FIFTH EDITION

Software Testing

A Craftsman's Approach

Paul C. Jorgensen • Byron DeVries

CRC Press
Taylor & Francis Group
AN AUERBACH BOOK

Software Testing

Software Testing

A Craftsman's Approach

Fifth Edition

Paul C. Jorgensen and Byron DeVries

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business
AN AUERBACH BOOK

Fifth edition published [2021]
by CRC Press
6000 Broken Sound Parkway NW, Suite 300, Boca Raton, FL 33487-2742

and by CRC Press
2 Park Square, Milton Park, Abingdon, Oxon, OX14 4RN

[First edition published by CRC Press 1995]
[Fourth edition published by CRC Press 2014]

CRC Press is an imprint of Taylor & Francis Group, LLC

ISBN: 978-0-367-35849-5 (hbk)
ISBN: 978-0-367-76762-4 (pbk)
ISBN: 978-1-003-16844-7 (ebk)

Typeset in Garamond
by SPi Global, India

To Carol, Kirsten, and Katia; Angela, Bryce, and Wesley

Contents

PART III Beyond Unit Testing

Preface

The fifth edition of *Software Testing—A Craftsman's Approach* appears 25 years after the first edition and now there is a co-author, Dr. Byron DeVries. Together, Paul and Byron have 32 years of industrial experience and a few more years of university teaching and research in Software Engineering. Paul's testing experience is on telephone switching systems software; Byron's is on avionics systems.

The book has evolved over four editions and 25 years of classroom and industrial use. We continue the pattern of presenting theory, using it to describe testing techniques, and illustrating all of this with carefully selected examples. We retained some of the classical examples and replaced others with a comprehensive web-based example, the Foodies Wish List, that is used throughout where appropriate. This lends a unifying "leitmotif" to our book.

Here are some the highlights of the Fifth Edition…

- This book now has a website, softwaretestcraft.org (also .com) that contains all Java code, powerpoint presentations, and various notes.
- Parts 2, 3, and 4 are essentially object-oriented. All pseudo-code examples are now converted to Java. Unit testing examples use JUnit.
- We included specific information on commercial and open-source tools for code-based testing. Also, we added three examples of commercial Model-Based Testing products to the Model-Based Testing chapter.
- Testing object-oriented software is consolidated in a single chapter.
- There is a new chapter on the feature interaction problem.
- There is a new emphasis (and example) for modeling and testing event-driven systems.
- We retained the chapter on technical inspections and the corresponding appendix.

Some things have remained constant across all five editions. In the Preface to the First Edition, Paul wrote:

> *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.*

For the past 25 years, the goal of this book is to provide you with a better set of crystals. As the title suggests, we believe that software (and system) testing is a craft, and we have some mastery of that craft. We bring our combined industrial and academic backgrounds to the theory, techniques, and examples. We hope that all of this will crystalize into your software testing craft.

Paul C. Jorgensen
Rockford, Michigan

Byron DeVries
Grand Rapids, Michigan
December, 2020

Authors

Paul Jorgensen, Ph.D., spent his 20-year first career in all phases of software development for the research and development laboratory of a telephone switching systems company. He began his university career in 1986 teaching graduate courses in software engineering at Arizona State University, and since 1988, at Grand Valley State University where he is a full professor. Paul retired from the university in the summer of 2017 and is now a Professor Emeritus. He jokes that he has seven-day weekends, every week. This schedule permits a lot of family contact and also allows time for his consulting business, Software Paradigms. He has served on major CODASYL, ACM, IEEE, and ISTQB committees, and in 2012, his university recognized his lifetime accomplishments with its "Distinguished Contribution to a Discipline Award."

In addition to this software testing book, he is also the author of *Modeling Software Behavior: A Craftsman's Approach* and *The Craft of Model-Based Testing*. He is a co-author of *Mathematics for Data Processing* (McGraw-Hill, 1970) and *Structured Methods--Merging Models, Techniques, and CASE* (McGraw-Hill, 1993).

Living and working in Italy for three years made him a confirmed "Italophile." He, his wife Carol, and daughters Kirsten and Katia have visited friends there several times. Paul and Carol have volunteered at the Porcupine School on the Pine Ridge Reservation in South Dakota for 19 years. His preferred email addresses are jorgensp@gvsu.edu; paul@softwaretestcraft.org

Byron DeVries, Ph.D., has taught undergraduate and graduate software engineering courses at Grand Valley State University since he joined the faculty in 2017 as an assistant professor. Prior to teaching, he spent over a dozen years in a variety of avionics software development roles, often focused on verification. He actively publishes in and serves on program committees for a variety of IEEE and ACM conferences. In 2021, he was recognized by his university with the "Distinguished Early Career Scholar Award."

In the summers, you can most often find him either close to, or on, the water around West Michigan with his wife, Angela. Though an avid sailor, he begrudgingly spends more time on power boats for the sake of his two young boys: Bryce and Wesley. You can reach him at his email addresses: devrieby@gvsu.edu and byron@softwaretestcraft.org.

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 framework within which we can examine software testing.

