definition of components of a graph:

Definition

A component of a graph is a maximal set of connected nodes.

Nodes in the equivalence classes are components of the graph. The classes are maximal due to the transitivity part of the equivalence relation. The graph in Figure 4.1 has two components: $\{n_1, n_2, n_3, n_4, n_5, n_6\}$ and $\{n_7\}$.

4.1.6 Condensation Graphs

We are finally in a position to formalize an important simplification mechanism for testers.

Definition

Given a graph G = (V, E), its condensation graph is formed by replacing each component by a condensing node.

Developing the condensation graph of a given graph is an unambiguous (i.e., algorithmic) process. We use the adjacency matrix to identify path connectivity, and then use the equivalence relation to identify components. The absolute nature of this process is important: the condensation graph of a given graph is unique. This implies that the resulting simplification represents an important aspect of the original graph.

The components in our continuing example are $S_1 = \{n_1, n_2, n_3, n_4, n_5, n_6\}$ and $S_2 = \{n_7\}$.

No edges can be present in a condensation graph of an ordinary (undirected) graph. Two reasons are:

1. Edges have individual nodes as endpoints, not sets of nodes. (Here, we can finally use the distinction between n_7 and $\{n_7\}$.)
2. Even if we fudge the definition of edge to ignore this distinction, a possible edge would mean that nodes from two different components were connected, thus in a path, thus in the same (maximal!) component.

The implication for testing is that components are independent in an important way, thus they can be tested separately.

4.1.7 Cyclomatic Number

Another property of graphs has deep implications for testing: cyclomatic complexity.

Definition

The cyclomatic number of a graph G is given by V(G) = e − n + p, where

e is the number of edges in G
n is the number of nodes in G
p is the number of components in G

V(G) is the number of distinct regions in a graph. Recall our discussion of vector spaces and the notion of a basis set. One formulation of structural testing postulates the notion of basis paths in a program and shows that the cyclomatic number of the program graph (see the end of this chapter) is the number of these basis elements.

The cyclomatic number of our example graph is V(G) = 5 − 7 + 2 = 0. When we use cyclomatic complexity in testing, we will (usually) have strongly connected graphs, which will generate graphs with larger cyclomatic complexity.

4.2 Directed Graphs

Directed graphs are a slight refinement to ordinary graphs: edges acquire a sense of direction. Symbolically, the unordered pairs (n_i, n_j) become ordered pairs $\langle n_i, n_j \rangle$, and we speak of a directed edge going from node n_i to n_j, instead of being between the nodes.

Definition

A directed graph (or digraph) D = (V, E) consists of: a finite set $V = \{n_1, n_2, \ldots, n_m\}$ of nodes, and a set $E = \{e_1, e_2, \ldots, e_p\}$ of edges, where each edge $e_k = \langle n_i, n_j \rangle$ is an ordered pair of nodes $n_i, n_j \in V$.

In the directed edge $e_k = \langle n_i, n_j \rangle$, n_i is the initial (or start) node, and n_j is the terminal (or finish) node. Edges in directed graphs fit naturally with many software concepts: sequential behavior, imperative programming languages, time-ordered events, define/reference pairings, messages, function and procedure calls, and so on. Given this, you might ask why we spent (wasted?) so much time on ordinary graphs. The difference between ordinary and directed graphs is very analogous to the difference between declarative and imperative programming languages. In imperative languages (e.g., COBOL, FORTRAN, Pascal, C, Ada®), the sequential order of source language statements determines the execution time order of compiled code. This is not true for declarative languages (such as Prologue). The most common declarative situation for most software developers is Entity/Relationship modeling. In an E/R model, we choose entities as nodes and identify relationships as edges. (If a relationship involves three or more entities, we need the notion of a "hyper-edge" that has three or more endpoints.) The resulting graph of an E/R model is more properly interpreted as an ordinary graph. Good E/R modeling practice suppresses the sequential thinking that directed graphs promote.

When testing a program written in a declarative language, the only concepts available to the tester are those that follow from ordinary graphs. Fortunately, most software is developed in imperative languages; so testers usually have the full power of directed graphs at their disposal.

The next series of definitions roughly parallels the ones for ordinary graphs. We modify our now familiar continuing example to the one shown in Figure 4.2.

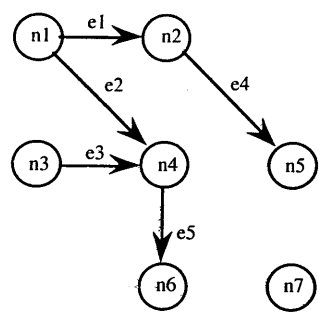

Figure 4.2 A directed graph.

We have the same node set V = {n_1, n_2, n_3, n_4, n_5, n_6, n_7}, and the edge set appears to be the same: E = {e_1, e_2, e_3, e_4, e_5}. The difference is that the edges are now ordered pairs of nodes in V:

$$E = \{<n_1, n_2>, <n_1, n_4>, <n_3, n_4>, <n_2, n_5>, <n_4, n_6>\}$$

4.2.1 Indegrees and Outdegrees

The degree of a node in an ordinary graph is refined to reflect direction, as follows:

Definition

The indegree of a node in a directed graph is the number of distinct edges that have the node as a terminal node. We write indeg(n) for the indegree of node n.

The outdegree of a node in a directed graph is the number of distinct edges that have the node as a start point. We write outdeg(n) for the outdegree of node n.

The nodes in the digraph in Figure 4.2 have the following indegrees and outdegrees:

$indeg(n_1) = 0$ $outdeg(n_1) = 2$
$indeg(n_2) = 1$ $outdeg(n_2) = 1$
$indeg(n_3) = 0$ $outdeg(n_3) = 1$
$indeg(n_4) = 2$ $outdeg(n_4) = 1$
$indeg(n_5) = 1$ $outdeg(n_5) = 0$
$indeg(n_6) = 1$ $outdeg(n_6) = 0$
$indeg(n_7) = 0$ $outdeg(n_7) = 0$

Ordinary and directed graphs meet through definitions that relate obvious correspondences, such as: deg(n) = indeg(n) + outdeg(n).

4.2.2 Types of Nodes

The added descriptive power of directed graphs lets us define different kinds of nodes:

Definition

A node with indegree = 0 is a source node.
A node with outdegree = 0 is a sink node.
A node with indegree ≠ 0 and outdegree ≠ 0 is a transfer node.

Source and sink nodes constitute the external boundary of a graph. If we made a directed graph of a context diagram (from a set of dataflow diagrams produced by structured analysis), the external entities would be source and sink nodes.

In our continuing example, n_1, n_3 and n_7 are source nodes: n_5, n_6 and n_7 are sink nodes; and n_2 and n_4 are transfer (also known as interior) nodes. A node that is both a source and a sink node is an isolated node.

4.2.3 Adjacency Matrix of a Directed Graph

As we might expect, the addition of direction to edges changes the definition of the adjacency matrix of a directed graph. (It also changes the incidence matrix, but this matrix is seldom used in conjunction with digraphs.)

Definition

The adjacency matrix of a directed graph D = (V, E) with m nodes is an m × m matrix: A = (a(i, j)) where a(i, j) is a 1 if and only if there is an edge from node i and node j; otherwise, the element is 0.

The adjacency matrix of a directed graph is not necessarily symmetric. A row sum is the outdegree of the node; a column sum is the indegree of a node. The adjacency matrix of our continuing example is:

	n_1	n_2	n_3	n_4	n_5	n_6	n_7
n_1	0	1	0	1	0	0	0
n_2	0	0	0	0	1	0	0
n_3	0	0	0	1	0	0	0
n_4	0	0	0	0	0	1	0
n_5	0	0	0	0	0	0	0
n_6	0	0	0	0	0	0	0
n_7	0	0	0	0	0	0	0

One common use of directed graphs is to record family relationships, in which siblings, cousins, and so on are connected by an ancestor: and parents, grandparents, and so on are connected by a descendant. Entries in powers of the adjacency matrix now show existence of directed paths.

4.2.4 Paths and Semipaths

Direction permits a more precise meaning to paths that connect nodes in a directed graph. As a handy analogy, you may think in terms of one-way and two-way streets.

Definition

A (directed) path is a sequence of edges such that, for any adjacent pair of edges e_i, e_j, in the sequence, the terminal node of the first edge is the initial node of the second edge.
A cycle is a directed path that begins and ends at the same node.

A (directed) semipath is a sequence of edges such that, for at least one adjacent pair of edges e_i, e_j in the sequence, the initial node of the first edge is the initial node of the second edge or the terminal node of the first edge is the terminal node of the second edge.

Directed paths are sometimes called chains; we will use this concept in Chapter 9. Our continuing example contains the following paths and semipaths (not all are listed):

A path from n_1 to n_6
A semipath between n_1 and n_3
A semipath between n_2 and n_4
A semipath between n_5 and n_6

4.2.5 Reachability Matrix

When we model an application with a digraph, we often ask questions that deal with paths that let us reach (or "get to") certain nodes. This is an extremely useful capability and is made possible by the reachability matrix of a digraph.

Definition

The reachability matrix of a directed graph D = (V, E) with m nodes is an m × m matrix R = (r(i, j)), where r(i, j) is a 1 if and only if there is a path from node i and node j, otherwise the element is 0.

The reachability matrix of a directed graph D can be calculated from the adjacency matrix A as follows:

$$R = I + A + A^2 + A^3 + \ldots + A^k$$

where k is the length of the longest path in D, and I is the identity matrix. The reachability matrix for our continuing example is:

	n_1	n_2	n_3	n_4	n_5	n_6	n_7
n_1	0	1	0	1	1	1	0
n_2	0	0	0	0	1	0	0
n_3	0	0	0	1	0	1	0
n_4	0	0	0	0	0	1	0
n_5	0	0	0	0	0	0	0
n_6	0	0	0	0	0	0	0
n_7	0	0	0	0	0	0	0

The reachability matrix tells us that nodes n_2, n_4, n_5, and n_6 can be reached from n_1, node n_5 can be reached from n_2, and so on.

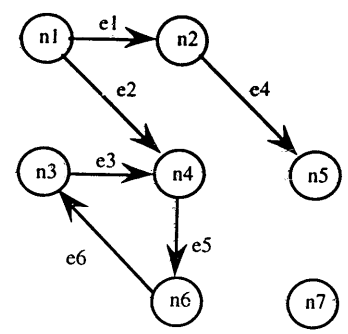

Figure 4.3 A directed graph with a cycle.

4.2.6 n-Connectedness

Connectedness of ordinary graphs extends to a rich, highly explanatory concept for digraphs.

Definition

Two nodes n_i and n_j in a directed graph are:

0-connected iff no path exists between n_i and n_j
1-connected iff a semi-path but no path exists between n_i and n_j
2-connected iff a path exists between n_i and n_j
3-connected iff a path goes from n_i to n_j and a path goes from n_j to n_i

No other degrees of connectedness exist.

We need to modify our continuing example to show 3-connectedness. The change is the addition of a new edge e_6 from n_6 to n_3, so the graph contains a cycle.

With this change, we have the following instances of n-connectivity in Figure 4.3 (not all are listed):

n_1 and n_7 are 0-connected
n_2 and n_6 are 1-connected
n_1 and n_6 are 2-connected
n_3 and n_6 are 3-connected

In terms of one-way streets, you cannot get from n_2 to n_6.

4.2.7 Strong Components

The analogy continues. We get two equivalence relations from n-connectedness: 1-connectedness yields what we might call "weak connection," and this in turn yields weak components. (These turn out to be the same as we had for ordinary graphs, which is what should happen, because 1-connectedness effectively ignores

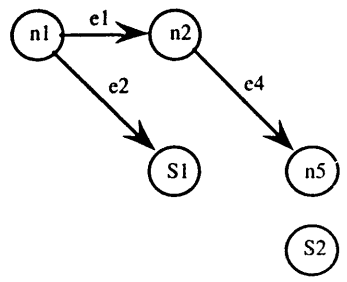

Figure 4.4 Condensation graph of the digraph in Figure 4.3.

direction.) The second equivalence relation, based on 3-connectedness is more interesting. As before, the equivalence relation induces a partition on the node set of a digraph, but the condensation graph is quite different. Nodes that previously were 0-, 1-, or 2-connected remain so. The 3-connected nodes become the strong components.

Definition

A strong component of a directed graph is a maximal set of 3-connected nodes.

In our amended example, the strong components are the sets $\{n_3, n_4, n_6\}$ and $\{n_7\}$. The condensation graph for our amended example is shown in Figure 4.4.

Strong components let us simplify by removing loops and isolated nodes. Although this is not as dramatic as the simplification we had in ordinary graphs, it does solve a major testing problem. Notice that the condensation graph of a digraph will never contain a loop. (If it did, the loop would have been condensed by the maximal aspect of the partition.) These graphs have a special name: directed acyclic graphs, sometimes written as DAG.

Many papers on structured testing make quite a point of showing how relatively simple programs can have millions of distinct execution paths. The intent of these discussions is to convince us that exhaustive testing is exactly that — exhaustive. The large number of execution paths comes from nested loops. Condensation graphs eliminate loops (or at least condense them down to a single node), therefore, we can use this as a strategy to simplify situations that otherwise are computationally untenable.

4.3 Graphs for Testing

We conclude this chapter with four special graphs that are widely used for testing. The first of these, the program graph, is used primarily at the unit testing level. The other two, finite state machines, state charts, and Petri nets, are best used to describe system-level behavior, although they can be used at lower levels of testing.

4.3.1 Program Graphs

At the beginning of this chapter, we made a point of avoiding interpretations on the graph theory definitions to preserve latitude in later applications. Here, we give the most common use of graph theory in software testing — the program graph. To better connect with existing testing literature, the traditional definition is given, followed by an improved definition.

Definition

Given a program written in an imperative programming language, its program graph is a directed graph in which:

1. (Traditional Definition)
   Nodes are program statements, and edges represent flow of control (there is an edge from node i to node j iff the statement corresponding to node j can be executed immediately after the statement corresponding to node i).
2. (Improved Definition)
   Nodes are either entire statements or fragments of a statement, and edges represent flow of control (there is an edge from node i to node j iff the statement or statement fragment corresponding to node j can be executed immediately after the statement or statement fragment corresponding to node i).

It is cumbersome to always say "statement or statement fragment," so we adopt the convention that a statement fragment can be an entire statement. The directed graph formulation of a program enables a very precise description of testing aspects of the program. For one thing, a very satisfying connection exists between this formulation and the precepts of structured programming. The basic structured programming constructs (sequence, selection, and repetition) all have clear, directed graphs, as shown in Figure 4.5.

When these constructs are used in a structured program, the corresponding graphs are either nested or concatenated. The single entrance and single exit criteria result in unique source and sink nodes in the program graph. In fact, the old (nonstructured) "spaghetti code" resulted in very complex program graphs. GOTO statements, for example, introduce edges; and when these are used to branch into or out of loops, the resulting program graphs become even more complex. One of the pioneering analysts of this is Thomas McCabe, who popularized the cyclomatic number of a graph as an indicator of program complexity (McCabe, 1976). When a program executes, the statements that execute comprise a path in the program graph. Loops and decisions greatly increase the number of possible paths and therefore similarly increase the need for testing.

One of the problems with program graphs is how to treat nonexecutable statements such as comments and data declaration statements. The simplest answer is to ignore them. A second problem has to do with the difference between topologically possible and semantically feasible paths. We will discuss this in more detail in Part III.

