

# Designing Programs That Are Less Likely To Hide Faults\*

Jeffrey M. Voas

Reliable Software Technologies Corp.

Keith W. Miller

College of William & Mary

Jeffery E. Payne

Reliable Software Technologies Corp.

**Abstract:** *An important motivation for software testing is to increase confidence that the software no longer contains faults. In this paper we explain a technique for using fewer tests to gain an equivalent confidence in software. Our techniques complement random black box testing. To be able to use fewer tests and gain equivalent confidence, we must either (1) isolate and remove software characteristics that discourage software from revealing faults during testing, or (2) find a method of selecting tests that have a greater ability to reveal the existence of any existing faults. The first of these two alternatives is the subject of this paper. We present a conjecture concerning “testability,” a software characteristic that frequently increases the likelihood that faults are detected during random black-box testing. We propose design measures to increase testability.*

**Index Terms:** Software testing, quality, reliability, correct, software design, testability, domain/range ratio.

## 1 Introduction

Software quality is a popular phrase used with diverse meanings. One interpretation of software quality is *software reliability*, defined as the probability of failure-free execution given a specific environment and a fixed time interval. Another interpretation often attached to software quality is *readability*; for example, structured code, which is easier to read, is frequently referred to as higher quality code than unstructured code (or “spaghetti” code). A broad definition of software quality is a measure of the characteristics that we want in our software. For example, if ease of maintenance is the most important issue for our software, then we would classify software as having high quality if it displays characteristics that facilitate modifications. In this paper we

---

\*Supported by a National Research Council NASA-Langley Resident Research Associateship and NASA-Langley Grant NAG-1-884. Address Correspondence to Jeffrey Voas, Reliable Software Technologies Corporation, Penthouse Suite, 1001 North Highland Street, Arlington, VA 22201.

focus on the *absence of faults* as the determining characteristic of high quality software. But establishing the absolute absence of faults is generally not possible. So we fall back to a less stringent definition of high software quality: we consider software as being of high quality if it is “probably correct.”

In order to determine whether software is probably correct, software engineers use a variety of techniques. We focus on proof of correctness and software testing in this paper because both these techniques can quantify a probability that the software is correct: if a proof of correctness can be relied on, then a program proven correct is correct with probability 1; software testing uses statistical analysis to establish a probability less than 1 that the software is correct.

Proof of correctness has the theoretical potential to establish correctness. However, in practice this goal has proved elusive for most software. For software of even modest complexity, most developers do not complete a formal proof of correctness. Even when a proof has been completed, there is no guarantee that the proof itself is flawless. Software testing has remained the most frequently used validation technique. Software testing cannot show the absolute absence of faults unless it is exhaustive [2], however software testing can assert the probable absence of faults with a particular confidence. In this paper we assume that non-exhaustive testing will be used to establish confidence in a program. We define a program as *probably correct* if we have confidence  $c$  after  $T$  randomly selected tests that the probability that the program will fail is less than a preset threshold  $\theta$ .  $\theta$  can be either (1) an estimate of the minimum probability that the program will fail, (2) a prediction of the minimum probability that the program will fail, or (3) an arbitrary value specified in some *a priori* fashion. Our notion of probably correct is similar to Hamlet’s notion of probable correctness [3]— the difference in his notion and ours will be in the white-box analysis techniques employed to attain  $\theta$ .

To assert that a program is probably correct, we use Hamlet’s probable correctness equation that relates the number of tests, the desired confidence, and the threshold probability of failure:

$$1 - (1 - \theta)^T = c, \tag{1}$$

In Hamlet’s formulation, the method of establishing an acceptable  $\theta$  is left to the user, and the equation defines legal pairs of  $T$  and  $c$  values. As  $\theta$  becomes small, larger and larger numbers of tests ( $T$ ) are needed to gain a given confidence  $c$ . Assessing software that must have very small  $\theta$  values is a daunting task, and many researchers have explored this problem. Several models for finding  $\theta$  are found in [9, 5, 3]. These models can generally be broken into two classes: *black-box* and *white-box*. Black-box analysis models observe the behavior of the program during execution, but they do not observe the program itself; white-box analysis models instead observe the code to find  $\theta$ . All this previous research emphasizes the practical and theoretical problems that arise when seeking to establish high confidence in a small  $\theta$ .