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 International Software Testing Qualification Board (ISTQB) has an extensive glossary of testing terms (see the website http://www.istqb.org/downloads/glossary.html). The terminology here (and throughout this book) is compatible with the ISTQB definitions, and they, in turn, are compatible with the standards developed by the Institute of Electronics and Electrical Engineers (IEEE) Computer Society. To get started, here is 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, UML diagrams, hierarchy charts, source code, and so on. Defect (see ISTQB Glossary) is a good synonym for fault, as is bug. Faults can be elusive. An error of omission results in a fault is that something is missing that should be present in the representation. This suggests a useful refinement, 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.

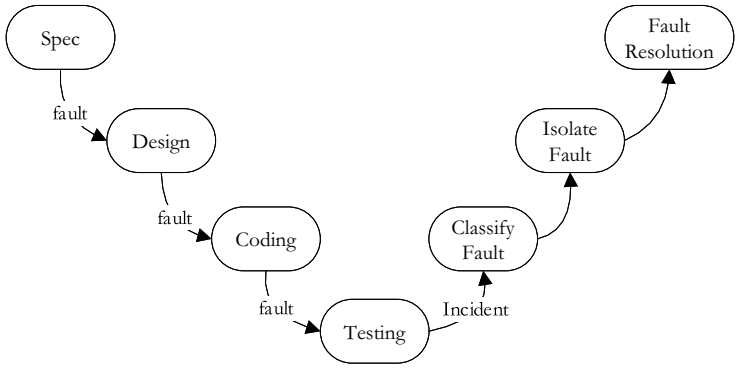

Figure 1.1 A testing life cycle.

Failure—a failure occurs when the code corresponding to a fault executes. Two subtleties arise here: one is that failures only occur in an executable representation, which is usually 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? Technical reviews (see Chapter 20) 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 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 may propagate through the remainder of the development process. 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. A test case is (or should be) a recognized work product. A complete test case will contain a test case identifier, a brief statement of purpose (*e.g.,* a business rule),

a description of pre-conditions, the actual test case inputs, the expected outputs, a description of expected post-conditions, and an execution history. The execution history is primarily for test management use—it may contain the date when the test was run, the person who ran it, the version on which it was run, and the Pass/Fail result.

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 that "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.

Test case execution entails establishing the necessary preconditions, providing the test case inputs, observing the outputs, comparing these with the expected outputs, and then ensuring that the expected post-conditions exist to determine whether the test passed. 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 code-based view common to software (and system) developers. A quick distinction is that the code-based view focuses on what it ***is*** and the behavioral view considers what it ***does***. One of the continuing sources of difficulty for testers is that the base documents are usually written by and for developers; the emphasis is therefore on code-based, 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 1.2 shows the relationship between the specified and programmed behaviors. Of all the possible program behaviors, the specified ones are in the circle labeled S; and all those behaviors

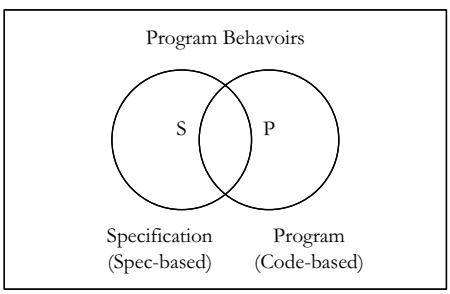

Figure 1.2 Specified and implemented program behaviors.

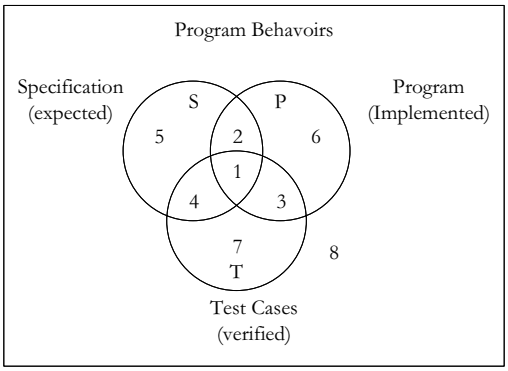

Figure 1.3 Specified, implemented, and tested behaviors.

actually programmed 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 1.3 is for test cases. Notice the slight discrepancy with our universe of discourse and the set of program behaviors. Because a test case results in 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 behaviors that were not implemented (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, some possibilities arise: either such a test case is unwarranted, the specification is deficient, or the tester wishes to determine that specified non-behavior does not occur. (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.)

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 Chapter 10.

1.4 Identifying Test Cases

Two fundamental approaches are used to identify test cases; for decades, these have been called functional and structural testing. Why functional? In a sense, a program is a function that maps elements of its input space to elements of its output space. The "structural" part is less clear—to be generous, it might refer to the structure of the code being tested. Specification-based and code-based are more descriptive names, and they will be used here. Both approaches have several distinct test case identification methods, they are generally just called testing methods. They are methodical in the sense that two testers following the same "method" will devise very similar (equivalent?) test cases.

1.4.1 Specification-based Testing

The reason that specification-based testing was originally called "functional testing" is 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 led to another synonymous term—black box testing, in which the content (implementation) of the black box is not known, and the function of the black box is understood completely in terms of its inputs and outputs (see Figure 1.4). 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 specification-based approach to test case identification, the only information used is the specification of the software. Therefore, the test cases have two distinct advantages: (1) they are independent of how the software is implemented, so if 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, specification-based test cases frequently suffer from two problems: significant redundancies may exist among test cases, compounded by the possibility of gaps of untested software.