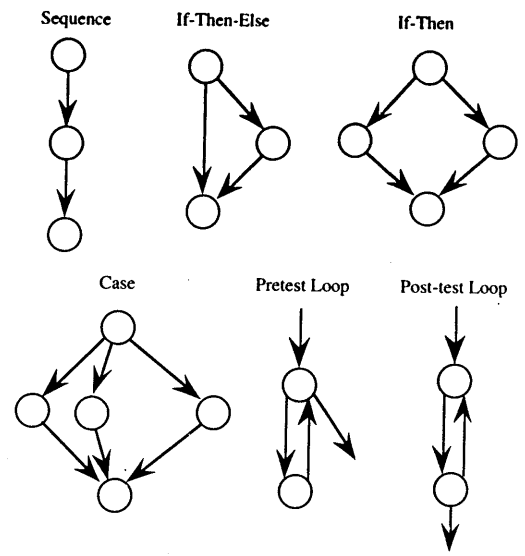

Figure 4.5 Digraphs of the structured programming constructs.

4.3.2 Finite State Machines

Finite state machines have become a fairly standard notation for requirements specification. All the real-time extensions of structured analysis use some form of finite state machine, and nearly all forms of object-oriented analyses require them. A finite state machine is a directed graph in which states are nodes and transitions are edges. Source and sink states become initial and terminal nodes, paths are modeled as paths, and so on. Most finite state machine notations add information to the edges (transitions) to indicate the cause of the transition and actions that occur as a result of the transition.

Figure 4.6 is a finite state machine for the personal identification number (PIN) try portion of the simple automated teller machine (SATM) system. This machine contains five states (Idle, Awaiting First PIN Try, and so on) and eight transitions, which are shown as edges. The labels on the transitions follow a convention that the "numerator" is the event that causes the transition, and the "denominator" is the action that is associated with the transition. The events are mandatory — transitions do not just happen, but the actions are optional. Finite state machines are simple ways to represent situations in which a variety of events may occur, and their occurrences have different consequences. In the PIN entry portion of the SATM system, for example, a customer has three chances to enter the correct PIN digits. If the correct PIN is entered on the first try, the SATM system exhibits the output action of displaying screen 5 (which invites the customer to choose a transaction type). If an incorrect PIN is entered, the machine goes to a different state, one in which it awaits a second PIN attempt. Notice that the same events and actions occur on the transitions from the Awaiting Second PIN Try state. This is the way finite state machines can keep a history of past events.

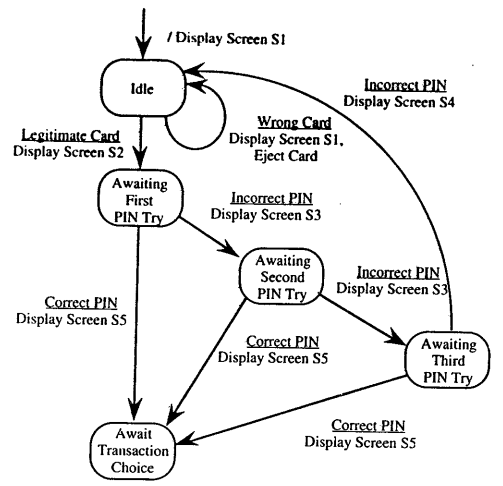

Figure 4.6 Finite state machine for PIN tries.

Finite state machines can be executed, but a few conventions are needed first. One is the notion of the active state. We speak of a system being "in" a certain state; when the system is modeled as a finite state machine, the active state refers to the state "we are in." Another convention is that finite state machines may have an initial state, which is the state that is active when a finite state machine is first entered. (The Idle state is the initial state in Figure 4.6; this is indicated by the transition that comes from nowhere. Final states are recognized by the absence of outgoing transitions.) Exactly one state can be active at any time. We also think of transitions as instantaneous occurrences, and the events that cause transitions also occur one at a time. To execute a finite state machine, we start with an initial state and provide a sequence of events that causes state transitions. As each event occurs, the transition changes the active state and a new event occurs. In this way, a sequence of events selects a path of states (or equivalently, of transitions) through the machine.

4.3.3 Petri Nets

Petri nets were the topic of Carl Adam Petri's Ph.D. dissertation in 1963; today, they are the accepted model for protocols and other applications involving concurrency and distributed processing. Petri nets are a special form of directed graph: a bipartite directed graph. (A bipartite graph has two sets of nodes, V_1 and V_2, and a set of edges E, with the restriction that every edge has its initial node on one of the sets V_1, V_2, and its terminal node in the other set.) In a Petri net, one of the sets is referred to as "places," and the other is referred to as "transitions." These sets are usually denoted as P and T, respectively. Places are inputs to and outputs of transitions; the input and output relationships are functions, and they are usually denoted as In and Out, as in the following definition.

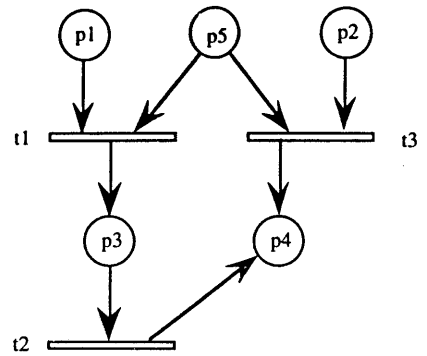

Figure 4.7 A Petri net.

Definition

A Petri net is a bipartite directed graph (P, T, In, Out), in which P and T are disjoint sets of nodes, and In and Out are sets of edges, where In $\subseteq$ P $\times$ T, and Out $\subseteq$ T $\times$ P.

For the sample Petri net in Figure 4.7, the sets P, T, In, and Out are:

$$P = \{p_1, p_2, p_3, p_4, p_5\}$$
$$T = \{t_1, t_2, t_3\}$$
$$In = \{<p_1, t_1>, <p_5, t_1>, <p_5, t_3>, <p_2, t_3>, <p_3, t_2>\}$$
$$Out = \{<t_1, p_3>, <t_2, p_4>, <t_3, p_4>\}$$

Petri nets are executable in more interesting ways than finite state machines. The next few definitions lead us to Petri net execution.

Definition

A marked Petri net is a 5-tuple (P, T, In, Out, M) in which (P, T, In, Out) is a Petri net and M is a set of mappings of places to positive integers.

The set M is called the marking set of the Petri net. Elements of M are n-tuples, where n is the number of places in the set P. For the Petri net in Figure 4.7, the set M contains elements of the form $<n_1, n_2, n_3, n_4, n_5>$, where the n's are the integers associated with the respective places. The number associated with a place refers to the number of tokens that are said to be "in" the place. Tokens are abstractions that can be interpreted in modeling situations. For example, tokens might refer to the number of times a place has been used, or the number of things in a place, or whether the place is true. Figure 4.8 shows a marked Petri net.

The marking tuple for the marked Petri net in Figure 4.8 is <1, 1, 0, 2, 0>. We need the concept of tokens to make two essential definitions.

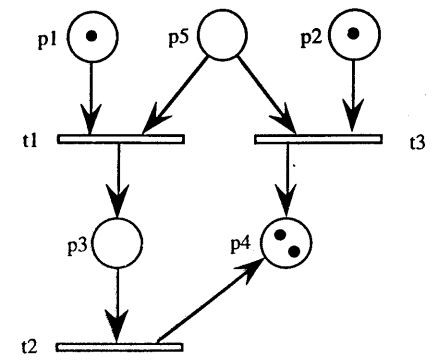

Figure 4.8 A marked Petri net.

Definition

A transition in a Petri net is enabled if at least one token is in each of its input places.

No enabled transitions are in the marked Petri net in Figure 4.8. If we put a token in place p_3, then transition t_2 would be enabled.

Definition

When an enabled Petri net transition fires, one token is removed from each of its input places and one token is added to each of its output places.

In Figure 4.9, transition t_2 is enabled in the upper net and has been fired in the lower net.

The marking set for the net in Figure 4.9 contains two tuples — the first shows the net when t_2 is enabled, and the second shows the net after t_2 has fired.

$$M = \{<1, 1, 1, 2, 0>, <1, 1, 0, 3, 0>\}$$

Tokens may be created or destroyed by transition firings. Under special conditions, the total number of tokens in a net never changes; such nets are called conservative. We usually do not worry about token conservation. Markings let us execute Petri nets in much the same way that we execute finite state machines. (It turns out that finite state machines are a special case of Petri nets.) Suppose we had a different marking of the net in Figure 4.7; in this new marking, places p_1, p_2, and p_5 are all marked. With such a marking, transitions t_1 and t_3 are both enabled. If we choose to fire transition t_1, the token in place p_5 is removed and t_3 is no longer enabled. Similarly, if we choose to fire t_3, we disable t_1. This pattern is known as Petri net conflict. More specificly, we say that transitions t_1 and t_3 are in conflict with respect to place p_5. Petri net conflict exhibits an interesting form of interaction between two transitions; we will revisit this (and other) interactions in Chapter 15.

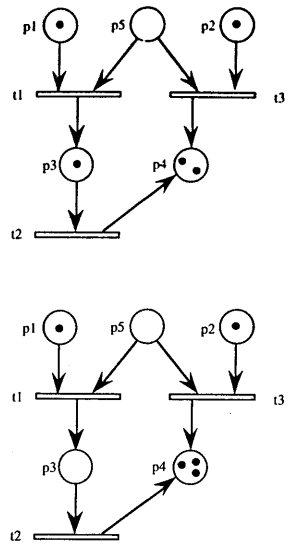

Figure 4.9 Before and after firing t_2.

4.3.4 Event-Driven Petri Nets

Basic Petri nets need two slight enhancements to become event-driven Petri Nets (EDPNs). The first enables them to express more closely event-driven systems, and the second deals with Petri net markings that express event quiescence, an important notion in object-oriented applications. Taken together, these extensions result in an effective, operational view of software requirements; elsewhere they are known as OSD nets (for Operational Software Development) (Jorgensen, 1989).

Definition

An EDPN is a tripartite-directed graph (P, D, S, In, Out) composed of three sets of nodes, P, D, and S, and two mappings, In and Out, where:

P is a set of port events
D is a set of data places
S is a set of transitions
In is a set of ordered pairs from $(P \cup D) \times S$
Out is a set of ordered pairs from $S \times (P \cup D)$

EDPNs express four of the five basic system constructs defined in Chapter 14; only devices are missing. The set S of transitions corresponds to ordinary Petri net transitions, which are interpreted as actions. Two kinds of places, port events and data places, are inputs to or outputs of transitions in S as defined by the

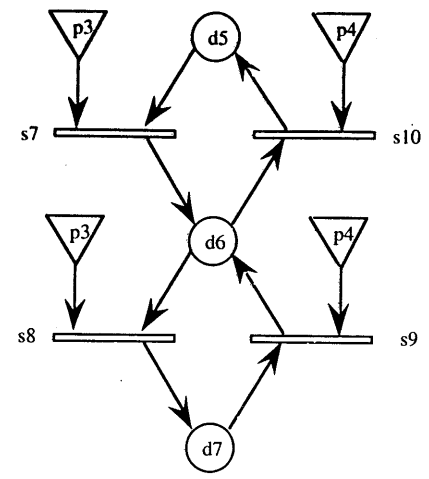

Figure 4.10 An EDPN.

input and output functions In and Out. A thread is a sequence of transitions in S, so we can always construct the inputs and outputs of a thread from the inputs and outputs of the transitions in the thread. EDPNs are graphically represented in much the same way as ordinary Petri nets; the only difference is the use of triangles for port event places. The EDPN in Figure 4.10 has four transitions, s_7, s_8, s_9, and s_{10}; two port input events, p_3 and p_4; and three data places, d_5, d_6, and d_7. It does not have port output events.

This is the EDPN that corresponds to the finite state machine developed for the dial portion of the Saturn windshield wiper system in Chapter 15 (see Figure 15.10). The components of this net are described in Table 4.1.

Markings for an EDPN are more complicated because we want to be able to deal with event quiescence.

Table 4.1 EDPN Elements in Figure 4.10

Element	*Type*	*Description*
p_3	port input event	rotate dial clockwise
p_4	port input event	rotate dial counterclockwise
d_5	data place	dial at position 1
d_6	data place	dial at position 2
d_7	data place	dial at position 3
s_7	transition	state transition: d_5 to d_6
s_8	transition	state transition: d_6 to d_7
s_9	transition	state transition: d_7 to d_6
s_{10}	transition	state transition: d_6 to d_5

Table 4.2 A Marking of the EDPN in Figure 4.10

tuple	*(p_3, p_4, d_5, d_6, d_7)*	*Description*
m_1	(0, 0, 1, 0, 0)	initial condition, in state d_5
m_2	(1, 0, 1, 0, 0)	p_3 occurs
m_3	(0, 0, 0, 1, 0)	in state d_6
m_4	(1, 0, 0, 1, 0)	p_3 occurs
m_5	(0, 0, 0, 0, 1)	in state d_7
m_6	(0, 1, 0, 0, 1)	p_4 occurs
m_7	(0, 0, 0, 1, 0)	in state d_6

Definition

A marking M of an EDPN (P, D, S, In, Out) is a sequence M = <m1, m2, ...> of p-tuples, where p = k + n, and k and n are the number of elements in the sets P and D, and individual entries in a p-tuple indicate the number of tokens in the event or data place.

By convention, we will put the data places first, followed by the input event places and then the output event places. An EDPN may have any number of markings; each corresponds to an execution of the net. Table 4.2 shows a sample marking of the EDPN in Figure 4.10.

The rules for transition enabling and firing in an EDPN are exact analogs of those for traditional Petri nets; a transition is enabled if there is at least one token in each input place; and when an enabled transition fires, one token is removed from each of its input places, and one token is placed in each of its output places. Table 4.3 follows the marking sequence given in Table 4.2, showing which transitions are enabled and fired.

The important difference between EDPNs and traditional Petri nets is that event quiescence can be broken by creating a token in a port input event place. In traditional Petri nets, when no transition is enabled, we say that the net is deadlocked. In EDPNs, when no transition is enabled, the net is at a point of event quiescence. (Of course, if no event occurs, this is the same as deadlock.) Event quiescence occurs four times in the thread in Table 4.3; at m_1, m_3, m_5, and m_7.

Table 4.3 Enabled and Fired Transitions in Table 4.2

tuple	*(p_3, p_4, d_5, d_6, d_7)*	*Description*
m_1	(0, 0, 1, 0, 0)	nothing enabled
m_2	(1, 0, 1, 0, 0)	s_7 enabled; s_7 fired
m_3	(0, 0, 0, 1, 0)	nothing enabled
m_4	(1, 0, 0, 1, 0)	s_8 enabled; s_8 fired
m_5	(0, 0, 0, 0, 1)	nothing enabled
m_6	(0, 1, 0, 0, 1)	s_9 enabled; s_9 fired
m_7	(0, 0, 0, 1, 0)	nothing enabled

The individual members in a marking can be thought of as snapshots of the executing EDPN at discrete points in time; these members are alternatively referred to as time steps, p-tuples, or marking vectors. This lets us think of time as an ordering that allows us to recognize "before" and "after." If we attach instantaneous time as an attribute of port events, data places, and transitions, we obtain a much clearer picture of thread behavior. One awkward part to this is how to treat tokens in a port output event place. Port output places always have outdegree = 0; in an ordinary Petri net, tokens cannot be removed from a place with a zero outdegree. If the tokens in a port output event place persist, this suggests that the event occurs indefinitely. Here again, the time attributes resolve the confusion; this time we need a duration of the marked output event. (Another possibility is to remove tokens from a marked output event place after one time step; this works reasonably well.)

4.3.5 StateCharts

David Harel had two goals when he developed the StateChart notation: he wanted to devise a visual notation that combined the ability of Venn diagrams to express hierarchy and the ability of directed graphs to express connectedness (Harel, 1998). Taken together, these capabilities provide an elegant answer to the "state explosion" problem of ordinary finite state machines. The result is a highly sophisticated and very precise notation that is supported by commercially available CASE tools, notably the StateMate system from i-Logix. StateCharts are now the control model of choice for the unified modeling language (UML) from Rational Corp. (See www.rational.com for more details.)

