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: (1) Failures only occur in an executable representation, which is usually taken to be source code, or

more precisely, loaded object code. (2) 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 Michelangelo virus is an example of such a fault — it does not execute until 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 and 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.

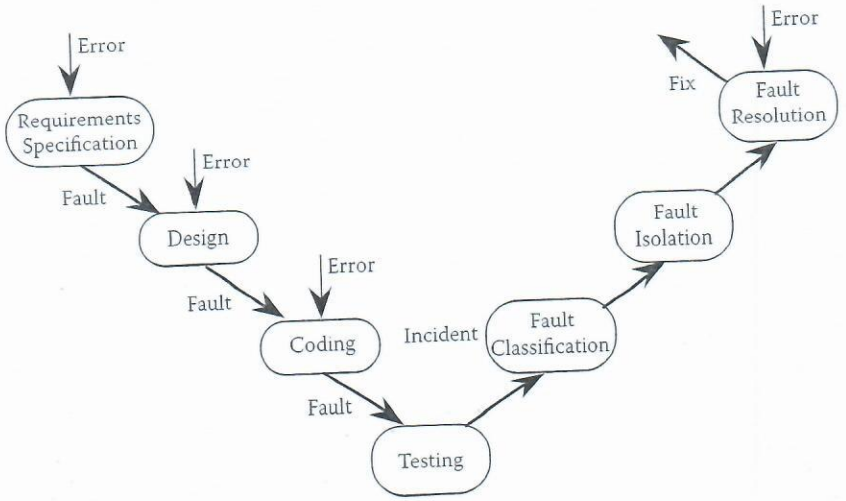

Figure 1.1 A testing life cycle.

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:

> Inputs, 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, comparing these with the expected outputs, and then ensuring that the expected postconditions exist 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.

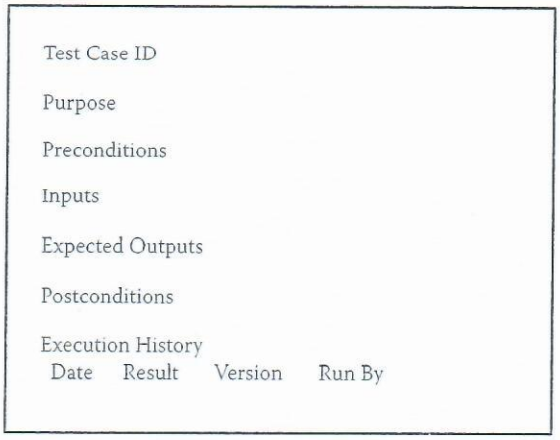

Figure 1.2 Typical test case information.

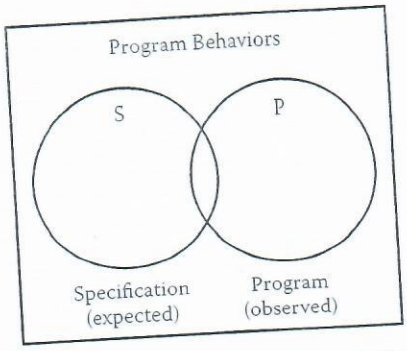

Figure 1.3 Specified and implemented program behaviors.

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 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, some possibilities arise: either such a test case is unwarranted, the specification is deficient, or the tester wishes to determine that specified nonbehavior does not

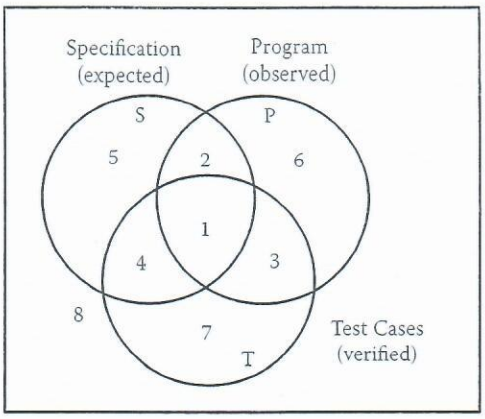

Figure 1.4 Specified, implemented, and tested behaviors.

occur. (In my experience, good testers often postulate test cases of this last 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 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

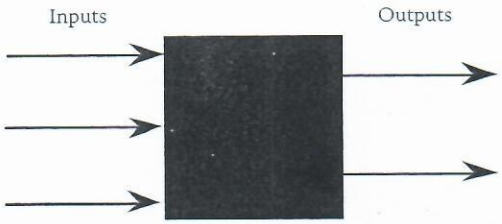

Figure 1.5 An engineer's black box.