Figure 1.5 shows the results of test cases identified by two specification-based 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 specification-based methods are based on the specified behavior, it is hard to imagine these methods identifying behaviors that are not specified. In Chapter 10, we will see direct comparisons of test cases generated by various specification-based methods for the examples defined in Chapter 2.

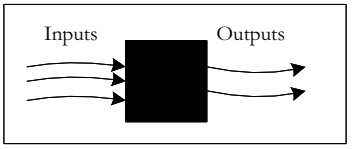

Figure 1.4 An engineer's black box.

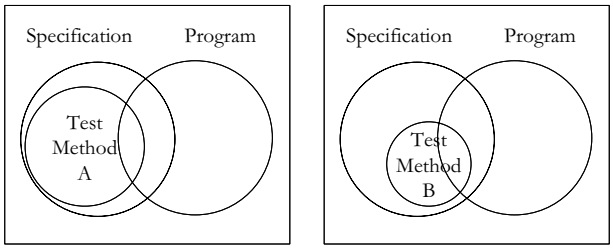

Figure 1.5 Comparing specification-based test case identification methods.

In Chapters 5, 6, and 7, we will examine the mainline approaches to specification-based testing, including boundary value analysis, robustness testing, worst-case analysis, special value testing, input (domain) 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. Some of the mathematical background presented in Chapter 3 applies primarily to the specification-based approaches.

1.4.2 Code-based Testing

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

Code-based testing has been the subject of some fairly strong theory. To really understand code-based 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, code-based 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 1.6 shows the results of test cases identified by two code-based 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 code-based 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

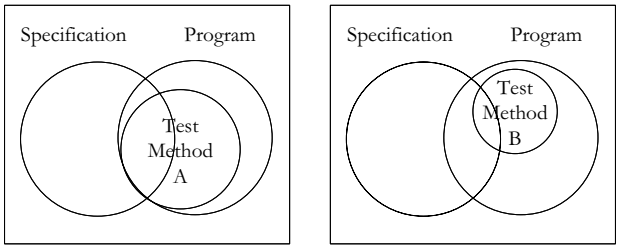

Figure 1.6 Comparing code-based test case identification methods.

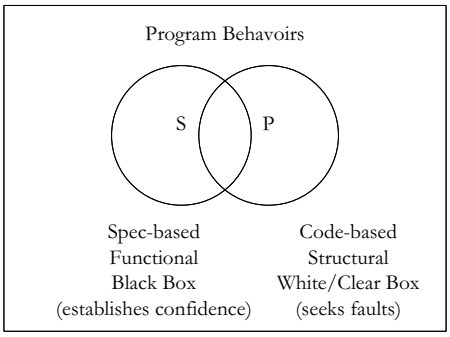

Figure 1.7 Sources of test cases.

behavior. Because code-based 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 code-based test cases is relatively small with respect to the full set of programmed behaviors. In Chapter 10, we will see direct comparisons of test cases generated by various code-based methods.

1.4.3 The Specification-based versus Code-based Debate

Given the 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.

The Venn diagrams presented earlier yield a strong resolution to this debate. Recall that the goal of both approaches is to identify test cases. Specification-based testing uses only the specification to identify test cases, while code-based testing uses the program source code (implementation) as the basis of test case identification. Later chapters will establish that neither approach by itself is sufficient. Consider program behaviors: if all specified behaviors have not been implemented, code-based 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 Specification-based test cases. (A Trojan Horse 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 specification-based testing and the measurement of code-based testing. Earlier, we asserted that specification-based testing often suffers from twin problems of redundancies and gaps. When specification-based test cases are executed in combination with code-based test coverage metrics, both of these problems can be recognized and resolved.

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 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 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 (fault) occurrence: one time only, intermittent, recurring, or repeatable.

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 we have added some of our favorites.

Since the primary purpose of a software review is to find faults, review checklists (see Chapter 20) are another good source of fault classifications. Karl Wiegers has an excellent set of checklists on his website: [http://www.processimpact.com/pr_goodies.shtml].

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

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 identifying distinct levels of testing and for clarifying the objectives that pertain to each level. A diagrammatic variation of the Waterfall Model, known as the V-Model in ISTQB parlance, is given in Figure 1.8; this variation emphasizes the correspondence between testing and design levels. Notice that, especially in terms of Specification-based testing, the three levels of definition (specification, preliminary design, and detailed design) correspond directly to three levels of testing—system, integration, and unit testing.

A practical relationship exists between levels of testing versus Specification-based and code-based testing. Most practitioners agree that code-based testing is most appropriate at the unit level, while Specification-based 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 code-based testing make the most sense at the unit level, and similar constructs are only now becoming

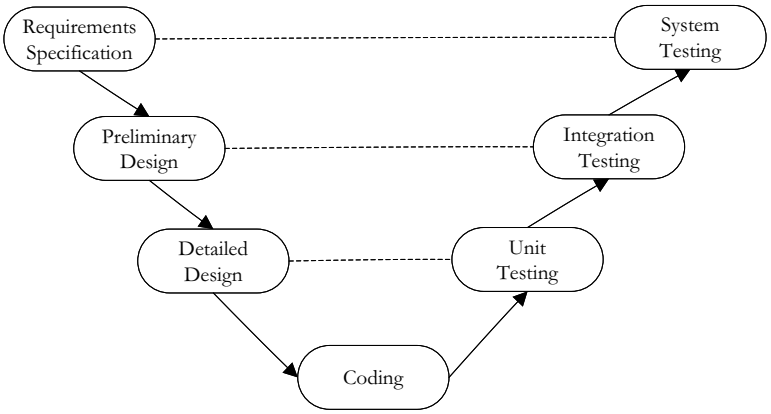

Figure 1.8 Levels of abstraction and testing in the Waterfall Model.

available for the integration and system levels of testing. We develop such structures in Chapters 9, 12, and 13 to support code-based testing at the integration and system levels for both traditional and object-oriented software.