Harel uses the methodology neutral term "blob" to describe the basic building block of a StateChart. Blobs can contain other blobs in the same way that Venn diagrams show set containment. Blobs can also be connected to other blobs with edges in the same way that nodes in a directed graph are connected. In Figure 4.11, blob A contains two blobs (B and C), and they are connected by edges. Blob A is also connected to blob D by an edge.

As Harel intends, we can interpret blobs as states, and edges as transitions. The full StateChart system supports an elaborate language that defines how and when transitions occur (their training course runs for a full week, so this section is a highly simplified introduction). StateCharts are executable in a much more elaborate way than ordinary finite state machines. Executing a StateChart requires a notion similar to that of Petri net markings. The "initial state" of a StateChart is indicated by an edge that has no source state. When states are nested within

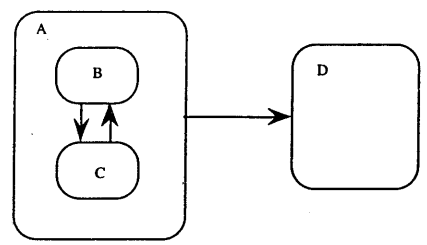

Figure 4.11 Blobs in a StateChart.

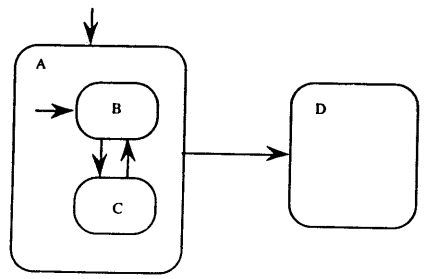

Figure 4.12 Initial states in a StateChart.

other states, the same indication is used to show the lower level initial state. In Figure 4.12, state A is the initial state; and when it is entered, state B is also entered at the lower level. When a state is entered, we can think of it as active in a way analogous to a marked place in a Petri net. (The StateChart tool used colors to show which states are active, and this is equivalent to marking places in a Petri net.) A subtlety exists in Figure 4.12, the transition from state A to state D seems ambiguous at first because it has no apparent recognition of states B and C. The convention is that edges must start and end on the outline of a state. If a state contains substates, as state A does, the edge "refers" to all substates. Thus, the edge from A to D means that the transition can occur either from state B or from state C. If we had an edge from State D to state A, as in Figure 4.13, the fact that state B is indicated as the initial state means that the transition is really from state D to state B. This convention greatly reduces the tendency of finite state machines to look like "spaghetti code."

The last aspect of StateCharts we will discuss is the notion of concurrent StateCharts. The dotted line in state D (see Figure 4.14) is used to show that state D really refers to two concurrent states. E and F. (Harel's convention is to move the state label of D to a rectangular tag on the perimeter of the state.) Although not shown here, we can think of E and F as parallel machines that execute concurrently. Because the edge from state A terminates on the perimeter of state D, when that transition occurs, both machines E and F are active (or marked, in the Petri net sense).

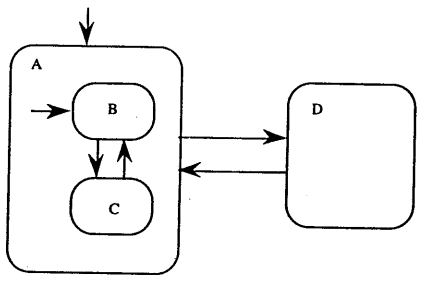

Figure 4.13 Default entry into substates.

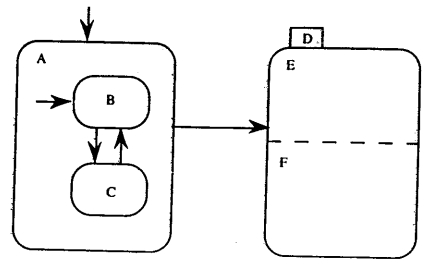

Figure 4.14 Concurrent states.

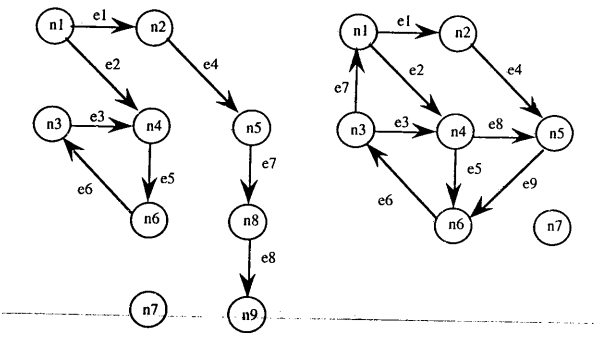

Figure 4.15

Reference

Harel, David, On visual formalisms, *Communications of the ACM,* Vol. 31, No. 5, pp. 514–530, May, 1988.

Exercises

1. Propose a definition for the length of a path in a graph.
2. What loop(s) is/are created if an edge is added between nodes n_5 and n_6 in the graph in Figure 4.1?
3. Convince yourself that 3-connectedness is an equivalence relation on the nodes of a digraph.
4. Compute the cyclomatic complexity for each of the structured programming constructs in Figure 4.5.
5. The digraphs in Figure 4.15 were obtained by adding nodes and edges to the digraph in Figure 4.3. Compute the cyclomatic complexity of each new digraph, and explain how the changes affected the complexity.

6. Suppose we make a graph in which nodes are people and edges correspond to some form of social interaction, such as "talks to" or "socializes with." Find graph theory concepts that correspond to social concepts such as popularity, cliques, and hermits.

II FUNCTIONAL TESTING

Chapter 5

Boundary Value Testing

In Chapter 3, we saw that a function maps values from one set (its domain) to values in another set (its range) and that the domain and range can be cross products of other sets. Any program can be considered to be a function in the sense that program inputs form its domain and program outputs form its range. Input domain testing is the best-known functional testing technique. In this and the next two chapters, we examine how to use knowledge of the functional nature of a program to identify test cases for the program. Historically, this form of testing has focused on the input domain; but it is often a good supplement to apply many of these techniques to develop range-based test cases.

5.1 Boundary Value Analysis

For the sake of comprehensible drawings, the discussion relates to a function, F, of two variables x_1 and x_2. When the function F is implemented as a program, the input variables x_1 and x_2 will have some (possibly unstated) boundaries:

$$a \leq x_1 \leq b$$
$$c \leq x_2 \leq d$$

Unfortunately, the intervals [a, b] and [c, d] are referred to as the ranges of x_1 and x_2, so right away we have an overloaded term. The intended meaning will always be clear from its context. Strongly typed languages (such as Ada® and Pascal) permit explicit definition of such variable ranges. In fact, part of the historical reason for strong typing was to prevent programmers from making the kinds of errors that result in faults that are easily revealed by boundary value testing. Other languages (such as COBOL, FORTRAN, and C) are not strongly typed, so boundary value testing is more appropriate for programs coded in such languages. The input space (domain) of our function F is shown in Figure 5.1. Any point within the shaded rectangle is a legitimate input to the function F.

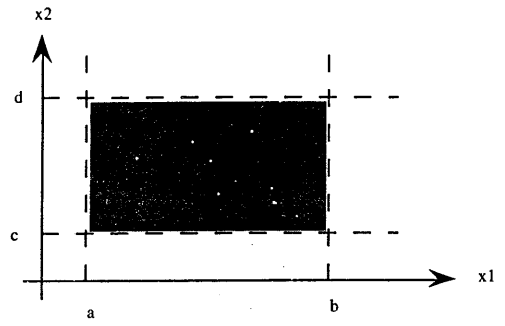

Figure 5.1 Input domain of a function of two variables.

Boundary value analysis focuses on the boundary of the input space to identify test cases. The rationale behind boundary value testing is that errors tend to occur near the extreme values of an input variable. Loop conditions, for example, may test for < when they should test for ≤, and counters often are "off by one." The desktop publishing program in which this manuscript is typed has an interesting boundary value problem. Two modes of textual display are used: one indicates new pages by a dotted line, and the other displays a page image showing where the text is placed on the page. If the cursor is at the last line of a page and new text is added, an anomaly occurs: in the first mode, the new line(s) simply appear, and the dotted line (page break) is adjusted. In the page display mode, however, the new text is lost — it does not appear on either page. The U.S. Army (CECOM) made a study of its software and found that a surprising portion of faults turned out to be boundary value faults.

The basic idea of boundary value analysis is to use input variable values at their minimum, just above the minimum, a nominal value, just below their maximum, and at their maximum. A commercially available testing tool (named T) generates such test cases for a properly specified program. This tool has been successfully integrated with two popular front-end CASE tools (Teamwork from Cadre Systems, and Software through Pictures from Aonix). The T tool refers to these values as min, min+, nom, max-, and max. We will use these conventions here.

The next part of boundary value analysis is based on a critical assumption; it is known as the "single fault" assumption in reliability theory. This says that failures are only rarely the result of the simultaneous occurrence of two (or more) faults. Thus, the boundary value analysis test cases are obtained by holding the values of all but one variable at their nominal values, and letting that variable assume its extreme values. The boundary value analysis test cases for our function F of two variables (illustrated in Figure 5.2) are:

$$\{\langle x_{1nom}, x_{2min}\rangle, \langle x_{1nom}, x_{2min+}\rangle, \langle x_{1nom}, x_{2nom}\rangle, \langle x_{1nom}, x_{2max-}\rangle, \langle x_{1nom}, x_{2max}\rangle, \langle x_{1min}, x_{2nom}\rangle, \langle x_{1min+}, x_{2nom}\rangle, \langle x_{1nom}, x_{2nom}\rangle, \langle x_{1max-}, x_{2nom}\rangle, \langle x_{1max}, x_{2nom}\rangle\}$$

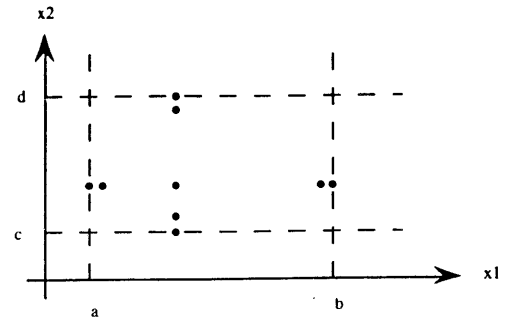

Figure 5.2 Boundary value analysis test cases for a function of two variables.

5.1.1 Generalizing Boundary Value Analysis

The basic boundary value analysis technique can be generalized in two ways: by the number of variables and by the kinds of ranges. Generalizing the number of variables is easy: if we have a function of n variables, we hold all but one at the nominal values and let the remaining variable assume the min, min+, nom, max-, and max values, repeating this for each variable. Thus, for a function of n variables, boundary value analysis yields 4n + 1 test cases.

Generalizing ranges depends on the nature (or more precisely, the type) of the variables themselves. In the NextDate function, for example, we have variables for the month, the day, and the year. In a FORTRAN-like language, we would most likely encode these, so that January would correspond to 1, February to 2, and so on. In a language that supports user-defined types (like Pascal or Ada), we could define the variable month as an enumerated type {Jan., Feb., ..., Dec.}. Either way, the values for min, min+, nom, max-, and max are clear from the context. When a variable has discrete, bounded values, as the variables in the commission problem have, the min, min+, nom, max-, and max are also easily determined. When no explicit bounds are present, as in the triangle problem, we usually have to create "artificial" bounds. The lower bound of side lengths is clearly 1 (a negative side length is silly): but what might we do for an upper bound? By default, the largest representable integer (called MAXINT in some languages) is one possibility; or we might impose an arbitrary upper limit such as 200 or 2000.

Boundary value analysis does not make much sense for Boolean variables; the extreme values are TRUE and FALSE, but no clear choice is available for the remaining three. We will see in Chapter 7 that Boolean variables lend themselves to decision table-based testing. Logical variables also present a problem for boundary value analysis. In the ATM example, a customer's PIN is a logical variable, as is the transaction type (deposit, withdrawal, or inquiry). We could "go through the motions" of boundary value analysis testing for such variables, but the exercise is not very satisfying to the "tester's intuition."

5.1.2 Limitations of Boundary Value Analysis

Boundary value analysis works well when the program to be tested is a function of several independent variables that represent bounded physical quantities. The key words here are independent and physical quantities. A quick look at the boundary value analysis test cases for NextDate (in Section 5.5) shows them to be inadequate. Very little stress occurs on February and on leap years, for example. The real problem here is that interesting dependencies exist among the month, day, and year variables. Boundary value analysis presumes the variables to be truly independent. Even so, boundary value analysis happens to catch end-of-month and end-of-year faults. Boundary value analysis test cases are derived from the extrema of bounded, independent variables that refer to physical quantities, with no consideration of the nature of the function, nor of the semantic meaning of the variables. We see boundary value analysis test cases to be rudimentary because they are obtained with very little insight and imagination. As with so many things, you get what you pay for.

The physical quantity criterion is equally important. When a variable refers to a physical quantity, such as temperature, pressure, air speed, angle of attack, load, and so forth, physical boundaries can be extremely important. (In an interesting example of this, Sky Harbor International Airport in Phoenix had to close on June 26, 1992, because the air temperature was 122°F. Aircraft pilots were unable to make certain instrument settings before take-off: the instruments could only accept a maximum air temperature of 120°F.) In another case, a medical analysis system uses stepper motors to position a carousel of samples to be analyzed. It turns out that the mechanics of moving the carousel back to the starting cell often causes the robot arm to miss the first cell.

As an example of logical (versus physical) variables, we might look at PINs or telephone numbers. It is hard to imagine what faults might be revealed by PINs of 0000, 0001, 5000, 9998, and 9999.

5.2 Robustness Testing

Robustness testing is a simple extension of boundary value analysis: in addition to the five boundary value analysis values of a variable, we see what happens when the extrema are exceeded with a value slightly greater than the maximum (max+) and a value slightly less than the minimum (min-). Robustness test cases for our continuing example are shown in Figure 5.3.

Most of the discussion of boundary value analysis applies directly to robustness testing, especially the generalizations and limitations. The most interesting part of robustness testing is not with the inputs, but with the expected outputs. What happens when a physical quantity exceeds its maximum? If it is the angle of attack of an airplane wing, the aircraft might stall. If it is the load capacity of a public elevator, we hope nothing special would happen. If it is a date, like May 32, we would expect an error message. The main value of robustness testing is that it forces attention on exception handling. With strongly typed languages, robustness testing may be very awkward. In Pascal, for example, if a variable is defined to be within a certain range, values outside that range result in run-time errors that abort normal execution. This raises an interesting question of implementation philosophy: Is it better to perform explicit range checking and use exception handling to deal with "robust values," or is it better to stay with strong typing? The exception handling choice mandates robustness testing.

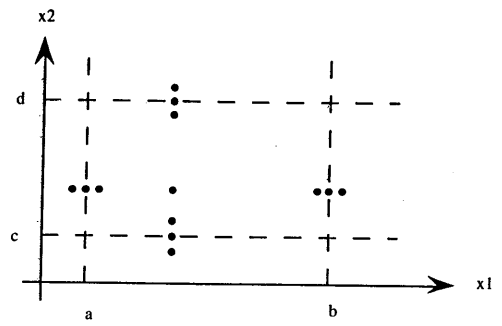

Figure 5.3 Robustness test cases for a function of two variables.