Regardless of how we attain  $\theta$ , equation 1 requires enormously more randomly selected *black-*

*box* tests  $T$  as  $\theta$  diminishes (See Table 1). To be able to assert that software has high quality when faced with this enormous number of required tests, we must find methods that cause random black-box testing to be more “powerful,” meaning that random black-box testing must reveal the existence of faults (if faults were to exist) by causing failures within a reasonable number of tests. Another way of stating this is that methods must allow a smaller number of tests to establish a given  $\theta$ . This paper suggests a radically new approach: instead of trying to prove that software has a very small  $\theta$ , we will attempt to establish, with some level of confidence, two related but distinct conjectures:

1. That if faults exist, the faults will result in a probability of failure of at least  $\theta$ .
2. The software has a failure rate less than  $\theta$ .

If these two conjectures can be demonstrated, the first by white-box analysis and the second by black-box testing, then we gain confidence that the code has no faults. Although indirect, this approach offers hope of establishing higher confidences than are possible using traditional means. The remainder of our paper is organized as follows. Section 2 discusses the relationship between probably correct, reliability, and testing—these three terms are related but distinct. Section 3 describes a quantifiable software characteristic that predicts whether software is likely to reveal faults during random black-box testing; this characteristic is found using white-box analysis techniques and is termed “testability.” We use testability to increase the “target”  $\theta$  necessary to increase our confidence about the absence of faults. Section 4 presents preliminary ideas for designing software with higher testability, and which is therefore less likely to hide faults than ordinary software.

## 2 Relating Probably Correct, Software Reliability, and Testing

For those that are more familiar with software reliability than the probable correctness model, Section 2 describes the relationship between the terms probably correct and reliability, and explains the role of testing in these techniques. *Software reliability* is defined to be the probability of failure-free operation of the software in a fixed environment with a specific input distribution for a fixed period of time. Software reliability can be quantified as a point estimate using the following expression:

$$\frac{\text{number of successful executions}}{\text{total number of executions}}.$$

Software reliability is estimated by repeated executions of the program according to some input distribution, with the assumption that there exists a means for checking the correctness of each output.

The point estimate equation shown above for reliability is only useful when some of the executions are unsuccessful. If all the executions are successful, we would produce a reliability estimate of 1.0 given the above expression; this prediction is not necessarily true, and is too gross a generalization for practical assessment. For programs that have not failed during testing and have not been exhaustively tested, this equation should not be applied; when the event of a software failure is this infrequent, more sensitive statistical techniques are required to quantify the reliability. Generally software with high reliability will have high quality, i.e., high reliability software will usually meet our definition of probably correct software. But we cannot immediately assume this. Examples can be created where programs are probably correct, however they have a lower reliability than expected. For instance, suppose that a program works for all tests in an finite but very large test set except test  $x$ , for which the program fails. In this case, if we were to uniformly select tests over the entire input domain with replacement,  $\theta$  is

$$\lim_{t \rightarrow \infty} \frac{1}{t}.$$

Further assume that  $x$  is selected 50% of the time in the program’s operational environment. Our program fits the criteria of probably correct (if we could perform the intractable amounts of testing needed for a high confidence), however its reliability in this environment is 0.5. Although this example is contrived, it demonstrates that probable correctness and reliability are distinct. In this paper, we generalize and assume that high quality  $\longleftrightarrow$  high reliability. Our goal is to assess software as having high quality. If we use black-box testing alone to assess software, an intractable amount of testing is required to establish a very small probability of failure. To get a feeling for how difficult it is to assess a probability of failure that is less than  $10^{-9}$  (i.e.,  $\theta = 10^{-9}$ ) with 99% confidence, it can be shown that approximately 4.6 billion successful executions (tests according to some input distribution) are needed. This translates into a software reliability estimate of  $1 - 10^{-9}$ . The equation for finding a  $T$  given  $c$  and  $\theta$  is:

$$T = \frac{\ln(1 - c)}{\ln(1 - \theta)}. \tag{2}$$

Even when an automated oracle is available, the practical problems of such testing is clear. Furthermore, if during random black-box testing the software *does* fail, then the software must be fixed and random black-box testing must completely restart (meaning throw out all previous successful executions and start again). This is due in part to statistics that have shown that as much as 30% of all new code contains new faults. We can only attain high confidence in low probability of failure estimates if we somehow increase the power of black-box testing to find faults and increase their power to indicate that faults do not exist. To reduce the number of required tests to a more tractable number, one of two techniques are available:

1. Select tests that have a greater ability to reveal faults; this is explored in [7].

$T$	$\theta$	$c$
7	0.1	0.5
44	0.1	0.99
458	0.01	0.99
298	0.01	0.95
460,514	$10^{-5}$	0.99
46,051,699	$10^{-7}$	0.99
693,147,181	$10^{-9}$	0.5
2,302,585,093	$10^{-9}$	0.9
2,995,732,273	$10^{-9}$	0.95

Table 1: Various  $T$ s,  $\theta$ s, and  $c$ s.

- Design software that has a greater ability to fail when faults *do* exist. This means that we must create programs that (1) are likely to have larger proportions of the code exercised for each input, (2) contain program constructs that are likely to cause the state of the program to become incorrect if the constructs are themselves incorrect, (3) propagate incorrect states into software failure. Models for quantifying the likelihood of these three events occurring are found in [9]. A design scheme that forces these events to occur reduces  $T$  by increasing the  $\theta$  necessary to establish the desired confidence that the software contains no faults. This is the focus of Section 4.

The following analogy illustrates how fewer tests can yield an equivalent confidence when we can be sure the software will not hide faults: imagine that you are writing a program for scanning black and white satellite photos looking for evidence of a large barge. If you are sure that the barge will appear as a black rectangle, and that any barge will cover at least a 10 by 20 pixel area in an image, then the program can use techniques that could not be used if the barge size were not established beforehand. For example, assume that the original image has been subsampled so that each pixel in the new image is the average of a five by five square of pixels in the original image. This subsampled image could be scanned 25 times more quickly than the original; with the barge size guaranteed to be large enough, any barge would still be detectable in the lower resolution image. (The shape of a suspected barge could be determined by more detailed examination of the original image at higher resolution.) But if a barge might exist in the image as a smaller rectangle, then the low resolution image might hide the barge inside of one of its averaged pixels. The lower bound on barge size makes the lower resolution image sufficient for locating barges. There is a direct relationship between the minimum barge size and the amount of speed-up that can be accomplished by subsampling.

Looking for a barge in the image is analagous to looking for faults in a program; instead of examining groups of pixels, we examine the output from a test case execution. The more inputs that cause the fault to produce incorrect output (the bigger the barge), the fewer random tests

will be required to reveal the fault (the coarser the grid necessary to locate the barge). If we can guarantee that any fault in a program will cause the program to fail for a sufficiently large proportion of tests, then we can reduce the number of tests necessary to be confident that no faults exist.

There are many distinct ways of implementing a specified function. Certain implementations of the function may have a greater ability to hide faults than others. Thus  $x$  test cases applied to a program  $Y$  of specification  $Z$  that readily hides faults should not produce the same confidence as the same  $x$  test cases applied to a different implementation of  $Z$  that does not hide faults. Therefore, if we need  $x$  test cases to show that an arbitrary program is not hiding faults with confidence  $c$ , then we need fewer than  $x$  test cases to show that a program is not hiding faults if we know this program does not readily hide faults.