Exercises

1.1 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 …"

1.2 Make a Venn Diagram that reflects the essence of Reinhold Niebuhr's "Serenity Prayer":

> God, grant me the serenity to accept the things I cannot change,
> Courage to change the things I can,
> And wisdom to know the difference.

1.3 Describe each of the eight regions in Figure 1.3. Can you recall examples of these in software you have written?

1.4 One of the folk tales of software lore describes a disgruntled employee who writes a payroll program which 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.

1.5 Figure 1.9 shows the V-Model (aka the Waterfall Model) phases in which mistakes might be made, thereby becoming faults. Try to map the faults in Tables 1.1 through 1.5 into the "fault insertion" phases in Figure 1.9.

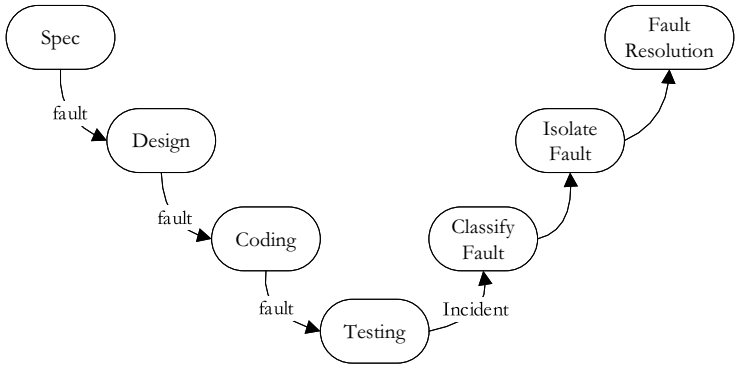

Figure 1.9 Possibilities for Fault Insertion in the V-Model.

References

IEEE Computer Society, *IEEE Standard Glossary of Software Engineering Terminology*, 4th Edition. 1983, ANSI/IEEE Std 729–1983.

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

Chapter 2

Examples

Three examples will be used in Chapters 5 through 10 to illustrate the various unit testing methods: the triangle problem (a venerable example in testing circles); a logically complex function, NextDate; and an online shopping example that typifies MIS applications, Foodies-Wish-List. Taken together, these examples raise most of the issues that testers will encounter at the unit level. The discussion of higher levels of testing in Chapters 11 through 17 uses a garage door controller example which also illustrates some of the issues of "systems of systems." Finally, Section 2.6 describes three examples that will be used in the exercise portions of selected chapters.

For the purposes of code-based testing, Java implementations of the three unit-level examples are given in this chapter. System-level descriptions of the garage door controller are given in Chapters 11 through 17. These applications are modeled with finite state machines, variations of Event-Driven Petri Nets, selected statecharts, and with the Universal Modeling Language (UML).

2.1 Structural Elements of Pseudo-code and Java

Previous editions of this book used pseudo-code as an "implementation" of code examples. Most of the pseudo-code has been rewritten as Java code. The pseudo-code was deliberately similar to Visual Basic for Applications (VBA). Tables 2.1 and 2.2 show most language constructs in "VBA' form and in Java.

Table 2.1

Comments	"VBA"	' <text>
	Java	//<text>
Data Structure / Class Declaration	"VBA"	Type <type name><list of field descriptions>
	Java	public class <class name> {<list of data declarations>

(*Continued*)

Table 2.1 (Continued)

Data Declaration	"VBA"	Dim <variable list> As <type>
	Java	<type> <variable list>;
Input/Output	"VBA"	Input (<variable list>) Output (<variable list>)
	Java	NA
Variable Naming	"VBA" and Java	A sequence of alphanumeric (and selected special) characters with no length limit. Descriptive names are preferred. By convention, variable names begin with a lower case letter. If the variable name consists of two or more words, the first letter of each successive word is capitalized, *e.g.*,, accountBalance.
Binary Arithmetic Operators Same for both "VBA" and Java		
Addition	both	+
Subtration	both	-
Multiplication	both	*
Division	both	/
Remainder	both	%
Unary Arithmetic Operators, Java only		
Positive value	Java	+
Negative value	Java	-
Increment by 1	Java	++
Decrement by 1	Java	--
Logical complement	Java	! (reverses value of a boolean variable)
Relational Operators		
Equals	"VBA"	=
	Java	==
Not Equals	"VBA"	<>
	Java	!=
Greater than	both	>
Greater than or equal	both	>=
Less than	both	<
Less than or equal	both	<=

(*Continued*)

Table 2.1 (Continued)