5.3 Worst-Case Testing

Boundary value analysis, as we said earlier, makes the single fault assumption of reliability theory. Rejecting this assumption means that we are interested in what happens when more than one variable has an extreme value. In electronic circuit analysis, this is called "worst-case analysis"; we use that idea here to generate worst-case test cases. For each variable, we start with the five-element set that contains the min, min+, nom, max-, and max values. We then take the Cartesian product (see Chapter 3) of these sets to generate test cases. The result of the two-variable version of this is shown in Figure 5.4.

Worst-case testing is clearly more thorough in the sense that boundary value analysis test cases are a proper subset of worst-case test cases. It also represents

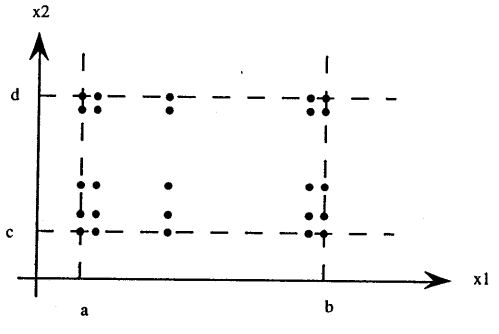

Figure 5.4 Worst-case test cases for a function of two variables.

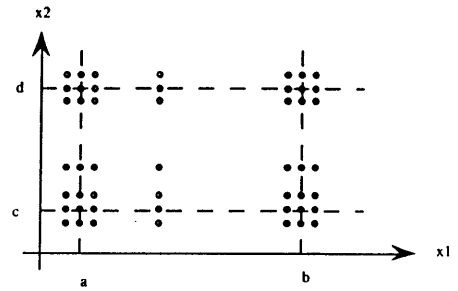

Figure 5.5 Robust worst-case test cases for a function of two variables.

much more effort: worst-case testing for a function of n variables generates 5_n test cases, as opposed to 4n + 1 test cases for boundary value analysis.

Worst-case testing follows the generalization pattern we saw for boundary value analysis. It also has the same limitations, particularly those related to independence. Probably the best application for worst-case testing is where physical variables have numerous interactions, and where failure of the function is extremely costly. For really paranoid testing, we could go to robust worst-case testing. This involves the Cartesian product of the seven-element sets we used in robustness testing. Figure 5.5 shows the robust worst-case test cases for our two variable function.

5.4 Special Value Testing

Special value testing is probably the most widely practiced form of functional testing. It also is the most intuitive and the least uniform. Special value testing occurs when a tester uses his or her domain knowledge, experience with similar programs, and information about "soft spots" to devise test cases. We might also call this "ad hoc testing" or "seat-of-the-pants (or –skirt)" testing. No guidelines are used other than to use "best engineering judgment." As a result, special value testing is very dependent on the abilities of the tester.

Despite all the apparent negatives, special value testing can be very useful. In the next section, you will find test cases generated by the methods we just discussed for three of our examples (not the ATM system). If you look carefully at these, especially for the NextDate function, you find that none is very satisfactory. If an interested tester defined special value test cases for NextDate, we would see several test cases involving February 28, February 29, and leap years. Even though special value testing is highly subjective, it often results in a set of test cases that is more effective in revealing faults than the test sets generated by the other methods we have studied — testimony to the craft of software testing.

5.5 Examples

Each of the three continuing examples is a function of three variables. Printing all the test cases from all the methods for each problem is very space consuming, so we just have selected examples.

5.5.1 Test Cases for the Triangle Problem

In the problem statement, no conditions are specified on the triangle sides, other than being integers. Obviously, the lower bounds of the ranges are all 1. We arbitrarily take 200 as an upper bound. Table 5.1 contains boundary value test cases using these ranges.

Table 5.1 Boundary Value Analysis Test Cases

Case	*a*	*b*	*c*	*Expected Output*
1	100	100	1	Isosceles
2	100	100	2	Isosceles
3	100	100	100	Equilateral
4	100	100	199	Isosceles
5	100	100	200	Not a Triangle
6	100	1	100	Isosceles
7	100	2	100	Isosceles
8	100	100	100	Equilateral
9	100	199	100	Isosceles
10	100	200	100	Not a Triangle
11	1	100	100	Isosceles
12	2	100	100	Isosceles
13	100	100	100	Equilateral
14	199	100	100	Isosceles
15	200	100	100	Not a Triangle

Table 5.2 Worst-Case Test Cases

Case	*a*	*b*	*c*	*Expected Output*
1	1	1	1	Equilateral
2	1	1	2	Not a Triangle
3	1	1	100	Not a Triangle
4	1	1	199	Not a Triangle
5	1	1	200	Not a Triangle
6	1	2	1	Not a Triangle
7	1	2	2	Isosceles
8	1	2	100	Not a Triangle
9	1	2	199	Not a Triangle
10	1	2	200	Not a Triangle
11	1	100	1	Not a Triangle
12	1	100	2	Not a Triangle
13	1	100	100	Isosceles
14	1	100	199	Not a Triangle
15	1	100	200	Not a Triangle
16	1	199	1	Not a Triangle
17	1	199	2	Not a Triangle
18	1	199	100	Not a Triangle

Table 5.2 Worst-Case Test Cases *(Continued)*

Case	*a*	*b*	*c*	*Expected Output*
19	1	199	199	Isosceles
20	1	199	200	Not a Triangle
21	1	200	1	Not a Triangle
22	1	200	2	Not a Triangle
23	1	200	100	Not a Triangle
24	1	200	199	Not a Triangle
25	1	200	200	Isosceles
26	2	1	1	Not a Triangle
27	2	1	2	Isosceles
28	2	1	100	Not a Triangle
29	2	1	199	Not a Triangle
30	2	1	200	Not a Triangle
31	2	2	1	Isosceles
32	2	2	2	Equilateral
33	2	2	100	Not a Triangle
34	2	2	199	Not a Triangle
35	2	2	200	Not a Triangle
36	2	100	1	Not a Triangle
37	2	100	2	Not a Triangle
38	2	100	100	Isosceles
39	2	100	199	Not a Triangle
40	2	100	200	Not a Triangle
41	2	199	1	Not a Triangle
42	2	199	2	Not a Triangle
43	2	199	100	Not a Triangle
44	2	199	199	Isosceles
45	2	199	200	Scalene
46	2	200	1	Not a Triangle
47	2	200	2	Not a Triangle
48	2	200	100	Not a Triangle
49	2	200	199	Scalene
50	2	200	200	Isosceles
51	100	1	1	Not a Triangle
52	100	1	2	Not a Triangle
53	100	1	100	Isosceles
54	100	1	199	Not a Triangle
55	100	1	200	Not a Triangle
56	100	2	1	Not a Triangle
57	100	2	2	Not a Triangle
58	100	2	100	Isosceles
59	100	2	199	Not a Triangle
60	100	2	200	Not a Triangle
61	100	100	1	Isosceles
62	100	100	2	Isosceles
63	100	100	100	Equilateral

Table 5.2 Worst-Case Test Cases *(Continued)*

Case	*a*	*b*	*c*	*Expected Output*
64	100	100	199	Isosceles
65	100	100	200	Not a Triangle
66	100	199	1	Not a Triangle
67	100	199	2	Not a Triangle
68	100	199	100	Isosceles
69	100	199	199	Isosceles
70	100	199	200	Scalene
71	100	200	1	Not a Triangle
72	100	200	2	Not a Triangle
73	100	200	100	Not a Triangle
74	100	200	199	Scalene
75	100	200	200	Isosceles
76	199	1	1	Not a Triangle
77	199	1	2	Not a Triangle
78	199	1	100	Not a Triangle
79	199	1	199	Isosceles
80	199	1	200	Not a Triangle
81	199	2	1	Not a Triangle
82	199	2	2	Not a Triangle
83	199	2	100	Not a Triangle
84	199	2	199	Isosceles
85	199	2	200	Scalene
86	199	100	1	Not a Triangle
87	199	100	2	Not a Triangle
88	199	100	100	Isosceles
89	199	100	199	Isosceles
90	199	100	200	Scalene
91	199	199	1	Isosceles
92	199	199	2	Isosceles
93	199	199	100	Isosceles
94	199	199	199	Equilateral
95	199	199	200	Isosceles
96	199	200	1	Not a Triangle
97	199	200	2	Scalene
98	199	200	100	Scalene
99	199	200	199	Isosceles
100	199	200	200	Isosceles
101	200	1	1	Not a Triangle
102	200	1	2	Not a Triangle
103	200	1	100	Not a Triangle
104	200	1	199	Not a Triangle
105	200	1	200	Isosceles
106	200	2	1	Not a Triangle
107	200	2	2	Not a Triangle
108	200	2	100	Not a Triangle

Table 5.2 Worst-Case Test Cases *(Continued)*

Case	*a*	*b*	*c*	*Expected Output*
109	200	2	199	Scalene
110	200	2	200	Isosceles
111	200	100	1	Not a Triangle
112	200	100	2	Not a Triangle
113	200	100	100	Not a Triangle
114	200	100	199	Scalene
115	200	100	200	Isosceles
116	200	199	1	Not a Triangle
117	200	199	2	Scalene
118	200	199	100	Scalene
119	200	199	199	Isosceles
120	200	199	200	Isosceles
121	200	200	1	Isosceles
122	200	200	2	Isosceles
123	200	200	100	Isosceles
124	200	200	199	Isosceles
125	200	200	200	Equilateral

5.5.2 Test Cases for the NextDate Function

Table 5.3 Worst-Case Test Cases

Case	*Month*	*Day*	*Year*	*Expected Output*
1	1	1	1812	January 2, 1812
2	1	1	1813	January 2, 1813
3	1	1	1912	January 2, 1912
4	1	1	2011	January 2, 2011
5	1	1	2012	January 2, 2012
6	1	2	1812	January 3, 1812
7	1	2	1813	January 3, 1813
8	1	2	1912	January 3, 1912
9	1	2	2011	January 3, 2011
10	1	2	2012	January 3, 2012
11	1	15	1812	January 16, 1812
12	1	15	1813	January 16, 1813
13	1	15	1912	January 16, 1912
14	1	15	2011	January 16, 2011
15	1	15	2012	January 16, 2012
16	1	30	1812	January 31, 1812
17	1	30	1813	January 31, 1813
18	1	30	1912	January 31, 1912
19	1	30	2011	January 31, 2011
20	1	30	2012	January 31, 2012

Table 5.3 Worst-Case Test Cases *(Continued)*

Case	*Month*	*Day*	*Year*	*Expected Output*
21	1	31	1812	February 1, 1812
22	1	31	1813	February 1, 1813
23	1	31	1912	February 1, 1912
24	1	31	2011	February 1, 2011
25	1	31	2012	February 1, 2012
26	2	1	1812	February 2, 1812
27	2	1	1813	February 2, 1813
28	2	1	1912	February 2, 1912
29	2	1	2011	February 2, 2011
30	2	1	2012	February 2, 2012
31	2	2	1812	February 3, 1812
32	2	2	1813	February 3, 1813
33	2	2	1912	February 3, 1912
34	2	2	2011	February 3, 2011
35	2	2	2012	February 3, 2012
36	2	15	1812	February 16, 1812
37	2	15	1813	February 16, 1813
38	2	15	1912	February 16, 1912
39	2	15	2011	February 16, 2011
40	2	15	2012	February 16, 2012
41	2	30	1812	error
42	2	30	1813	error
43	2	30	1912	error
44	2	30	2011	error
45	2	30	2012	error
46	2	31	1812	error
47	2	31	1813	error
48	2	31	1912	error
49	2	31	2011	error
50	2	31	2012	error
51	6	1	1812	June 1, 1812
52	6	1	1813	June 1, 1813
53	6	1	1912	June 2, 1912
54	6	1	2011	June 2, 2011
55	6	1	2012	June 2, 2012
56	6	2	1812	June 3, 1812
57	6	2	1813	June 3, 1813
58	6	2	1912	June 3, 1912
59	6	2	2011	June 3, 2011
60	6	2	2012	June 3, 2012
61	6	15	1812	June 16, 1812
62	6	15	1813	June 16, 1813
63	6	15	1912	June 16, 1912
64	6	15	2011	June 16, 2011
65	6	15	2012	June 16, 2012

Table 5.3 Worst-Case Test Cases *(Continued)*

Case	Month	Day	Year	Expected Output
66	6	30	1812	July 31, 1812
67	6	30	1813	July 31, 1813
68	6	30	1912	July 31, 1912
69	6	30	2011	July 31, 2011
70	6	30	2012	July 31, 2012
71	6	31	1812	error
72	6	31	1813	error
73	6	31	1912	error
74	6	31	2011	error
75	6	31	2012	error
76	11	1	1812	November 2, 1812
77	11	1	1813	November 2, 1813
78	11	1	1912	November 2, 1912
79	11	1	2011	November 2, 2011
80	11	1	2012	November 2, 2012
81	11	2	1812	November 3, 1812
82	11	2	1813	November 3, 1813
83	11	2	1912	November 3, 1912
84	11	2	2011	November 3, 2011
85	11	2	2012	November 3, 2012
86	11	15	1812	November 16, 1812
87	11	15	1813	November 16, 1813
88	11	15	1912	November 16, 1912
89	11	15	2011	November 16, 2011
90	11	15	2012	November 16, 2012
91	11	30	1812	December 1, 1812
92	11	30	1813	December 1, 1813
93	11	30	1912	December 1, 1912
94	11	30	2011	December 1, 2011
95	11	30	2012	December 1, 2012
96	11	31	1812	error
97	11	31	1813	error
98	11	31	1912	error
99	11	31	2011	error
100	11	31	2012	error
101	12	1	1812	December 2, 1812
102	12	1	1813	December 2, 1813
103	12	1	1912	December 2, 1912
104	12	1	2011	December 2, 2011
105	12	1	2012	December 2, 2012
106	12	2	1812	December 3, 1812
107	12	2	1813	December 3, 1813
108	12	2	1912	December 3, 1912
109	12	2	2011	December 3, 2011
110	12	2	2012	December 3, 2012

Table 5.3 Worst-Case Test Cases *(Continued)*

Case	Month	Day	Year	Expected Output
111	12	15	1812	December 16, 1812
112	12	15	1813	December 16, 1813
113	12	15	1912	December 16, 1912
114	12	15	2011	December 16, 2011
115	12	15	2012	December 16, 2012
116	12	30	1812	December 31, 1812
117	12	30	1813	December 31, 1813
118	12	30	1912	December 31, 1912
119	12	30	2011	December 31, 2011
120	12	30	2012	December 31, 2012
121	12	31	1812	January 1, 1813
122	12	31	1813	January 1, 1814
123	12	31	1912	January 1, 1913
124	12	31	2011	January 1, 2012
125	12	31	2012	January 1, 2013

5.5.3 Test Cases for the Commission Problem

Instead of going through 125 boring test cases again, we will look at some more interesting test cases for the commission problem. This time, we will look at boundary values for the output range, especially near the threshold points of $1000 and $1800. The output space of the commission is shown in Figure 5.6. The intercepts of these threshold planes with the axes are shown.

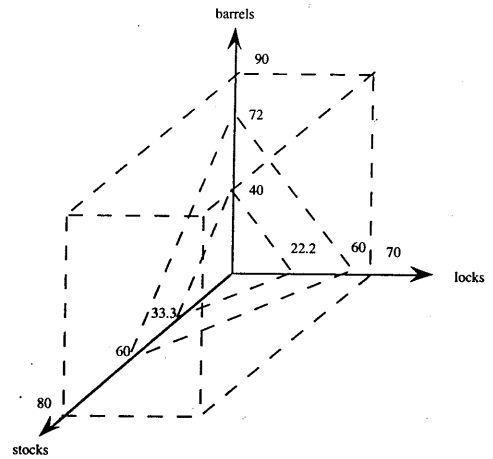

Figure 5.6 Input space of the commission problem.