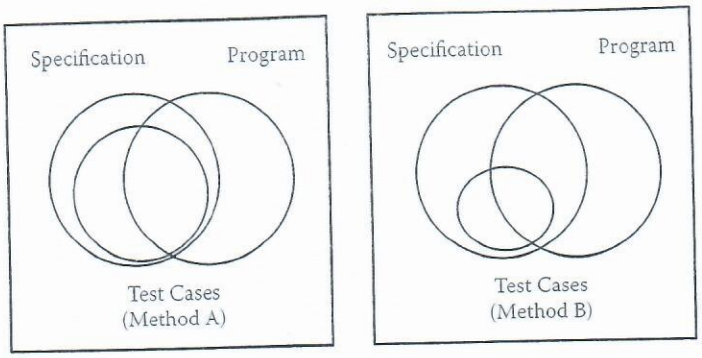

Figure 1.6 Comparing functional test case identification methods.

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 the implementation changes, the test cases are still useful; and (2) test case development can occur in parallel with the implementation, thereby reducing the 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 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.

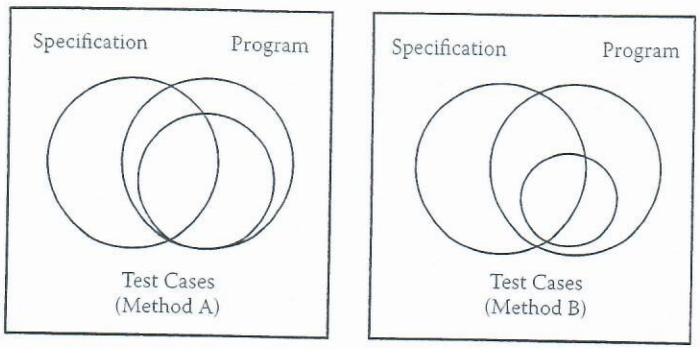

Figure 1.7 Comparing structural test case identification methods.

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. In Chapter 11 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 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 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 functional testing and the measurement of structured testing. Earlier,

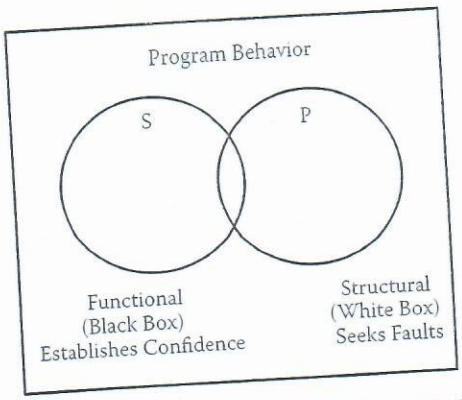

Figure 1.8 Sources of test cases.

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 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 Table 1.1 through Table 1.5; most of these are from the IEEE standard, but I have added some of my favorites.

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. (From Beizer, B., *Software System Testing and Quality Assurance*, Van Nostrand Reinhold, New York, 1984.)

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 testing as a means of identifying distinct levels of testing and for clarifying the objectives that pertain to each level. A

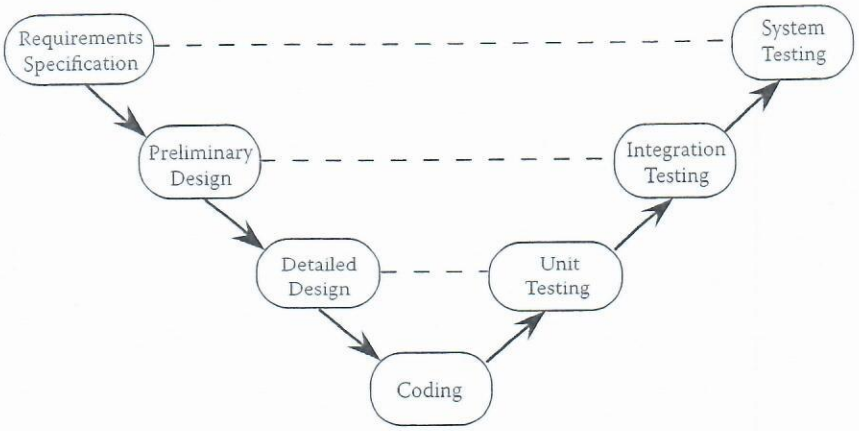

Figure 1.10 Levels of abstraction and testing in the waterfall model.

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 — system, integration, and unit 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, B., *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, E.F., Jr., Automated software testing: a technical perspective, *American Programmer*, Vol. 4, No. 4, April 1991, pp. 38–43.

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

Poston, R.M., *T: Automated Software Testing Workshop*, Programming Environments, Inc., Tinton Falls, NJ, 1990.

Poston, R.M., A complete toolkit for the software tester, *American Programmer*, Vol. 4, No. 4, April 1991, pp. 28–37. Reprinted in CrossTalk, a USAF publication.

vels of abstraction.
the software devel-
sting as a means of
ain to each level. A