Conditional Operators		
Conjunction	"VBA"	AND
	Java	&&
Disjunction	"VBA"	OR
	Java	||
Negation	"VBA"	NOT
	Java	!
Expressions		In both "VBA" and java, an expression can be a single variable, a single procedure, (or method invocation) or a compound built out of these with operators.
Assignment Statement	both	<variable> = <expression>

Table 2.2

Control Flow Statements (usually more than one line)		
Conditional Statement	*"VBA"*	*Java*

If-then	If <condition>
	Then <block of statements>
	EndIf
		if <condition> {
		<block of statements> ;
		}

If-then-else	If <condition>
	Then <block of statements>
	Else <block of statements>
	EndIf
		if <condition> {
		<block of statements> ;
		} else {
		<block of statements> ;
		}

If-ElseIF	If <condition>
	Then <block of statements>
	ElseIF <block of statements>
	ElseIF <block of statements>
	...
	EndIf
		if <condition> {
		<block of statements> ;
		} else if <condition> {
		<block of statements> ;
		} else if <condition> {
		<block of statements> ;
		}

(*Continued*)

Table 2.2 (Continued)

Mutual Exclusive Alternatives	Case <variable> of
	Case 1 variable = value
	Case 2 variable = value
	Case 3 variable = value
	End Case
		switch <variable> {
		case 1: <block of statements>
		break;
		case 2: <block of statements>
		break;
		}

Pre-test loop	While <condition>
	<block of statements>
	EndWhile
		while <condition> {
		<block of statements>
		}

For (also a pre-test loop)	For Index = first, last, increment
	<block of statements>
	EndFor
		for(<type> index = first, index <= last, index++) {
		<block of statements>
		}

Post-test loop	Do <block of statements> Until <condition>	do { <block of statements> } while <condition> ;
Other (Java only) Sequence-changing Statements		
Branching Statement	Java (description)	
break	Terminates a switch or repetition	
continue	Terminates innermost repetition, Then continues the loop	
return <value>	Returns <value> and exits from a method	
return	Exits from a void method	
Procedure/Method Definition		
	"VBA"	Java
	Procedure <procedure name> (Input: <list of variables>;Output: <list of variables>) <body>	<modifier> <return type> methodName (<parameter list>) {modifiers: public, private, protected; return type is the type of value returned (items in the parameter list are preceded by their type
	End <procedure name>	}

(*Continued*)

Table 2.2 (Continued)

Functions	Function functionName(<parameter list>)	NA
	a Function is treated as a variable, *e.g., x = squareRoot(49*	
Inter-unit Communication	Call procedureName(<parameter list>	A message can be treated as a variable
Class/Object Definition		
	<name> (<attribute list>; <method list>, <body>) End <name>	public class <class name> {<list of data declarations>}
Object Instantiation		
	Instantiate <class name>.<object name> (list of attribute values)	<class name> <object name> = new <class name>(<parameter list>);

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 four decades of testing literature are Gruenberger (1973); Brown (1975); Myers (1979); Pressman (1982) and subsequent editions; Clarke (1983); Clarke (1984); Chellappa (1987); and Hetzel (1988). There are others, but this list makes the point.

2.2.1 Problem Statement

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 Not A Triangle. Sometimes this problem is extended to include right triangles as a fifth type; we will use this extension in some of the exercises.

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.	$1 \le a \le 200$	c4.	$a < b + c$
c2.	$1 \le b \le 200$	c5.	$b < a + c$
c3.	$1 \le c \le 200$	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 conditions c1, c2, or c3, 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 is not met, 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 Inequality: 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.

We use this example because:

- it is well-known in the software testing literature.
- it is easy to determine expected outputs, and
- it is an easy example for infeasible program paths.

The Java code is given next. Note, for this and other examples, Java source code will be written in Monaco 8.5 font.

2.2.3 Java Implementation

```
public class Triangle {

       public static final int OUT_OF_RANGE = -2;
       public static final int INVALID = -1;
       public static final int SCALENE = 0;
       public static final int ISOSELES = 1;
       public static final int EQUILATERAL = 2;

       public static int triangle(int a, int b, int c) {

              boolean c1, c2, c3, isATriangle;

              // Step 1: Validate Input
              c1 = (1 <= a) && (a <= 200);
              c2 = (1 <= b) && (b <= 200);
              c3 = (1 <= c) && (c <= 200);

              int triangleType = INVALID;
              if (!c1 || !c2 || !c3)
                     triangleType = OUT_OF_RANGE;
              else {
                     // Step 2: Is A Triangle?
                     if ((a < b + c) && (b < a + c) && (c < a + b))
                            isATriangle = true;
                     else
                            isATriangle = false;
```

```
                    // Step 3: Determine Triangle Type
                    if (isATriangle) {
                          if ((a == b) && (b == c))
                                triangleType = EQUILATERAL;
                          else if ((a != b) && (a != c) && (b != c))
                                triangleType = SCALENE;
                          else
                                triangleType = ISOSELES;
                    } else
                          triangleType = INVALID;
              }
              return triangleType;
        }
}
```

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 Statement

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 (the year range ending in 2042 is arbitrary, and is from the First Edition):

c1. $1 \leq \text{month} \leq 12$
c2. $1 \leq \text{day} \leq 31$
c3. $1842 \leq \text{year} \leq 2042$