Table 5.4 Output Boundary Value Analysis Test Cases

Case	Locks	Stocks	Barrels	Sales	Comm	Comment
1	1	1	1	100	10	output minimum
2	1	1	2	125	12.5	output minimum +
3	1	2	1	130	13	output minimum +
4	2	1	1	145	14.5	output minimum +
5	5	5	5	500	50	midpoint
6	10	10	9	975	97.5	border point –
7	10	9	10	970	97	border point –
8	9	10	10	955	95.5	border point –
9	10	10	10	1000	100	border point
10	10	10	11	1025	103.75	border point +
11	10	11	10	1030	104.5	border point +
12	11	10	10	1045	106.75	border point +
13	14	14	14	1400	160	midpoint
14	18	18	17	1775	216.25	border point –
15	18	17	18	1770	215.5	border point –
16	17	18	18	1755	213.25	border point –
17	18	18	18	1800	220	border point
18	18	18	19	1825	225	border point +
19	18	19	18	1830	226	border point +
20	19	18	18	1845	229	border point +
21	48	48	48	4800	820	midpoint
22	70	80	89	7775	1415	output maximum –
23	70	79	90	7770	1414	output maximum –
24	69	80	90	7755	1411	output maximum –
25	70	80	90	7800	1420	output maximum

The volume below the lower plane corresponds to sales below the $1000 threshold. The volume between the two planes is the 15% commission range. Part of the reason for using the output range to determine test cases is that cases from the input range are almost all in the 20% zone. We want to find input variable combinations that stress the boundary values: $100, $1000, $1800, and $7800. These test cases were developed with a spreadsheet, which saves a lot of calculator pecking. The minimum and maximum were easy, and the numbers happen to work out so that the border points are easy to generate. Here is where it gets interesting: test case 9 is the $1000 border point. If we tweak the input variables, we get values just below and just above the border (cases 6–8 and 10–12). If we wanted to, we could pick values near the intercepts (such as 22, 1, 1) and (21, 1, 1). As we continue in this way, we have a sense that we are "exercising" interesting parts of the code. We might claim that this is really a form of special value testing, because we used our mathematical insight to generate test cases.

Table 5.5 Output Special Value Test Cases

Case	Locks	Stocks	Barrels	Sales	Comm	Comment
1	10	11	9	1005	100.75	border point +
2	18	17	19	1795	219.25	border point –
3	18	19	17	1805	221	border point +

Table 5.6 Random Test Cases for the Triangle Program

Test Cases	Non-triangles	Scalene	Isosceles	Equilateral
1289	663	593	32	1
15436	7696	7372	367	1
17091	8556	8164	367	1
2603	1284	1252	66	1
6475	3197	3122	155	1
5978	2998	2850	129	1
9008	4447	4353	207	1
avg.	49.83%	47.87%	2.29%	0.01%

Table 5.7 Random Test Cases for the Commission Program

Test Cases	10%	15%	20%
91	1	6	84
27	1	1	25
72	1	1	70
176	1	6	169
48	1	1	46
152	1	6	145
125	1	4	120
avg.	1.01%	3.62%	95.37%

5.6 Random Testing

At least 2 decades of discussion of random testing are included in the literature. Most of this interest is among academics, and in a statistical sense, it is interesting. Our three sample problems lend themselves nicely to random testing. The basic idea is that, rather than always choose the min, min+, nom, max-, and max values of a bounded variable, use a random number generator to pick test case values. This avoids a form of bias in testing. It also raises a serious question: How many random test cases are sufficient? Later, when we discuss structural test coverage metrics, we will have an elegant answer. For now, Tables 5.6, 5.7, and 5.8 show the results of randomly generated test cases. They are derived from a Visual Basic application that picks values for a bounded variable $a \leq x \leq b$ as follows:

$$x = \text{Int}(b - a + 1) * \text{Rnd} + a$$

where the function Int returns the integer part of a floating point number, and the function Rnd generates random numbers in the interval [0, 1]. The program keeps generating random test cases until at least one of each output occurs. In

Table 5.8 Random Test Cases for the NextDate Program

Test Cases	*Days 1–30 of 31-day months*	*Days 31 of 31-day months*	*Days 1–29 of 30-day months*	*Days 30 of 30-day months*
913	542	17	274	10
1101	621	9	358	8
4201	2448	64	1242	46
1097	600	21	350	9
5853	3342	100	1804	82
3959	2195	73	1252	42
1436	786	22	456	13
avg	56.76%	1.65%	30.91%	1.13%
probab	56.45%	1.88%	31.18%	1.88%

Days 1–27 of Feb.	*Feb. 28 of a leap year*	*Feb. 28 of a non-leap year*	*Feb. 29 of a leap year*	*Impossible days*
45	1	1	1	22
83	1	1	1	19
312	1	8	3	77
92	1	4	1	19
417	1	11	2	94
310	1	6	5	75
126	1	5	1	26
7.46%	0.04%	0.19%	0.08%	1.79%
7.26%	0.07%	0.20%	0.07%	1.01%

each table, the program went through seven "cycles" that ended with the "hard-to-generate" test case.

5.7 Guidelines for Boundary Value Testing

With the exception of special value testing, the test methods based on the input domain of a function (program) are the most rudimentary of all functional testing methods. They share the common assumption that the input variables are truly independent; and when this assumption is not warranted, the methods generate unsatisfactory test cases (such as February 31, 1912 for NextDate). These methods have two other distinctions: normal versus robust values, and the single-fault versus the multiple-fault assumption. Just using these distinctions carefully will result in better testing. Each of these methods can be applied to the output range of a program, as we did for the commission problem.

Another useful form of output-based test cases is for systems that generate error messages. The tester should devise test cases to check that error messages are generated when they are appropriate, and are not falsely generated. Domain analysis can also be used for internal variables, such as loop control variables, indices, and pointers. Strictly speaking, these are not input variables; but errors

in the use of these variables are quite common. Robustness testing is a good choice for testing internal variables.

Exercises

1. Develop a formula for the number of robustness test cases for a function of n variables.
2. Develop a formula for the number of robust worst-case test cases for a function of n variables.
3. Make a Venn diagram showing the relationships among test cases from boundary value analysis, robustness testing, worst-case testing, and robust worst-case testing.
4. What happens if we try to do output range robustness testing? Use the commission problem as an example.
5. If you did exercise 8 in Chapter 2, you are already familiar with the CRC Press Web site for downloads (http://www.crcpress.com). There you will find an executable Visual Basic program named BlackBox.exe. (It is a file-oriented version of Naive.exe, and it contains the same inserted faults.) The files A1.txt, B1.txt, and C1.txt contain worst-case boundary value test cases for the Triangle, NextDate, and Commission problems, respectively. Run these sets of test cases, save the results in files that you name, and compare the results with your naive testing from Chapter 2.

Chapter 6

Equivalence Class Testing

The use of equivalence classes as the basis for functional testing has two motivations: we would like to have a sense of complete testing; and, at the same time, we would hope to avoid redundancy. Neither of these hopes is realized by boundary value testing: looking at the tables of test cases, it is easy to see massive redundancy — and looking more closely, serious gaps exist. Equivalence class testing echoes the two deciding factors of boundary value testing, robustness, and the single/multiple fault assumption. Three forms of equivalence class testing were identified in the first edition of this book; here, we identify four. The single versus multiple fault assumption yields the weak/strong distinction made in the first edition. The focus on invalid data yields a new distinction: robust versus normal.

Most of the standard testing texts (Myers, 1979; Mosley, 1993) discuss what we shall call weak robust equivalence class testing. This traditional form focuses on invalid data values, and it is/was a consequence of the dominant style of programming in the 1960s and 1970s. Input data validation was an important issue at the time, and "garbage in, garbage out" was the programmer's watchword. The usual response to this problem was extensive input validation sections of a program. Authors and seminar leaders frequently commented that, in the classic afferent/central/efferent architecture of structured programming, the afferent portion often represented 80% of the total source code. In this context, it is natural to emphasize input data validation. The gradual shift to modern programming languages, especially those that feature strong data typing, and then to graphical user interfaces (GUIs) obviated much of the need for input data validation.

6.1 Equivalence Classes

In Chapter 3, we noted that the important aspect of equivalence classes is that they form a partition of a set, where partition refers to a collection of mutually disjoint subsets when the union is the entire set. This has two important implications for testing: the fact that the entire set is represented provides a form of

completeness, and the disjointedness ensures a form of nonredundancy. Because the subsets are determined by an equivalence relation, the elements of a subset have something in common. The idea of equivalence class testing is to identify test cases by using one element from each equivalence class. If the equivalence classes are chosen wisely, this greatly reduces the potential redundancy among test cases. In the Triangle Problem, for example, we would certainly have a test case for an equilateral triangle, and we might pick the triple (5, 5, 5) as inputs for a test case. If we did this, we would not expect to learn much from test cases such as (6, 6, 6) and (100, 100, 100). Our intuition tells us that these would be "treated the same" as the first test case, thus, they would be redundant. When we consider structural testing in Part III, we shall see that "treated the same" maps onto "traversing the same execution path."

The key (and the craft!) of equivalence class testing is the choice of the equivalence relation that determines the classes. Very often, we make this choice by second-guessing the likely implementation and thinking about the functional manipulations that must somehow be present in the implementation. We will illustrate this with our continuing examples; but first, we need to make a distinction between weak and strong equivalence class testing. After that, we will compare these to the traditional form of equivalence class testing.

We need to enrich the function we used in boundary value testing. Again, for the sake of comprehensible drawings, the discussion relates to a function, F, of two variables x_1 and x_2. When F is implemented as a program, the input variables x_1 and x_2 will have the following boundaries, and intervals within the boundaries:

$a \le x_1 \le d$, with intervals [a, b), [b, c), [c, d]
$e \le x_2 \le g$, with intervals [e, f), [f, g]

where square brackets and parentheses denote, respectively, closed and open interval endpoints. Invalid values of x_1 and x_2 are: $x_1 < a$, $x_1 > d$, and $x_2 < e$, $x_2 > g$.

6.1.1 Weak Normal Equivalence Class Testing

With the notation as given previously, weak normal equivalence class testing is accomplished by using one variable from each equivalence class (interval) in a test case. (Note the effect of the single fault assumption.) For the previous example, we would end up with the weak equivalence class test cases shown in Figure 6.1

These three test cases uses one value from each equivalence class. We identify these in a systematic way, thus the apparent pattern. In fact, we will always have the same number of weak equivalence class test cases as classes in the partition with the largest number of subsets.

6.1.2 Strong Normal Equivalence Class Testing

Strong equivalence class testing is based on the multiple fault assumption, so we need test cases from each element of the Cartesian product of the equivalence classes, as shown in Figure 6.2.

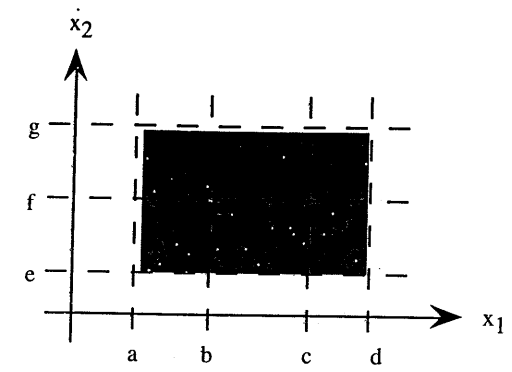

Figure 6.1 Weak normal equivalence class test cases.

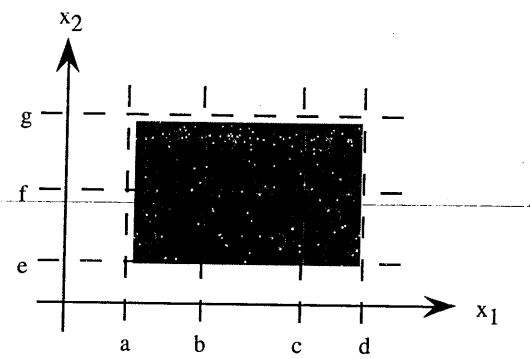

Figure 6.2 Strong normal equivalence class test cases.

Notice the similarity between the pattern of these test cases and the construction of a truth table in propositional logic. The Cartesian product guarantees that we have a notion of "completeness" in two senses: we cover all the equivalence classes, and we have one of each possible combination of inputs.

As we shall see from our continuing examples, the key to "good" equivalence class testing is the selection of the equivalence relation. Watch for the notion of inputs being "treated the same." Most of the time, equivalence class testing defines classes of the input domain. There is no reason why we could not define equivalence relations on the output range of the program function being tested; in fact, this is the simplest approach for the Triangle Problem.

6.1.3 Weak Robust Equivalence Class Testing

The name for this form is admitedly counterintuitive and oxymoronic. How can something be both weak and robust? The robust part comes from consideration

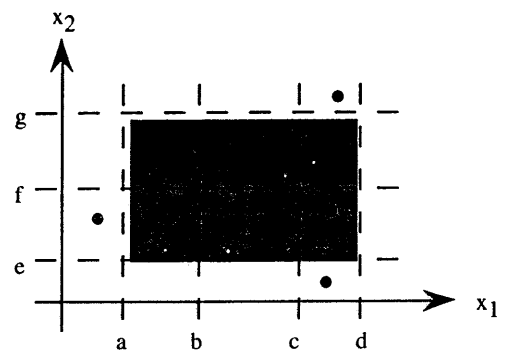

Figure 6.3 Weak robust equivalence class test cases.

of invalid values, and the weak part refers to the single fault assumption. (This form was referred to as "traditional equivalence class testing" in the first edition of this book.)

1. For valid inputs, use one value from each valid class (as in what we have called weak normal equivalence class testing. Note that each input in these test cases will be valid.)
2. For invalid inputs, a test case will have one invalid value and the remaining values will all be valid. (Thus, a "single failure" should cause the test case to fail.)

The test cases resulting from this strategy are shown in Figure 6.3.

Two problems occur with robust equivalence testing. The first is that, very often, the specification does not define what the expected output for an invalid test case should be. (We could argue that this is a deficiency of the specification, but that does not get us anywhere.) Thus, testers spend a lot of time defining expected outputs for these cases. The second problem is that strongly typed languages eliminate the need for the consideration of invalid inputs. Traditional equivalence testing is a product of the time when languages such as FORTRAN and COBOL were dominant, thus, this type of error was common. In fact, it was the high incidence of such errors that led to the implementation of strongly typed languages.

6.1.4 Strong Robust Equivalence Class Testing

At least the name for this form is neither counterintuitive nor oxymoronic, just redundant.. As before, the robust part comes from consideration of invalid values, and the strong part refers to the multiple fault assumption. (This form was omitted in the first edition of this book.)

We obtain test cases from each element of the Cartesian product of all the equivalence classes, as shown in Figure 6.4.

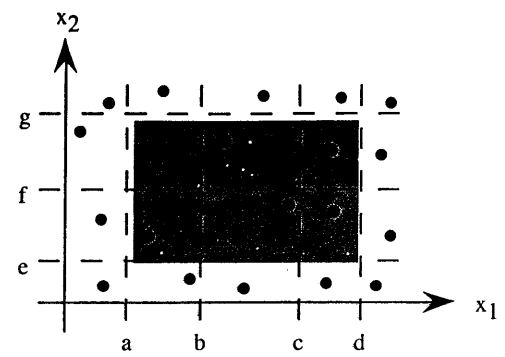

Figure 6.4 Strong robust equivalence class test cases.

6.2 Equivalence Class Test Cases for the Triangle Problem

In the problem statement, we note that four possible outputs can occur: NotA-Triangle, Scalene, Isosceles, and Equilateral. We can use these to identify output (range) equivalence classes as follows.