### 3 Software Testability

In our analogy, the size of the barge determined how difficult it was to scan photographs. In the software domain, we need to precisely quantify characteristics of code that affect how difficult it is to test software. In the past few years, a technique for determining whether a particular piece of software is likely to reveal faults during random black-box testing has been developed, called *sensitivity analysis* [9]. This analysis is a predictive technique that makes use of three probability estimates at each location of interest in the software:

1. The probability that the location is executed on inputs selected from the assumed input distribution of the software.
2. The probability that if a mutant exists at this location it will adversely change the data state.
3. The probability that if the data state is adversely changed that the change will propagate to the output.

These three probability estimates are combined to characterize, for each code location and for the software as a whole, a prediction of the tendency to reveal or to hide software faults during random testing. Sensitivity analysis's predictions are based upon repeated executions, syntactic mutations, and data state manipulations to obtain these probabilities. Although the process requires many program executions, it does *not* require an oracle. Sensitivity analysis concentrates on ascertaining the testability of code after it is written. In this paper, we concentrate on estimating and controlling testability *before* it is written, during the design phase. Readers interested in more details about sensitivity analysis can find them in [9].

Software faults that infrequently affect software's output are dangerous. Such faults reduce the quality of our software by decreasing the  $\theta$  we must test for to establish that no faults

exist, thereby increasing the number of tests needed. If we can establish, with a high degree of confidence, that any faults that exist will be “obvious” to testing, then we can establish the absence of faults more quickly. If software has high testability, we are assured of “bigger targets” for our random black-box tests. When a software fault causes frequent software failures, testing is likely to reveal the fault before the software is released; when the fault remains undetected during testing, the hidden fault can cause disaster after the software is installed.

One method for deciding whether a program has this desirable property of “quickly” revealing the existence of faults can be found in [9]. To quantify this characteristic, Voas [9] uses a technique termed *PIE* to quantify the likelihood that a location  $l$  is executed ( $\hat{\epsilon}_l$ ), the likelihood that a mutant  $m$  of a location  $l$  causes the state of the machine to be altered ( $\hat{\lambda}_{ml}$ ), and the likelihood that an altered state  $a$  after location  $l$  will propagate ( $\hat{\psi}_{al}$ ). The product of these three probabilities can be composed into a way of estimating the testability of a location,  $\theta_l = \hat{\epsilon}_l \cdot \hat{\lambda}_{ml} \cdot \hat{\psi}_{al}$ . The location of the program that has the minimum product when this process is applied then is a conservative statement concerning the testability of the program itself.

It is important to understand how using a *prediction*  $p$  of the minimum failure probability, such as testability, differs from using an *estimate*  $e$  of the minimum failure probability, such as  $(1 - \text{reliability})$ . The prediction  $p$  seeks to establish how large the effect of any one fault will be if any faults exist at all.  $e$  seeks to establish the effect of all existing faults. The fundamental difference in the intent of these measures is reflected in the way they are established.  $e$ ,  $(1 - \text{reliability})$ , is found using black-box analysis, whereas testability is found using white-box analysis.  $e$  is dependent on having the correct program or its specification, and is determined from repeated executions of the program;  $p$  does not rely on a specification or oracle, and can be based strictly on code as described in [9]. In general,  $p$  should be a “more conservative”  $\theta$  than  $e$ . This is because the testability predictor  $p$  is a function of a single “potential” fault, whereas  $e$  may be a function of several different “actual” faults. We define a *potential fault* to be a fault that has a likelihood of existing that is greater than some preset probability. Determination of what constitutes a potential fault is subjective. A discussion about establishing reasonable limits on the probability threshold are discussed more fully in [9].

If  $\theta$  were found using a black-box analysis model and found to have a tiny value, then we would expect to also find a tiny testability predictor from a white-box analysis model. If in this case we do not find a tiny testability predictor, then our white-box analysis model of finding testability is too liberal. Regardless of which is used for determining whether the software is probably correct, if we design software in a manner that the software cannot retain undetected faults, either model will result in higher required  $\theta$ s, and this results in fewer required tests.

## 4 Designing for Testability

There are three reasons why a fault can remain hidden in software for a given input:

1. The fault is not executed,
2. The fault is executed, but it does not affect the computational state of the program, i.e., it produces the same effect in the computational state as the corrected code would, or
3. The fault is executed, and it affects the computational state of the program, but this effect does not *propagate* to an output state of the program.