As we did with the triangle program, we could make our problem statement more specific. This entails defining responses for invalid values of the input values for the day, month, and year. We could also define responses for invalid combinations of inputs, 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 1996,

2016, 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 Java Implementation

```
public class NextDate {

      public static SimpleDate nextDate(SimpleDate date) {

            int tomorrowDay, tomorrowMonth, tomorrowYear;

            tomorrowMonth = date.month;
            tomorrowDay = date.day;
            tomorrowYear = date.year;
            switch (date.month) {

            // 31 day months (except Dec.)
            case 1:
            case 3:
            case 5:
            case 7:
            case 8:
            case 10:
                  if (date.day < 31)
                        tomorrowDay = date.day + 1;
                  else {
                        tomorrowDay = 1;
                        tomorrowMonth = date.month + 1;
                  }
                  break;

            // 30 day months
            case 4:
            case 6:
            case 9:
            case 11:
                  if (date.day < 30)
                        tomorrowDay = date.day + 1;
                  else {
                        tomorrowDay = 1;
                        tomorrowMonth = date.month + 1;
                  }

                  break;
            // December
            case 12:
                  if (date.day < 31)
                        tomorrowDay = date.day + 1;
                  else {
                        tomorrowDay = 1;
                        tomorrowMonth = 1;
                        if (date.year == 2042)
                              System.out.println("Date beyond 2042 ");
                        else
```

```
                                tomorrowYear = date.year + 1;
                        }
                        break;
                // February
                case 2:
                        if (date.day < 28)
                                tomorrowDay = date.day + 1;
                        else {
                                if (date.day == 28) {
                                        if (date.isLeap())
                                                tomorrowDay = 29;
                                        else {
                                                tomorrowDay = 1;
                                                tomorrowMonth = 3;
                                        }
                                } else if(date.day == 29) {
                                        tomorrowDay = 1;
                                        tomorrowMonth = 3;
                                }
                        }
                        break;
                }
                return new SimpleDate(tomorrowMonth, tomorrowDay,
tomorrowYear);
        }
}
public class SimpleDate {
        int month;
        int day;
        int year;
        public SimpleDate(int month, int day, int year) {
                if(!rangesOK(month, day, year))
                        throw new IllegalArgumentException("Invalid Date");
                this.month = month;
                this.day = day;
                this.year = year;
        }
        public int getMonth() {
                return month;
        }
        public void setMonth(int month) {
                this.month = month;
        }
        public int getDay() {
                return day;
        }
        public void setDay(int day) {
                this.day = day;
        }
        public int getYear() {
                return year;
        }
```

```
        public void setYear(int year) {
                this.year = year;
        }
        boolean rangesOK(int month, int day, int year) {
                boolean dateOK = true;
                dateOK &= (year > 1841) && (year < 2043); // Year OK?
                dateOK &= (month > 0) && (month < 13); // Month OK?
                dateOK &= (day > 0) && (
                                        ((month == 1 || month == 3 || month == 5
|| month == 7 || month == 8 || month == 10 || month == 12) && day < 32) ||
                                        ((month == 4 || month == 6 || month == 9
|| month == 11) && day < 31) ||
                                ((month == 2 && isLeap(year)) && day < 30) ||
                                ((month == 2 && !isLeap(year)) && day < 29));
                return dateOK;
        }
        private boolean isLeap(int year) {
                boolean isLeapYear = true;
                if(year % 4 != 0)
                        isLeapYear = false;
                else if(year % 100 != 0)
                        isLeapYear = true;
                else if(year % 400 != 0)
                        isLeapYear = false;
                return isLeapYear;
        }
        public boolean isLeap() {
                return isLeap(year);
        }
        @Override
        public boolean equals(Object obj) {
                boolean areEqual = false;
                if(obj instanceof SimpleDate) {
                        SimpleDate simpleDate = (SimpleDate) obj;
                        areEqual = simpleDate.getDay() == getDay() &&
                                        simpleDate.getMonth() == getMonth() &&
                                        simpleDate.getYear() == getYear();
                }
                return areEqual;
        }
}
```

2.4 The Foodies-Wish-List Online Shopping Application

Foodies-Wish-List is an online shopping service for extremely rare (and expensive!) gourmet foods. It can be used either on a one-time basis as a guest, or repeatedly by members. There is no initial cost for either category, but to be a Foodies-Wish-List member, one must register with customary information, such as:

- member name
- address

- shipping address
- telephone number
- email address
- preferred payment method
  - member credit card
  - PayPal

The registration process ends with an account number being assigned to the new Foodies-Wish-List member.

Registered Foodies-Wish-List members receive discounts based on the price of an individual order as follows:

- orders less than $200 receive no discount
- orders between $200 and $800 (inclusive) receive a 10% discount
- orders over $800 receive a 15% discount

There is no discount for any guest order.

Foodies-Wish-List members receive free shipping on any order over $200. For orders less than $200, there is a standard shipping price of $5.00. All guest orders are charged a $10 shipping fee.