R1 = {<a, b, c> : the triangle with sides a, b, and c is equilateral}
R2 = {<a, b, c> : the triangle with sides a, b, and c is isosceles}
R3 = {<a, b, c> : the triangle with sides a, b, and c is scalene}
R4 = {<a, b, c> : sides a, b, and c do not form a triangle}

The four weak normal equivalence class test cases are:

Test Case	*a*	*b*	*c*	*Expected Output*
WN1	5	5	5	Equilateral
WN2	2	2	3	Isosceles
WN3	3	4	5	Scalene
WN4	4	1	2	Not a Triangle

Because no valid subintervals of variables a, b, and c exist, the strong normal equivalence class test cases are identical to the weak normal equivalence class test cases.

Considering the invalid values for a, b, and c yields the following additional weak robust equivalence class test cases:

Test Case	*a*	*b*	*c*	*Expected Output*
WR1	–1	5	5	Value of a is not in the range of permitted values
WR2	5	–1	5	Value of b is not in the range of permitted values
WR3	5	5	–1	Value of c is not in the range of permitted values

Test Case	a	b	c	Expected Output
WR4	201	5	5	Value of a is not in the range of permitted values
WR5	5	201	5	Value of b is not in the range of permitted values
WR6	5	5	201	Value of c is not in the range of permitted values

Here is one "corner" of the cube in 3-space of the additional strong robust equivalence class test cases:

Test Case	a	b	c	Expected Output
SR1	–1	5	5	Value of a is not in the range of permitted values
SR2	5	–1	5	Value of b is not in the range of permitted values
SR3	5	5	–1	Value of c is not in the range of permitted values
SR4	–1	–1	5	Values of a, b are not in the range of permitted values
SR5	5	–1	–1	Values of b, c are not in the range of permitted values
SR6	–1	5	–1	Values of a, c are not in the range of permitted values
SR7	–1	–1	–1	Values of a, b, c are not in the range of permitted values

Notice how thoroughly the expected outputs describe the invalid input values.

Equivalence class testing is clearly sensitive to the equivalence relation used to define classes. Here is another instance of craftsmanship. If we base equivalence classes on the input domain, we obtain a richer set of test cases. What are some of the possibilities for the three integers, a, b, and c? They can all be equal, exactly one pair can be equal (this can happen in three ways), or none can be equal.

D1 = {<a, b, c> : a = b = c}
D2 = {<a, b, c> : a = b, a ≠ c}
D3 = {<a, b, c> : a = c, a ≠ b}
D4 = {<a, b, c> : b = c, a ≠ b}
D5 = {<a, b, c> : a ≠ b, a ≠ c, b ≠ c}

As a separate question, we can apply the triangle property to see if they even constitute a triangle. (For example, the triplet <1, 4, 1> has exactly one pair of equal sides, but these sides do not form a triangle.)

D6 = {<a, b, c> : a ≥ b + c}
D7 = {<a, b, c> : b ≥ a + c}
D8 = {<a, b, c> : c ≥ a + b}

If we wanted to be still more thorough, we could separate the "less than or equal to" into the two distinct cases; thus, the set D6 would become:

D6' = {<a, b, c> : a = b + c}
D6" = {<a, b, c> : a > b + c}

and similarly for D7 and D8.

6.3 Equivalence Class Test Cases for the NextDate Function

The NextDate function illustrates very well the craft of choosing the underlying equivalence relation. Recall that NextDate is a function of three variables — month, day, and year — and these have intervals of valid values defined as follows:

M1 = {month : 1 ≤ month ≤ 12}
D1 = {day : 1 ≤ day ≤ 31}
Y1 = {year : 1812 ≤ year ≤ 2012}

The invalid equivalence classes are:

M2 = {month : month < 1}
M3 = {month : month > 12}
D2 = {day : day < 1}
D3 = {day : day > 31}
Y2 = {year : year < 1812}
Y3 = {year : year > 2012}

Because the number of valid classes equals the number of independent variables, only one weak normal equivalence class test case occurs, and it is identical to the strong normal equivalence class test case:

Case ID	Month	Day	Year	Expected Output
WN1, SN1	6	15	1912	6/16/1912

Here is the full set of weak robust test cases:

Case ID	Month	Day	Year	Expected Output
WR1	6	15	1912	6/16/1912
WR2	–1	15	1912	Value of month not in the range 1..12
WR3	13	15	1912	Value of month not in the range 1..12
WR4	6	–1	1912	Value of day not in the range 1..31
WR5	6	32	1912	Value of day not in the range 1..31
WR6	6	15	1811	Value of year not in the range 1812..2012
WR7	6	15	2013	Value of year not in the range 1812..2012

As with the Triangle Problem, here is one "corner" of the cube in 3-space of the additional strong robust equivalence class test cases:

Case ID	Month	Day	Year	Expected Output
SR1	–1	15	1912	Value of month not in the range 1..12

Case ID	Month	Day	Year	Expected Output
SR2	6	–1	1912	Value of day not in the range 1..31
SR3	6	15	1811	Value of year not in the range 1812..2012
SR4	–1	–1	1912	Value of month not in the range 1..12 Value of day not in the range 1..31
SR5	6	–1	1811	Value of day not in the range 1..31 Value of year not in the range 1812..2012
SR6	–1	15	1811	Value of month not in the range 1..12 Value of year not in the range 1812..2012
SR7	–1	–1	1811	Value of month not in the range 1..12 Value of day not in the range 1..31 Value of year not in the range 1812..2012

If we more carefully choose the equivalence relation, the resulting equivalence classes will be more useful. Recall that earlier we said that the gist of the equivalence relation is that elements in a class are "treated the same way." One way to see the deficiency of the traditional approach is that the "treatment" is at the valid/invalid level. We next reduce the granularity by focusing on more specific treatment.

What must be done to an input date? If it is not the last day of a month, the NextDate function will simply increment the day value. At the end of a month, the next day is 1 and the month is incremented. At the end of a year, both the day and the month are reset to 1, and the year is incremented. Finally, the problem of leap year makes determining the last day of a month interesting. With all this in mind, we might postulate the following equivalence classes:

M1 = {month : month has 30 days}
M2 = {month : month has 31 days}
M3 = {month : month is February}
D1 = {day : $1 \leq$ day ≤ 28}
D2 = {day : day = 29}
D3 = {day : day = 30}
D4 = {day : day = 31}
Y1 = {year : year = 2000}
Y2 = {year : year is a leap year}
Y3 = {year : year is a common year}

By choosing separate classes for 30- and 31-day months, we simplify the question of the last day of the month. By taking February as a separate class, we can give more attention to leap year questions. We also give special attention to day values: days in D1 are (nearly) always incremented, while days in D4 only have meaning for months in M2. Finally, we have three classes of years, the special case of the year 2000, leap years, and non-leap years. This is not a perfect set of equivalence classes, but its use will reveal many potential errors.

6.3.1 Equivalence Class Test Cases

These classes yield the following weak equivalence class test cases. As before, the inputs are mechanically selected from the approximate middle of the corresponding class:

Case ID	Month	Day	Year	Expected Output
WN1	6	14	2000	6/15/2000
WN2	7	29	1996	7/30/1996
WN3	2	30	2002	2/31/2002 (impossible date)
WN4	6	31	2000	7/1/2000 (impossible input date)

Mechanical selection of input values makes no consideration of our domain knowledge, thus the two impossible dates. This will always be a problem with "automatic" test case generation, because all of our domain knowledge is not captured in the choice of equivalence classes. The strong normal equivalence class test cases for the revised classes are:

Case ID	Month	Day	Year	Expected Output
SN1	6	14	2000	6/15/2000
SN2	6	14	1996	6/15/1996
SN3	6	14	2002	6/15/2002
SN4	6	29	2000	6/30/2000
SN5	6	29	1996	6/30/1996
SN6	6	29	2002	6/30/2002
SN7	6	30	2000	6/31/2000 (impossible date)
SN8	6	30	1996	6/31/1996 (impossible date)
SN9	6	30	2002	6/31/2002 (impossible date)
SN10	6	31	2000	7/1/2000 (invalid input)
SN11	6	31	1996	7/1/1996 (invalid input)
SN12	6	31	2002	7/1/2002 (invalid input)
SN13	7	14	2000	7/15/2000
SN14	7	14	1996	7/15/1996
SN15	7	14	2002	7/15/2002
SN16	7	29	2000	7/30/2000
SN17	7	29	1996	7/30/1996
SN18	7	29	2002	7/30/2002
SN19	7	30	2000	7/31/2000
SN20	7	30	1996	7/31/1996
SN21	7	30	2002	7/31/2002
SN22	7	31	2000	8/1/2000
SN23	7	31	1996	8/1/1996
SN24	7	31	2002	8/1/2002
SN25	2	14	2000	2/15/2000

Case ID	Month	Day	Year	Expected Output
SN26	2	14	1996	2/15/1996
SN27	2	14	2002	2/15/2002
SN28	2	29	2000	3/1/2000 (invalid input)
SN29	2	29	1996	3/1/1996
SN30	2	29	2002	3/1/2002 (impossible date)
SN31	2	30	2000	3/1/2000 (impossible date)
SN32	2	30	1996	3/1/1996 (impossible date)
SN33	2	30	2002	3/1/2002 (impossible date)
SN34	2	31	2000	7/1/2000 (impossible date)
SN35	2	31	1996	7/1/1996 (impossible date)
SN36	2	31	2002	7/1/2002 (impossible date)

Moving from weak to strong normal testing raises some of the issues of redundancy that we saw with boundary value testing. The move from weak to strong, whether with normal or robust classes, always makes the presumption of independence; and this is reflected in the cross-product of the equivalence classes. Three month classes times four day classes times three year classes results in 36 strong normal equivalence class test cases. Adding two invalid classes for each variable will result in 150 strong robust equivalence class test cases (too many to show here!).

We could also streamline our set of test cases by taking a closer look at the year classes. If we merge Y1 and Y3, and call the result the set of common years, our 36 test cases would drop down to 24. This change suppresses special attention to considerations in the year 2000, and it also adds some complexity to the determination of which years are leap years. Balance this against how much might be learned form the present test cases.

6.4 Equivalence Class Test Cases for the Commission Problem

The input domain of the commission problem is "naturally" partitioned by the limits on locks, stocks, and barrels. These equivalence classes are exactly those that would also be identified by traditional equivalence class testing. The first class is the valid input, the other two are invalid. The input domain equivalence classes lead to very unsatisfactory sets of test cases. Equivalence classes defined on the output range of the commission function will be an improvement.

The valid classes of the input variables are:

L1 = {locks : $1 \leq$ locks ≤ 70}
L2 = {locks = –1}
S1 = {stocks : $1 \leq$ stocks ≤ 80}
B1 = {barrels : $1 \leq$ barrels ≤ 90}

The corresponding invalid classes of the input variables are:

L_3 = {locks : locks = 0 OR locks < –1}
L_4 = {locks : locks > 70}
S2 = {stocks : stocks < 1}
S3 = {stocks : stocks > 80}
B2 = {barrels : barrels < 1}
B3 = {barrels : barrels > 90}

One problem occurs, however. The variable Locks is also used as a sentinel to indicate no more telegrams. When a value of –1 is given for Locks, the While loop terminates, and the values of totalLocks, totalStocks, and totalBarrels are used to compute sales, and then commission.

Except for the names of the variables and the interval endpoint values, this is identical to our first version of the NextDate function. Therefore, we will have exactly one weak normal equivalence class test case — and again, it is identical to the strong normal equivalence class test case. Also, we will have seven weak robust test cases. Finally, a "corner" of the cube will be in 3-space of the additional strong robust equivalence class test cases:

Case ID	Locks	Stocks	Barrels	Expected Output
SR1	–1	40	45	Value of Locks not in the range 1..70
SR2	35	–1	45	Value of Stocks not in the range 1..80
SR3	35	40	–1	Value of Barrels not in the range 1..90
SR4	–1	–1	45	Value of Locks not in the range 1..70 Value of Stocks not in the range 1..80
SR5	–1	40	–1	Value of Locks not in the range 1..70 Value of Barrels not in the range 1..90
SR6	35	–1	–1	Value of Stocks not in the range 1..80 Value of Barrels not in the range 1..90
SR7	–1	–1	–1	Value of Locks not in the range 1..70 Value of Stocks not in the range 1..80 Value of Barrels not in the range 1..90

6.4.1 Output Range Equivalence Class Test Cases

Notice that, of strong test cases — whether normal or robust — only one is a legitimate input. If we were really worried about error cases, this might be a good set of test cases. It can hardly give us a sense of confidence about the calculation portion of the problem, however. We can get some help by considering equivalence classes defined on the output range. Recall that sales is a function of the number of locks, stocks, and barrels sold:

$$\text{sales} = 45 \times \text{locks} + 30 \times \text{stocks} + 25 \times \text{barrels}$$

We could define equivalence classes of three variables by commission ranges:

S1 = {<locks, stocks, barrels> : sales ≤ 1000}

S2 = {<locks, stocks, barrels> : 1000 < sales ≤ 1800}
S3 = {<locks, stocks, barrels> : sales > 1800}

Figure 5.6 helps us get a better feel for the input space. Elements of S1 are points with integer coordinates in the pyramid near the origin. Elements of S2 are points in the "triangular slice" between the pyramid and the rest of the input space. Finally, elements of S3 are all those points in the rectangular volume that are not in S1 or in S2. All the error cases found by the strong equivalence classes of the input domain are outside of the rectangular space shown in Figure 5.6.

As was the case with the Triangle Problem, the fact that our input is a triplet means that we no longer take test cases form a Cartesian product.

6.4.2 Output Range Equivalence Class Test Cases

Test Case	*Locks*	*Stocks*	*Barrels*	*Sales*	*Commission*
OR1	5	5	5	500	50
OR2	15	15	15	1500	175
OR3	25	25	25	2500	360

These test cases give us some sense that we are exercising important parts of the problem. Together with the weak robust test cases, we would have a pretty good test of the commission problem. We might want to add some boundary checking, just to make sure the transitions at sales of $1000 and $1800 are correct. This is not particularly easy because we can only choose values of Locks, Stocks, and Barrels. It happens that the constants in this example are contrived so that there are "nice" triplets.

6.5 Guidelines and Observations

Now that we have gone through three examples, we conclude with some observations about, and guidelines for, equivalence class testing.

1. Obviously, the weak forms of equivalence class testing (normal or robust) are not as comprehensive as the corresponding strong forms.
2. If the implementation language is strongly typed (and invalid values cause run-time errors), it makes no sense to use the robust forms.
3. If error conditions are a high priority, the robust forms are appropriate.
4. Equivalence class testing is appropriate when input data is defined in terms of intervals and sets of discrete values. This is certainly the case when system malfunctions can occur for out-of-limit variable values.
5. Equivalence class testing is strengthened by a hybrid approach with boundary value testing. (We can "reuse" the effort made in defining the equivalence classes.)

6. Equivalence class testing is indicated when the program function is complex. In such cases, the complexity of the function can help identify useful equivalence classes, as in the NextDate function.
7. Strong equivalence class testing makes a presumption that the variables are independent, and the corresponding multiplication of test cases raises issues of redundancy. If any dependencies occur, they will often generate "error" test cases, as they did in the NextDate function. (The decision table technique in Chapter 7 resolves this problem.)
8. Several tries may be needed before the "right" equivalence relation is discovered, as we saw in the NextDate example. In other cases, there is an "obvious" or "natural" equivalence relation. When in doubt, the best bet is to try to second-guess aspects of any reasonable implementation.
9. The difference between the strong and weak forms of equivalence class testing is helpful in the distinction between progression and regression testing.