If any of these situations are true for a given test/fault pair, the program will not fail for that pair. Software that frequently suffers from these conditions for tests from a given input distribution is software that has low testability for that distribution. It is our goal to produce programs where the likelihood of any of these three conditions occurring is reduced. If we can produce software that reveals faults (when they exist) with greater probabilities, then we will observe higher failure probabilities (when faults exist) during testing.

Consider that the estimated minimum failure probability of some implementation of function  $X$  is  $\theta_s$  when it is designed in some ad hoc fashion, and let the estimated minimum failure probability be  $\theta_l$  when we design the software to have a higher testability; then  $\theta_s < \theta_l$ . When we use  $\theta_s$  and  $c$  in equation 1, we get some number of tests  $T_s$  that must be successfully executed to gain confidence  $c$  that the actual probability that the program will fail is less than  $\theta_s$ . Likewise, when we use  $\theta_l$  and  $c$  in equation 1, we get some number of tests  $T_l$  that must be successfully executed to gain confidence  $c$  that the actual probability that the program will fail is less than  $\theta_l$ . So we are getting equivalent confidences in two slightly different entities: one is the actual probability that the program will fail is less than  $\theta_l$ , and the other is the actual probability that the program will fail is less than  $\theta_s$ .

If the model we apply for finding an estimated minimum failure probability is *truly* conservative, then there should be no potential fault that can be injected that can cause an actual failure probability that is less than the estimated minimum failure probability. If the white-box analysis estimated minimum failure probabilities  $\theta_l$  and  $\theta_s$  have been found in a manner such that there is no potential fault that can be injected in these two programs whose probability of causing failure is less than  $\theta_l$  and  $\theta_s$  respectively, then we effectively have confidence  $c$  that our programs are probably correct. Thus by designing for testability,  $T_s - T_l$  is our reward in the number of tests saved.

By using the sensitivity analysis described above [9], we can measure the testability of code segments. And with this technique, we can show that different implementations of the same function have different testabilities. We start with a trivial example, replacing the code segment

`a := a div b;`

with

```
a1 := a;  
a := 0;  
while a1 > b do begin  
a1 := a1 - b;  
a := a + 1; end
```

This substitution immediately increases the testability, because we can study the propagation of computational state errors within the loop, whereas we cannot study computational state errors in the “atomic” `div` operator. The way in which we can do so follows: if the value of variable `a` coming into the `div` instruction is incorrect, then there is an increasing greater probability (as the value of `b` increases) that the result of this `div` instruction will still be correct. This wipes out the existence of such a data state error. By breaking up the `div` instruction into repeated subtractions, the value of `a1` is available for validation at the end of the loop. This could be used to signal that `a` was in error. By breaking up the `div` operator or applying some other programming trick (such as not reusing the value of `a`), the incorrectness contained in `a` will be nullified. This example was shown *not* to suggest that `divs` should be implemented in such a laborious (and low “readability”) manner, but instead to highlight how such operators can affect software testability and thus help to hide faults during software testing.

As a non-trivial example, consider a module that takes as input a matrix of weather data from several different locations and at several different times. The module gives a single output, a binary decision on whether to keep an airport open. When this module is tested, almost all of its internal computations are hidden from the tester. It is conceivable that a software fault that would be obvious from looking at some intermediate values (for example, the calculation of the change in temperature over a given time span) would not necessarily be obvious in the single binary output. If the intermediate values of tests could be compared with the expected values, then we would expect that a larger proportion of tests would reveal any given fault. Although this observation may seem pedestrian, it can have a significant impact on how software is designed and tested.