Foodie Items

1. Vanilla beans; $112/pound
2. hop shoots; $128/pound
3. Jamon Iberico de Belotta; $220/pound
4. Kopi Luwak coffee; $200/pound
5. Kobe beef; $200/pound
6. Moose House cheese; $400 – $500/pound
7. Italian white truffles: $2000/pound
8. Saffron; $4540/pound, $10/gram
9. Almas caviar; $11,364/pound

2.4.1 Problem Statement

The full Foodies Wish List problem will be used as an integration testing and data-flow testing example. Here we only describe two parts of the problem—building an order and computing the final price.

2.4.2 Discussion

The public void method completeOrder is used to illustrate how Behavior-Driven Development (BDD) and decision tables can be combined to improve the bottom-up process inherent in agile software development.

The usual format for a BDD scenario uses key words Given, When, and Then. Here we move from this format directly into a partial decision table, and then expand the table using the mechanisms of a decision table.

Given: the running price total of an Order
And: the Order was placed by a Member,
When: the Member selects “Finish”

Then: compute discount
And: apply any taxes
And: apply shipping charges
And: open Payment Screen

The Given, When portions are modeled as conditions, and the Then portion is the action portion of the decision table.

c1. Member Order
c2. Order price < $20
c3. Member selects "Finish"
a1. no discount.
a2. 10% discount.
a3. 15% discount.
a4. apply any taxes.
a5. apply shipping charges.
a6. open Payment Screen.

This scenario yields the first rule of a decision table:

c1. Member Order	T
c2. Order price < $20	T
c3. Member selects "Finish"	T
a1. no discount	x
a2. 10% discount	—
a3. 15% discount	—
a4. apply any taxes	x
a5. apply shipping charges	x
a6. open Payment Screen.	x

Since this is a Limited Entry Decision Table (LEDT), we can mechanically expand it to the following (incomplete) decision table: (DT1).

c1. Member Order	T	T	T	T	F	F	F	F
c2. Order price < $200	T	T	F	F	T	T	F	F
c3. Member selects "Finish"	T	F	T	F	T	F	T	F
a1. no discount	x	—	—	—	—	—	—	—
a2. 10% discount	—	—	—	—	—	—	—	—
a3. 15% discount	—	—	—	—	—	—	—	—
a4. apply any taxes	x	—	—	—	—	—	—	—
a5. apply shipping charges	x	—	—	—	—	—	—	—
a6. open Payment Screen.	x	—	—	—	—	—	—	—

The mechanical expansion raises a few questions; these might be answered by additional BDD scenarios, or by discussion with the Customer. (Note that one of the values of a good model is that it stimulates discovery. One historical example: the Periodic Table of Elements predicted the existence of several chemical elements before they were shown to exist.) We can/should ask:

1. What does it mean for c1. Member order to be false?
2. What is so special about c2. Order price < $200?
3. What happens when c3. Member selects "Finish" is false?

Some possible answers are as follows:

1. More than one category of people submitting an order. For now, we will assume just one category, Non-member.
2. Actions a1, a2, and a3 suggest there are three ranges of order prices. For now, we could call them small, medium, or large.
3. Since this is an online shopping example, we might assume that when c3. Member selects "Finish" is false, the customer selects a "Order" screen. This, in turn, will create a new action, a7. Order. A simpler solution is to just not perform action a6.

c1. Member Order	T	T	T	T	F	F	F	F
c2. Order price < $200	T	T	F	F	T	T	F	F
c3. Member selects "Finish"	T	F	T	F	T	F	T	F
a1. no discount	x	—	—	—	—	—	—	—
a2. 10% discount	—	—	—	—	—	—	—	—
a3. 15% discount	—	—	—	—	—	—	—	—
a4. apply any taxes	x	—	—	—	—	—	—	—
a5. apply shipping charges	x	—	—	—	—	—	—	—
a6. open Payment Screen.	x	—	—	—	—	—	—	—
a7. open Order screen	—	x	—	x	—	x	—	x

With these assumptions, the Member half of our decision table is

c1. Order by	Member					
c2. Order price is	< $200		$200 to $800		> $800	
c3. Member selects "Finish"	T	F	T	F	T	F
a1. no discount	x	—	—	—	—	—

(*Continued*)

a2. 10% discount	—	—	x	—	—	—
a3. 15% discount	—	—	—	—	x	—
a4. apply any taxes	x	—	x	—	x	—
a5. apply shipping charges	x	—	x	—	x	—
a6. open Payment Screen.	x	—	x	—	x	—
a7. open Continue Shopping screen	—	x	—	x	—	x

and the non-Member half is (note change to "Guest"):

c1. Order by	Guest					
c2. Order price is	< $200		$200 to $800		> $800	
c3. Member selects "Finish"	T	F	T	F	T	F
a1. no discount	x	—	—	—	—	—
a2. 10% discount	—	—	—	—	—	—
a3. 15% discount	—	—	—	—	—	—
a4. apply any taxes	x	—	x	—	x	—
a5. apply shipping charges	x	—	x	—	x	—
a6. open Payment Screen.	x	—	x	—	x	—
a7. open Continue Shopping screen	—	x	—	x	—	x

At this point, the modeler must either seek new BDD scenarios or speak with the Customer. For now, assume the modeler learns that

1. guests receive no discount, regardless of order size, and
2. shipping charges are applied to any order of price < $200.