References

Mosley, Daniel J., *The Handbook of MIS Application Software Testing,* Yourdon Press, Prentice Hall, Englewood Cliffs, NJ, 1993.
Myers, Glenford J., *The Art of Software Testing,* Wiley Interscience, New York, 1979.

Exercises

1. Starting with the 36 strong normal equivalence class test cases for the NextDate function, revise the day classes as discussed. and then find the other nine test cases.
2. If you use a compiler for a strongly typed language, discuss how it would react to robust equivalence class test cases.
3. Revise the set of weak normal equivalence classes for the extended triangle problem that considers right triangles.
4. Compare and contrast the single/multiple fault assumption with boundary value and equivalence class testing.
5. The spring and fall changes between standard and daylight savings time create an interesting problem for telephone bills. In the spring, this switch occurs at 2:00 a.m. on a Sunday morning (late March, early April) when clocks are reset to 3:00 a.m. The symmetric change takes place usually on the last Sunday in October, when the clock changes from 2:59:59 back to 2:00:00.

   Develop equivalence classes for a long-distance telephone service function that bills calls using the following rate structure:

   Call Duration < = 20 minutes charged at $0.05 per each minute or fraction of a minute
   Call Duration > 20 minutes charged at $1.00 plus $0.10 per each minute or fraction of a minute in excess of 20 minutes.

Make these assumptions:

- Chargeable time of a call begins when the called party answers, and ends when the calling party disconnects.
- Call durations of seconds are rounded up to the next larger minute.
- No call lasts more than 30 hours.

6. If you did exercise 8 in Chapter 2, and exercise 5 in Chapter 5, you are already familiar with the CRC Press Web site for downloads (http://www.crcpress.com). There you will find an executable Visual Basic program named BlackBox.exe. (It is a file-oriented version of Naive.exe, and it contains the same inserted faults.) The files A2.txt, B2.txt, and C2.txt contain equivalence class test cases for the Triangle, NextDate, and Commission problems, respectively. Run these sets of test cases, save the results in files that you name, and compare the results with your naive testing from Chapter 2 and your boundary value testing from Chapter 5.

Chapter 7

Decision Table-Based Testing

Of all the functional testing methods, those based on decision tables are the most rigorous because decision tables enforce logical rigor. Two closely related methods are used: cause-effect graphing (Elmendorf, 1973; Myers, 1979) and the decision tableau method (Mosley, 1993). These are more cumbersome to use and are fully redundant with decision tables, so we will not discuss them here. Both are covered in Mosley (1993).

7.1 Decision Tables

Decision tables have been used to represent and analyze complex logical relationships since the early 1960s. They are ideal for describing situations in which a number of combinations of actions are taken under varying sets of conditions. Some of the basic decision table terms are illustrated in Table 7.1

A decision table has four portions: the part to the left of the bold vertical line is the stub portion; to the right is the entry portion. The part above the bold line is the condition portion, and below is the action portion. Thus, we can refer to

Table 7.1 Portions of a Decision Table

Stub	*Rule 1*	*Rule 2*	*Rules 3, 4*	*Rule 5*	*Rule 6*	*Rules 7, 8*
c1	T	T	T	F	F	F
c2	T	T	F	T	T	F
c3	T	F	–	T	F	–
a1	X	X		X		
a2	X				X	
a3		X		X		
a4			X			X

the condition stub, the condition entries, the action stub, and the action entries. A column in the entry portion is a rule. Rules indicate which actions are taken for the conditional circumstances indicated in the condition portion of the rule. In the decision table in 7.1, when conditions c1, c2, and c3 are all true, actions a1 and a2 occur. When c1 and c2 are both true and c3 is false, then actions a1 and a3 occur. The entry for c3 in the rule where c1 is true and c2 is false is called a "don't care" entry. The don't care entry has two major interpretations: the condition is irrelevant, or the condition does not apply. Sometimes people will enter the "n/a" symbol for this latter interpretation.

When we have binary conditions (true/false, yes/no, 0/1), the condition portion of a decision table is a truth table (from propositional logic) that has been rotated 90°. This structure guarantees that we consider every possible combination of condition values. When we use decision tables for test case identification, this completeness property of a decision table guarantees a form of complete testing. Decision tables in which all the conditions are binary are called limited entry decision tables. If conditions are allowed to have several values, the resulting tables are called extended entry decision tables. We will see examples of both types for the NextDate problem.

Decision tables are deliberately declarative (as opposed to imperative); no particular order is implied by the conditions, and selected actions do not occur in any particular order.

7.1.2 Technique

To identify test cases with decision tables, we interpret conditions as inputs and actions as outputs. Sometimes conditions end up referring to equivalence classes of inputs, and actions refer to major functional processing portions of the item tested. The rules are then interpreted as test cases. Because the decision table can mechanically be forced to be complete, we know we have a comprehensive set of test cases.

Several techniques that produce decision tables are more useful to testers. One helpful style is to add an action to show when a rule is logically impossible.

In the decision table in 7.2, we see examples of don't care entries and impossible rule usage. If the integers a, b, and c do not constitute a triangle, we do not even care about possible equalities, as indicated in the first rule. In rules

Table 7.2 Decision Table for the Triangle Problem

c3: a, b, c form a triangle?	N	Y	Y	Y	Y	Y	Y	Y	Y
c2: a = b?	–	Y	Y	Y	Y	N	N	N	N
c3: a = c?	–	Y	Y	N	N	Y	Y	N	N
c4: b = c?	–	Y	N	Y	N	Y	N	Y	N
a1: Not a triangle	X								
a2: Scalene									X
a3: Isosceles					X		X	X	
a4: Equilateral		X							
a5: Impossible			X	X		X			

Table 7.3 Refined Decision Table for the Triangle Problem

c1: a<b+c?	F	T	T	T	T	T	T	T	T	T
c2: b<a+c?	–	F	T	T	T	T	T	T	T	T
c3: c<a+b?	–	–	F	T	T	T	T	T	T	T
c4: a = b?	–	–	–	T	T	T	F	F	F	F
c5: a = c?	–	–	–	T	T	F	F	T	F	F
c6: b = c?	–	–	–	T	F	T	F	T	F	F
a1: Not a triangle	X	X	X							
a2: Scalene										X
a3: Isosceles							X		X	X
a4: Equilateral				X						
a5: Impossible					X	X		X		

3, 4, and 6, if two pairs of integers are equal, by transitivity, the third pair must be equal; thus, the negative entry makes these rules impossible.

The decision table in 7.3 illustrates another consideration related to technique: the choice of conditions can greatly expand the size of a decision table. Here, we expanded the old condition (c1: a, b, c form a triangle?) to a more detailed view of the three inequalities of the triangle property. If any one of these fails, the three integers do not constitute sides of a triangle. We could expand this still further because in two ways an inequality could fail: one side could equal the sum of the other two, or it could be strictly greater.

When conditions refer to equivalence classes, decision tables have a characteristic appearance. Conditions in the decision table in 7.4 are from the NextDate Problem; they refer to the mutually exclusive possibilities for the month variable. Because a month is in exactly one equivalence class, we cannot ever have a rule in which two entries are true. The don't care entries (—) really mean "must be false." Some decision table aficionados use the notation F! to make this point.

Use of don't care entries has a subtle effect on the way in which complete decision tables are recognized. For limited entry decision tables, if n conditions exist, there must be 2^n rules. When don't care entries really indicate that the condition is irrelevant, we can develop a rule count as follows. Rules in which no don't care entries occur count as one rule. Each don't care entry in a rule doubles the count of that rule. The rule counts for the decision table in Table 7.3 are shown below in 7.5. Notice that the sum of the rule counts is 64 (as it should be).

Table 7.4 Decision Table with Mutually Exclusive Conditions

Conditions	*R1*	*R2*	*R3*
c1: month in M1?	T	–	–
c2: month in M2?	–	T	–
c3: month in M3?	–	–	T
a1			
a2			
a3			

Table 7.5 Decision Table for Table 7.3 with Rule Counts

c1: a<b+c?	F	T	T	T	T	T	T	T	T	T	T
c2: b<a+c?	—	F	T	T	T	T	T	T	T	T	T
c3: c<a+b?	—	—	F	T	T	T	T	T	T	T	T
c4: a = b?	—	—	—	T	T	T	T	F	F	F	F
c5: a = c?	—	—	—	T	T	F	F	T	T	F	F
c6: b = c?	—	—	—	T	F	T	F	T	F	T	F
Rule Count	32	16	8	1	1	1	1	1	1	1	1
a1: Not a triangle	X	X	X								
a2: Scalene											X
a3: Isosceles							X		X	X	
a4: Equilateral				X							
a5: Impossible					X	X		X			

Table 7.6 Rule Counts for a Decision Table with Mutually Exclusive Conditions

Conditions	*R1*	*R2*	*R3*
c1: month in M1	T	—	—
c2: month in M2	—	T	—
c3: month in M3	—	—	T
Rule Count	4	4	4
a1			

If we applied this simplistic algorithm to the decision table in 7.4, we get the rule counts shown in 7.6.

We should only have eight rules, so we clearly have a problem. To see where the problem lies, we expand each of the three rules, replacing the "—" entries with the T and F possibilities, as shown in 7.7.

Notice that we have three rules in which all entries are T: rules 1.1, 2.1, and 3.1. We also have two rules with T, T, F entries: rules 1.2 and 2.2. Similarly, rules 1.3 and 3.2 are identical; so are rules 2.3 and 3.3. If we delete the repetitions, we end up with seven rules; the missing rule is the one in which all conditions are false. The result of this process in shown in 7.8. The impossible rules are also shown.

The ability to recognize (and develop) complete decision tables puts us in a powerful position with respect to redundancy and inconsistency. The decision

Table 7.7 Expanded Version of Table 7.6

Conditions	*1.1*	*1.2*	*1.3*	*1.4*	*2.1*	*2.2*	*2.3*	*2.4*	*3.1*	*3.2*	*3.3*	*3.4*
c1: mo. in M1	T	T	T	T	T	T	F	F	T	T	F	F
c2: mo. in M2	T	T	F	F	T	T	T	T	T	F	T	F
c3: mo. in M3	T	F	T	F	T	F	T	F	T	T	T	T
Rule Count	1	1	1	1	1	1	1	1	1	1	1	1
a1												

Table 7.8 Mutually Exclusive Conditions with Impossible Rules

	1.1	*1.2*	*1.3*	*1.4*	*2.3*	*2.4*	*3.4*	
c1: mo. in M1	T	T	T	T	F	F	F	F
c2: mo. in M2	T	T	F	F	T	T	F	F
c3: mo. in M3	T	F	T	F	T	F	T	F
Rule Count	1	1	1	1	1	1	1	1
a1: Impossible	X	X	X		X			X

Table 7.9 A Redundant Decision Table

Conditions	*1–4*	*5*	*6*	*7*	*8*	*9*
c1	T	F	F	F	F	T
c2	—	T	T	F	F	F
c3	—	T	F	T	F	F
a1	X	X	X	—	—	X
a2	—	X	X	X	—	—
a3	X	—	X	X	X	X

table in 7.9 is redundant — three conditions and nine rules exist. (Rule 9 is identical to rule 4.)

Notice that the action entries in rule 9 are identical to those in rules 1 – 4. As long as the actions in a redundant rule are identical to the corresponding part of the decision table, we do not have much of a problem. If the action entries are different, as they are in 7.10, we have a bigger problem.

If the decision table in 7.10 were to process a transaction in which c1 is true and both c2 and c3 are false, both rules 4 and 9 apply. We can make two observations:

1. Rules 4 and 9 are inconsistent.
2. The decision table is nondeterministic.

Rules 4 and 9 are inconsistent because the action sets are different. The whole table is nondeterministic because there is no way to decide whether to apply rule 4 or rule 9. The bottom line for testers is that care should be taken when don't care entries are used in a decision table.

Table 7.10 An Inconsistent Decision Table

Conditions	*1–4*	*5*	*6*	*7*	*8*	*9*
c1	T	F	F	F	F	T
c2	—	T	T	F	F	F
c3	—	T	F	T	F	F
a1	X	X	X	—	—	—
a2	—	X	X	X	—	X
a3	X	—	X	X	X	—

Table 7.11 Test Cases from Table 7.3

Case ID	a	b	c	Expected Output
DT1	4	1	2	Not a Triangle
DT2	1	4	2	Not a Triangle
DT3	1	2	4	Not a Triangle
DT4	5	5	5	Equilateral
DT5	?	?	?	Impossible
DT6	?	?	?	Impossible
DT7	2	2	3	Isosceles
DT8	?	?	?	Impossible
DT9	2	3	2	Isosceles
DT10	3	2	2	Isosceles
DT11	3	4	5	Scalene

7.2 Test Cases for the Triangle Problem

Using the decision table in 7.3, we obtain 11 functional test cases: three impossible cases, three ways to fail the triangle property, one way to get an equilateral triangle, one way to get a scalene triangle, and three ways to get an isosceles triangle (see Table 7.11). If we extended the decision table to show both ways to fail an inequality, we would pick up three more test cases (where one side is exactly the sum of the other two). Some judgment is required in this because of the exponential growth of rules. In this case, we would end up with many more don't care entries and more impossible rules.

7.3 Test Cases for the NextDate Function

The NextDate function was chosen because it illustrates the problem of dependencies in the input domain. This makes it a perfect example for decision table-based testing, because decision tables can highlight such dependencies. Recall that, in Chapter 6, we identified equivalence classes in the input domain of the NextDate function. One of the limitations we found in Chapter 6 was that indiscriminate selection of input values from the equivalence classes resulted in "strange" test cases, such as finding the next date to June 31, 1812. The problem stems from the presumption that the variables are independent. If they are, a Cartesian product of the classes makes sense. When logical dependencies exist among variables in the input domain, these dependencies are lost (suppressed is better) in a Cartesian product. The decision table format lets us emphasize such dependencies using the notion of the "impossible action" to denote impossible combinations of conditions. In this section, we will make three tries at a decision table formulation of the NextDate function.

7.3.1 First Try

Identifying appropriate conditions and actions presents an opportunity for craftsmanship. Suppose we start with a set of equivalence classes close to the one we used in Chapter 6.

Table 7.12 First Try Decision Table with 256 Rules

Conditions											
c1: month in M1?	T										
c2: month in M2?		T									
c3: month in M3?			T								
c4: day in D1?											
c5: day in D2?											
c6: day in D3?											
c7: day in D4?											
c8: year in Y1?											
a1: impossible											
a2: next date											

M1 = {month : month has 30 days}
M2 = {month : month has 31 days}
M3 = {month : month is February}
D1 = {day : 1 ≤ day ≤ 28}
D2 = {day : day = 29}
D3 = {day : day = 30}
D4 = {day : day = 31}
Y1 = {year : year is a leap year}
Y2 = {year : year is not a leap year}

If we wish to highlight impossible combinations, we could make a limited entry decision table with the following conditions and actions. (Note that the equivalence classes for the year variable collapse into one condition in Table 7.12.)

This decision table will have 256 rules, many of which will be impossible. If we wanted to show why these rules were impossible, we might revise our actions to the following:

a1: Too many days in a month
a2: Cannot happen in a non-leap year
a3: Compute the next date

7.3.2 Second Try

If we focus on the leap year aspect of the NextDate function, we could use the set of equivalence classes as they were in Chapter 6. These classes have a Cartesian product that contains 36 triples, with several that are impossible.

To illustrate another decision table technique, this time we will develop an extended entry decision table, and we will take a closer look at the action stub. In making an extended entry decision table, we must ensure that the equivalence classes form a true partition of the input domain. (Recall from Chapter 3 that a partition is a set of disjoint subsets where the union is the entire set.) If there were any "overlaps" among the rule entries, we would have a redundant case in which more than one rule could be satisfied. Here, Y2 is the set of years between 1812 and 2012, evenly divisible by four excluding the year 2000.