During software design, we can isolate certain subfunctions of the software that will have a greater tendency to hide faults. A simple metric, available from software specifications, indicates software subfunctions that will tend to hide faults. The metric is the DRR, the “domain/range ratio.” The *domain/range ratio* of a function is the ratio of the cardinality of the function’s domain,  $d$ , to the cardinality of the function’s range,  $r$ . This ratio suggests the likelihood that software will more easily hide faults, and measures can be taken during design to increase software testability before the software is written. More specifically, the domain/range ratio “partially suggests” the likelihood that computational state errors will not propagate to the output. This was the third situation listed in Section 3 for why faults can remain hidden. We say “partially suggests” because there are other factors that can diminish the potential for

computational state error propagation. For certain functions, the tests can be found from the outputs by inverting the function. For example, for an infinite domain, the function  $f(x) = 2x$  has only one possible test  $x$  for any output  $f(x)$ . Other functions, for example  $f(x) = \tan(x)$ , can have many different  $x$  values that result in an identical  $f(x)$ ; i.e.,  $\tan^{-1}(x)$  is not a one-to-one function. All inverted functions that do not produce exactly one element of the domain for each element of the range **lose** information that uniquely identifies the test given an output. Restated, many-to-one functions mandate a loss of information; one-to-one functions do not. If we look at two small examples, **div** and **mod**, we see that these functions are not one-to-one functions; however addition and subtraction are one-to-one functions, at least in infinite arithmetic. When **div** or **mod** are executed, information is lost if the statement executing is of the form  $a := f(a)$ . By implementing **div** and **mod** with repeated addition and subtraction, we use statements with lower domain/range ratios. Since we are able to test what occurs internally in these “expanded” **div** and **mod** implementations, we can increase testabilities by exposing these internal states to testing. Similarly, in the airport weather analyzer, making intermediate calculation results available to the tester increases the effective DRR of the module; the software function may still be many-to-one, but on average we have decreased the size of “many.”

Software that implements a many-to-one function has information in its internal computational states that is not communicated in its output. We term this phenomenon *internal state collapse*. When internal state collapse occurs, we run a risk that the lost information may have included evidence that computational states were incorrect. Since such evidence is not visible in the output, the probability of observing a failure during testing is reduced. As the probability of observing a failure decreases, the probability of hidden faults increases. Modules that implement functions where  $d$  is greater than  $r$  must lose information, and this loss suggests a lower module testability. For most module descriptions, their domain and range are  $n$ -dimensional spaces. We admit that finding  $n$ -dimensional spaces for module descriptions is difficult. It is only for module descriptions where we can determine the spaces that designing with the spaces under consideration can occur. For those that we can, we can examine a DRR for a module’s description and know before implementation of the module how much internal state collapse may occur. This **a priori** information can be thought of as a very “rough” approximation of the module’s testability. We say rough because the degree of internal state collapse that occurs only predicts the likelihood of computational state error propagation. It is important to remember that there are other important factors that affect testability.

Preliminary research on the domain/range ratio is described in [8]. Several of the initial insights found from these studies for increasing testabilities include:

1. Limiting the reuse of variables, and attempting to perform analysis on reused variables before they are reused. Had we not used a function of the form  $a := f(a)$ , then **div** and **mod** would not cause information loss, e.g.,  $x := a \text{ div } b$  does not suffer from information loss.

2. Paying particular attention during validation to internal functions whose domain/range ratio is not one-to-one. These functions have a greater potential for hiding computational state errors.
3. Designing modules so that they either have a high domain/range ratio, or a low domain range ratio. Further, if a high domain/range ratio module is created, then the size of the module should be kept small. The intuition behind this is that small modules, even if they have high domain/range ratios, can potentially be verified. Specific heuristics for doing this are discussed in [6].
4. Validating information that is usually hidden during operational use. For instance, object-oriented languages purposely hide information. During validation, we can purposely validate this information through additional output statements, that can later be removed. This can be performed by internal “self-tests” or assertions for intermediate computations.