This reduces the Guest portion to:

c1. Order by	Guest	
c2. Order price is	—	
c3. Member selects "Finish"	T	F
a1. no discount	x	x

(*Continued*)

a2. 10% discount	—	—
a3. 15% discount	—	—
a4. apply any taxes	x	—
a5. apply shipping charges	x	x
a6. open Payment Screen.	x	—
a7. open Continue Shopping screen	—	x

And the final decision table is

c1. Order by	Member						Guest	
c2. Order price is	< $200		$200 to $800		> $800		—	
c3. Member selects "Finish"	T	F	T	F	T	F	T	F
a1. no discount	x	x	—	—	—	—	x	x
a2. 10% discount	—	—	x	—	—	—	—	—
a3. 15% discount	—	—	—	—	x	—	—	—
a4. apply any taxes	x	x	x	—	x	—	x	—
a5. apply shipping charges	x	—	—	—	—	—	x	—
a6. open Payment Screen.	x	—	x	—	x	—	x	—
a7. open Order screen	—	x	—	x	—	x	—	x

2.5 The Garage Door Controller

A system to open a garage door is comprised of several components: a drive motor, a drive chain, the garage door wheel tracks, end-of-track sensors, and a wireless control keypad. The garage door is controlled by the wireless keypad. In addition, there are two safety features, a laser beam near the floor, and an obstacle sensor. These latter two devices operate only when the garage door is closing. If the light beam is interrupted (possibly by a pet), the door immediately stops, and then reverses direction until the door is fully open. If the door encounters an obstacle while it is closing (say a child's tricycle left in the path of the door), the door stops and reverses direction until it is fully open. There is a third way to stop a door in motion, either when it is closing or opening—a signal from the wireless keypad. The response to this signal is different—the door stops in place. A subsequent signal starts the door in the same direction as when it was stopped. Finally, the end-of-track sensors detect when the door has moved to one of the extreme positions, either fully open or fully closed, and stop the drive motor.

2.6 Examples in Exercises

We identify three examples that we will use in the Exercise portions of selected chapters. Each example is similar to an ongoing example, yet different enough to be thought provoking.

2.6.1 The Quadrilateral Program

The Quadrilateral Program is deliberately similar to the Triangle Program. It accepts four integers, a, b, c, and d, as input. These are taken to be sides of a four-sided figure and they must satisfy the following conditions:

c1. $1 \le a \le 200$ (top)
c2. $1 \le b \le 200$ (left side)
c3. $1 \le c \le 200$ (bottom)
c4. $1 \le d \le 200$ (right side)

The output of the program is the type of quadrilateral determined by the four sides (see Figure 2.1): Square, Rectangle, Trapezoid, or General. (Since the problem statement only has information about lengths of the four sides, a square cannot be distinguished from a rhombus, similarly, a parallelogram cannot be distinguished from a rectangle.)

1. A *square* has two pairs of parallel sides (a||c, b||d), and all sides are equal (a = b = c = d).
2. A *kite* has two pairs of equal sides, but no parallel sides (a = d, b = c).
3. A *rhombus* has two pairs of parallel sides (a||c, b||d), and all sides are equal (a = b = c = d).
4. A *trapezoid* has one pair of parallel sides (a||c) and one pair of equal sides (b = d).
5. A *parallelogram* has two pairs of parallel sides (a||c, b||d), and two pairs of equal sides (a = c, b = d).
6. A *rectangle* has two pairs of parallel sides (a||c, b||d) and two pairs of equal sides (a = c, b = d).
7. A *(general) quadrilateral* has four sides, none equal and none parallel (aka a trapezium).

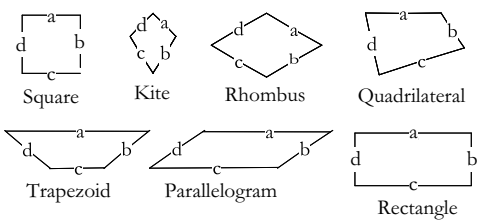

Figure 2.1 The Seven Quadrilaterals.

2.6.2 The NextWeek Function

The complexity in the quadrilateral program is primarily computational. The complexity of the NextWeek function, as with its clone, NextDate, is due to logical relationships among the input variables. NextWeek is a function of three variables: month, date, and year. It returns the date of the day one week 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. $1842 \leq year \leq 2042$

2.6.3 The Windshield Wiper Controller

An automobile windshield wiper 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

Exercises

1. 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 Switch 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 Switch clause for February.
2. 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, 1942? Why?
3. 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 public static int triangle3 to include the right triangle feature.
4. What would the public static int triangle3 do for the sides -3, -3, 5 if it did not contain the input validation code?

```
// Step 1: Validate Input
c1 = (1 <= a) && (a <= 200);
```

```
c2 = (1 <= b) && (b <= 200);
c3 = (1 <= c) && (c <= 200);
```

Discuss this in terms of the considerations we made in Chapter 1.

5. Consider a function YesterDate as the inverse of NextDate. Given a mon th, day, year, YesterDate returns the date of the day before. Develop a program for YesterDate. This is a "symmetric" program to NextDate. For testing purposes, we could implement

   NextDate(YesterDate(mm. dd. yyyy)

   and get the original date as a result.
6. Develop a program for NextWeek.

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.

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. Readers who are already comfortable with the topics in this chapter should skip to the next chapter and start right in on graph theory.

In general, discrete mathematics is more applicable to specification-based 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, and 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}