M1 = {month : month has 30 days}
M2 = {month : month has 31 days}
M3 = {month : month is February}
D1 = {day : 1 ≤ day ≤ 28}
D2 = {day : day = 29}
D3 = {day : day = 30}
D4 = {day : day = 31}
Y1 = {year : year = 2000}
Y2 = {year : year is a leap year}
Y3 = {year : year is a common year}

In a sense, we could argue that we have a "gray box" technique, because we take a closer look at the NextDate function. In order to produce the next date of a given date, only five possible manipulations can be used: incrementing and resetting the day and month, and incrementing the year. (We will not let time go backward by resetting the year.)

These conditions would result in a decision table with 36 rules that correspond to the Cartesian product of the equivalence classes. Combining rules with don't care entries yields the decision table in 7.13, which has 17 rules. We still have

Table 7.13 Second Try Decision Table with 36 Rules

	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*
c1: month in	M1	M1	M1	M1	M2	M2	M2	M2
c2: day in	D1	D2	D3	D4	D1	D2	D3	D4
c3: year in	—	—	—	--	—	—	—	—
Rule count	3	3	3	3	3	3	3	3
actions								
a1: impossible				X				
a2: increment day	X	X			X	X	X	
a3: reset day			X					X
a4: increment month			X					?
a5: reset month								?
a6: increment year								?

	9	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*
c1: month in	M3	M3	M3	M3	M3	M3	M3	M3	M3
c2: day in	D1	D1	D1	D2	D2	D2	D3	D3	D4
c3: year in	Y1	Y2	Y3	Y1	Y2	Y3	—	—	—
Rule count	1	1	1	1	1	1	3	3	3
actions									
a1: impossible						X	X	X	X
a2: increment day		X							
a3: reset day	X		X	X	X				
a4: increment month	X		X	X	X				
a5: reset month									
a6: increment year									

the problem with logically impossible rules, but this formulation helps us identify the expected outputs of a test case. If you complete the action entries in this table, you will find some cumbersome problems with December (in rule 8). We fix these next.

7.3.3 Third Try

We can clear up the end-of-year considerations with a third set of equivalence classes. This time, we are very specific about days and months, and we revert to the simpler leap year or non-leap year condition of the first try — so the year 2000 gets no special attention. (We could do a fourth try, showing year equivalence classes as in the second try, but by now you get the point.)

M1 = {month : month has 30 days}
M2 = {month : month has 31 days except December}
M3 = {month : month is December}
M4 = {month : month is February}
D1 = {day : 1 ≤ day ≤ 27}
D2 = {day : day = 28}
D3 = {day : day = 29}
D4 = {day : day = 30}
D5 = {day : day = 31}
Y1 = {year : year is a leap year}
Y2 = {year : year is a common year}

The Cartesian product of these contains 40 elements. The result of combining rules with don't care entries is given in Table 7.14; it has 22 rules, compared with the 36 of the second try. Recall from Chapter 1 the question of whether a large set of test cases is necessarily better than a smaller set. Here, we have a 22-rule decision table that gives a clearer picture of the NextDate function than does the 36-rule decision table. The first five rules deal with 30-day months; notice that the leap year considerations are irrelevant. The next two sets of rules (6 – 10 and 11 – 15) deal with 31-day months, where the first five deal with months other than December and the second five deal with December. No impossible rules are listed in this portion of the decision table, although there is some redundancy that an efficient tester might question. Eight of the ten rules simply increment the day. Would we really require eight separate test cases for this subfunction? Probably not; but note the insights we can get from the decision table. Finally, the last seven rules focus on February and leap year.

The decision table in 7.14 is the basis for the source code for the NextDate function in Chapter 2. As an aside, this example shows how good testing can improve programming. All the decision table analysis could have been done during the detailed design of the NextDate function.

We can use the algebra of decision tables to further simplify these 22 test cases. If the action sets of two rules in a decision table are identical, there must be at least one condition that allows two rules to be combined with a don't care entry. This is the decision table equivalent of the "treated the same" guideline that we used to identify equivalence classes. In a sense, we are identifying equivalence classes of rules. For example, rules 1, 2, and 3 involve day classes

Table 7.14 Decision Table for the NextDate Function

	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
c1: month in	M1	M1	M1	M1	M1	M2	M2	M2	M2	M2
c2: day in	D1	D2	D3	D4	D5	D1	D2	D3	D4	D5
c3: year in	—	—	—	—	—	—	—	—	—	—
actions										
a1: impossible					X					
a2: increment day	X	X	X			X	X	X	X	
a3: reset day				X						X
a4: increment month				X						X
a5: reset month										
a6: increment year										

	11	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*	*20*	*21*	*22*
c1: month in	M3	M3	M3	M3	M3	M4	M4	M4	M4	M4	M4	M4
c2: day in	D1	D2	D3	D4	D5	D1	D2	D2	D3	D3	D4	D5
c3: year in	—				—	—	Y1	Y2	Y1	Y2	—	—
actions												
a1: impossible										X	X	X
a2: increment day	X	X	X	X		X	X					
a3: reset day					X			X	X			
a4: increment month								X	X			
a5: reset month					X							
a6: increment year					X							

D1, D2, and D3 for 30-day months. These can be combined similarly for day classes D1, D2, D3, and D4 in the 31-day month rules, and D4 and D5 for February. The result is in Table 7.15.

The corresponding test cases are shown in Table 7.16.

7.4 Test Cases for the Commission Problem

The commission problem is not well served by a decision table analysis. This is not surprising because very little decisional logic is used in the problem. Because the variables in the equivalence classes are truly independent, no impossible rules will occur in a decision table in which conditions correspond to the equivalence classes. Thus, we will have the same test cases as we did for equivalence class testing.

7.5 Guidelines and Observations

As with the other testing techniques, decision table-based testing works well for some applications (such as NextDate) and is not worth the trouble for others (such as the Commission Problem). Not surprisingly, the situations in which it

Table 7.15 Reduced Decision Table for the NextDate Function

	1–3	*4*	*5*	*6–9*	*10*
c1: month in	M1	M1	M1	M2	M2
c2: day in	D1, D2, D3	D4	D5	D1, D2, D3, D4	D5
c3: year in	—	—	—	—	—
actions					
a1: impossible			X		
a2: increment day	X			X	
a3: reset day		X			X
a4: increment month		X			X
a5: reset month					
a6: increment year					

	11–14	*15*	*16*	*17*	*18*	*19*	*20*	*21, 22*
c1: month in	M3	M3	M4	M4	M4	M4	M4	M4
c2: day in	D1, D2, D3, D4	D5	D1	D2	D2	D3	D3	D4, D5
c3: year in	—	—	—	Y1	Y2	Y1	Y2	—
actions								
a1: impossible							X	X
a2: increment day	X		X	X				
a3: reset day		X			X	X		
a4: increment month					X	X		
a5: reset month		X						
a6: increment year		X						

Table 7.16 Decision Table Test Cases for NextDate

Case ID	*Month*	*Day*	*Year*	*Expected Output*
1–3	April	15	2001	April 16, 2001
4	April	30	2001	May 1, 2001
5	April	31	2001	Impossible
6–9	January	15	2001	January 16, 2001
10	January	31	2001	February 1, 2001
11–14	December	15	2001	December 16, 2001
15	December	31	2001	January 1, 2002
16	February	15	2001	February 16, 2001
17	February	28	2004	February 29, 2004
18	February	28	2001	March 1, 2001
19	February	29	2004	March 1, 2004
20	February	29	2001	Impossible
21, 22	February	30	2001	Impossible

works well are those in which a lot of decision making takes place (such as the Triangle Problem), and those in which important logical relationships exist among input variables (the NextDate function).

1. The decision table technique is indicated for applications characterized by any of the following:
   - Prominent if-then-else logic
   - Logical relationships among input variables
   - Calculations involving subsets of the input variables
   - Cause-and-effect relationships between inputs and outputs
   - High cyclomatic (McCabe) complexity (see Chapter 9)
2. Decision tables do not scale up very well (a limited entry table with n conditions has 2^n rules.) There are several ways to deal with this — use extended entry decision tables, algebraically simplify tables, "factor" large tables into smaller ones, and look for repeating patterns of condition entries. For more on these techniques, see Topper (1993).
3. As with other techniques, iteration helps. The first set of conditions and actions you identify may be unsatisfactory. Use it as a stepping stone, and gradually improve on it until you are satisfied with a decision table.

References

Elmendorf, William R., *Cause–Effect Graphs in Functional Testing,* IBM System Development Division, Poughkeepsie, NY, TR-00.2487, 1973.

Mosley, Daniel J., *The Handbook of MIS Application Software Testing,* Yourdon Press, Prentice Hall, Englewood Cliffs, NJ, 1993.

Myers, Glenford J., *The Art of Software Testing,* Wiley Interscience, New York, 1979.

Topper, Andrew et al., *Structured Methods: Merging Models, Techniques, and CASE,* McGraw-Hill, New York, 1993.

Exercises

1. Develop a decision table and additional test cases for the right triangle addition to the Triangle Problem (see Chapter 2 exercises.) Note that there can be isosceles right triangles, but not with integer sides.
2. Develop a decision table for the "second try" at the NextDate function. At the end of a 31-day month, the day is always reset to 1. For all non-December months, the month is incremented; and for December, the month is reset to January, and the year is incremented.
3. Develop a decision table for the YesterDate function (see Chapter 2 exercises).
4. Expand the Commission Problem to consider "violations" of the sales limits. Develop the corresponding decision tables and test cases for a "company friendly" version and a "salesperson friendly" version.
5. Discuss how well decision table testing deals with the multiple fault assumption.
6. Develop decision table test cases for the time change problem (Chapter 6, Problem 5.)

Chapter 8

Retrospective on Functional Testing

In the preceding three chapters, we studied as many types of functional testing. The common thread among these is that all view a program as a mathematical function that maps its inputs onto its outputs. With the boundary-based approaches, test cases are identified in terms of the boundaries of the ranges of the input variables, and variations give us four techniques — boundary value analysis, robustness testing, worst-case testing, and robust worst-case testing. We next took a closer look at the input variables, defining equivalence classes in terms of values that should receive "similar treatment" from the program being tested. Four forms of equivalence class testing are used — weak normal, strong normal, weak robust, and strong robust. The goal of examining similar treatment is to reduce the sheer number of test cases generated by the domain-based techniques. We pushed this a step further when we used decision tables to analyze the logical dependencies imposed by the function of the program. Whenever we have a choice among alternatives, we naturally want to know which is preferred — or at least how to make an informed choice. In this chapter, we look at questions about testing effort, testing efficiency, and then try to get a handle on test effectiveness.

8.1 Testing Effort

Let us return to our craftsperson metaphor for a minute. We usually think of such people as knowing their crafts so well that their time is spent very effectively. Even if it takes a little longer, we like to think that the time is well spent. We are finally in a position to see a hint of this as far as testing techniques are concerned. The functional methods we have studied vary both in terms of the number of test cases generated and the effort to develop these test cases. Figures 8.1 and 8.2 show the general trends, but the sophistication axis needs some explanation.

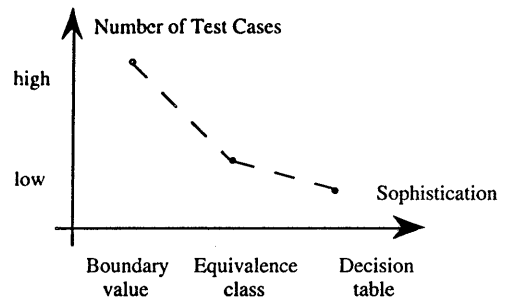

Figure 8.1 Trendline of test cases per testing method.

The domain-based techniques have no recognition of data or logical dependencies; they are very mechanical in the way they generate test cases. Because of this, they are also easy to automate. The equivalence class techniques pay attention to data dependencies and to the function itself. More thought is required to use these techniques — also more judgment, or craft. The thinking goes into the identification of the equivalence classes; after that, the process is also mechanical. The decision table technique is the most sophisticated, because it requires the tester to consider both data and logical dependencies. As we saw in our examples, you might not get the conditions of a decision table right on the first try; but once you have a good set of conditions, the resulting test cases are both complete and, in some sense, minimal.

The end result is a satisfying trade-off between test identification effort and test execution effort: methods that are easy to use generate numerous test cases, which in turn, are more time-consuming to execute. If we shift our effort toward more sophisticated testing methods, we are repaid with less test execution time. This is particularly important because tests are typically executed several times. We might also note that judging testing quality in terms of the sheer number of test cases has drawbacks similar to judging programming productivity in terms of lines of code.

Our examples bear out the trends of Figures 8.1 and 8.2. The following three graphs (Figures 8.2–8.5) are taken from a spreadsheet that summarized the number

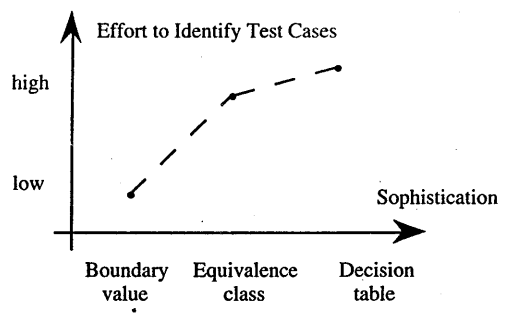

Figure 8.2 Trendline of test case identification effort per testing method.

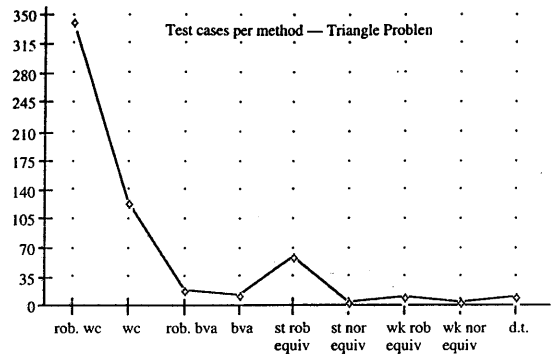

Figure 8.3 Test case trendline for the triangle problem.

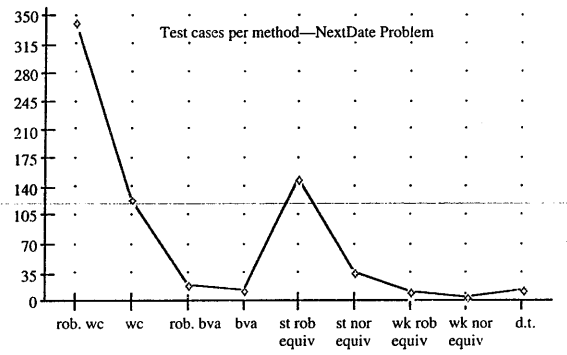

Figure 8.4 Test case trendline for the NextDate problem.

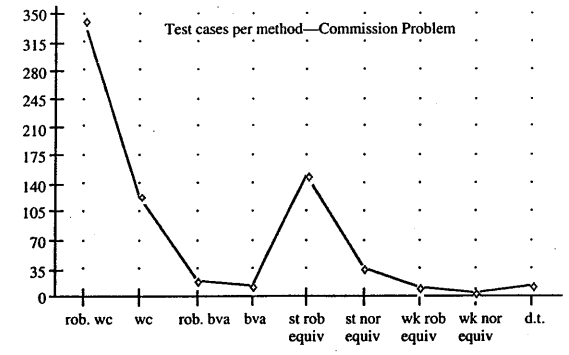

Figure 8.5 Test case trendline for the commission problem.