We acknowledge that several of these suggestions are controversial. We also acknowledge that testability is not the only characteristic of code that is important in the development. However, testability is at least *one* important characteristic during development, and when testability is sacrificed for some other desirable characteristic (such as readability or portability), that tradeoff should be conscious and deliberate. Particularly for software that requires very high reliabilities, some old notions of what is “proper” in software development may have to be adjusted in order to allow the assessment of very high reliabilities. Information hiding, for example, has many advantages during software development; however, if interpreted as hiding information during testing, information hiding is detrimental to testability. Testability (particularly as reflected in the DRR) gives an additional criterion by which to judge whether a module decomposition is appropriate during design.

It may be possible to combine concerns for testability with concerns for other desirable software characteristics. For example, modules could be designed with special read-only testing parameters that are used only during testing (either module test or system test). This restriction could be enforced either by convention or compiler switches. Such a software provision would be analagous to the testing pins used to verify hardware chips.

## 5 Summary

This paper has argued for the lack of faults as a measure of software quality. We have shown how to apply Hamlet’s probable correctness model to assess a confidence that the true failure probability of the program is less than some preset threshold  $\theta$ . If  $\theta \approx 0$ , we essentially have the confidence that there are *no* faults in the code. However if  $\theta$  is approximately zero, the number tests required is overwhelming. Therefore we opted to use the white-box analysis notion of

testability to determine a *larger*  $\theta$  that defines the number of tests required to establish confidence that the software is free of faults. We can only justify this larger  $\theta$  by establishing that faults, *if they exist*, will exhibit a probability of failure at least as large as the  $\theta$  we select. We then combine testability and probable correctness to assess high software quality. If the ability of software to hide faults is reduced through specific design measures, we can further reduce the costs of assessing quality.

Assessing and actually achieving high quality (meaning virtually defect-free) is an increasingly important goal as software becomes essential in many life-critical situations. Random black-box testing is the principle technique applied in order to establish a quantifiable confidence that software has no faults. As we have seen, when the desired reliabilities are very high (very tiny probabilities of failure), we need a way of reducing the number of tests required. This makes “designing-for-testability” an important software issue. Integrated circuit designers have “designed for testability” for years [1, 4]; we contend that software designers should too. We conjecture that only through changes in software design practices in the “front-end” of the software life-cycle can high quality in the “tail-end” of this cycle be achieved. The current practice of testing and “tweaking” the code at the end of the software life-cycle in order to slowly inch towards high quality appears futile. Design techniques that are based on testability predictions are a new direction that appears more practical for assessing and achieving the high quality software that many of today’s applications require.

## References

- [1] NEIL C. BERGLUND. Level-Sensitive Scan Design Tests Chips, Boards, System. *Electronics*, March 15 1979.
- [2] O. J. DAHL, E. W. DIJKSTRA, AND C. A. R. HOARE. *Structured Programming*. Academic Press, 1972.
- [3] RICHARD G. HAMLET. Probable Correctness Theory. *Information Processing Letters*, pages 17–25, April 1987.
- [4] MICHAEL C. MARKOWITZ. High-Density ICs Need Design-For-Test Methods. *EDN*, 33(24), November 24 1988.
- [5] K. MILLER, L. MORELL, R. NOONAN, S. PARK, D. NICOL, B. MURRILL, AND J. VOAS. Estimating the Probability of Failure When Testing Reveals No Failures. *IEEE Trans. on Software Engineering*, 18(1):33–44, January 1992.
- [6] J. VOAS AND K. MILLER. A Design Phase Semantic Metric for Software Testability. In *Proc. of the 4th Oregon Workshop on Software Metrics*, Silver Falls, OR, March 22–24 1992. Oregon Center for Advanced Technology Education.

- [7] J. VOAS AND K. MILLER. Applying A Dynamic Testability Technique To Debugging Certain Classes of Software Faults, *The Software Quality Journal*, To appear.
- [8] J. VOAS. Factors That Affect Program Testabilities. In *Proc. of the 9th Pacific Northwest Software Quality Conf.*, pages 235–247, Portland, OR, October 1991. Pacific Northwest Software Quality Conference, Inc., Beaverton, OR.
- [9] J. VOAS. *PIE*: A Dynamic Failure-Based Technique. *IEEE Trans. on Software Engineering*, 18(8), August 1992.