SOFTWARE NEGLIGENCE AND TESTING COVERAGE[1]

Cem Kaner, kaner@kaner.com

Several months ago, a respected member of the software quality community posed the following argument to me:

> A program fails in the field, and someone dies. This leads to a trial. When the QA manager takes the stand, the plaintiff's lawyer brings out three facts:
>
> 1. The failure occurred when the program reached a specific line of code.
> 2. That line had never been tested, and that's why the bug had not been found before the product was released.
> 3. A ***coverage monitor*** was available. This is a tool that allows the tester to determine which lines of code have not yet been tested. Had the QA manager used the tool, and tested to a level of complete coverage (all lines tested), this bug would have been found, and the victim would be alive today.
>
> Therefore, the lawyer has proved that the company was negligent and the victim's family will win the lawsuit.

The question is, what's wrong with this argument? Anything? After all, the company had a well-understood tool readily available and if they had only used it, someone would not have died. *How could the company **not** be liable for negligence?*

OVERVIEW

This presentation explores the legal concept of *negligence* and the technical concept of *coverage*. The article advances several related propositions:

Coverage

1. The phrase, *complete coverage*, is misleading. This "completeness" is measured only relative to a specific population of possible test cases, such as lines of code, branches, n-length sub-paths, predicates, etc. Even if you achieve complete coverage for a given population of tests (such as, all lines of code tested), you have not done complete, or even adequate, testing.
2. We can and should expand the list of populations of possible test cases. We can measure coverage against each of these populations. The decision as to whether to try for 1%, 10%, 50% or 100% coverage against any given population is non-obvious. It involves tradeoffs based on thoughtful judgment.

[1] Portions of this paper were originally published in C. Kaner, "Software Negligence & Testing Coverage", *Software QA Quarterly,* Vol. 2, #2, p. 18, 1995.

Negligence

3. Negligence liability attaches when injury or loss is caused by a failure to satisfy a duty that was imposed by law, as a matter of public policy. So the question of whether or not it is *negligent* to fail to test every line of code turns into a question of whether the company had a public duty to test every line of code.
4. The nature of the duty involved depends on the role of the company in the development and release of the program.
   - A company that publishes a software product has a duty to take reasonable measures to ensure that its products are safe.
   - A company that sells software development services *might* have a duty to its client to deliver a program that is reasonably competently coded and tested.
   - A company that acts as an independent test laboratory *might* owe a duty of competent testing and reporting to the client or to the public.
5. In any of these three cases, it is not obvious whether failure to achieve 100% line coverage is negligent. The plaintiff will have to *prove* that the tradeoff made by the software company was unreasonable.

WHAT IS NEGLIGENCE?

I'll start by considering the situation of the software developer/publisher. This provides the room to explore the coverage issues that are the focus of this paper. The situations of the service providers are of independent interest to the testing community and so will also be considered below.

Under negligence law, software development companies must not release products that pose an *unreasonable* risk of personal injury or property damage.[2] An injured customer can sue your company for negligence if your company did not take *reasonable measures* to ensure that the product was safe.

Reasonable measures are those measures that a reasonable, cautious company would take to protect the safety of its customers. How do we determine whether a company has taken reasonable measures? One traditional approach in law involves a simple cost-benefit analysis. This was expressed as a formula by Judge Learned Hand in the classic case of *United States v. Carroll Towing Co.*:[3]

> Let **B** be the burden (expense) of preventing a potential accident.
>
> Let **L** be the severity of the loss if the accident occurs.[4]
>
> Let **P** be the probability of the accident.
>
> Then *failure to attempt to prevent a potential accident is unreasonable if*
>
> **B < P x L.**

For example, suppose that a software error will cause a total of $1,000,000 in damage to your customers. If you could prevent this by spending less than $1,000,000, but don't, you are negligent. If prevention would cost more than $1,000,000, and you don't spend the money, you are not negligent.

In retrospect, after an accident has occurred, now that we know that there is an error and what it is, it will almost always look cheaper to have fixed the bug and prevented the accident. But if the company didn't know about this bug

---

[2] Note the restriction on negligence suits. Most lawsuits over defective software are for breach of contract or fraud, partially because they don't involve personal injury or property damage.

[3] *Federal Reporter, Second Series,* volume 159, page 169 (United States Court of Appeals, 2nd Circuit, 1947); for a more recent discussion see W. Landes and R. Posner, *The Economic Structure of Tort Law*, Harvard University Press, 1987.

[4] See C. Kaner "Quality Cost Analysis: Benefits and Risks" *Software QA*, Vol. 3, No. 1, p. 23, 1996. The amount, **P x L**, is an estimate of the External Failure Costs associated with this potential accident. Note that there are two different estimates of an external failure cost. You can estimate the amount it will cost your company if the failure occurs, or you can estimate how much it will cost your customers. Most cost-of-quality analyses will look at your company's costs. In negligence cases, judges and juries look at customers' losses.

when it released the program, our calculations should include the cost of finding the bug. *What would it have cost to make the testing process thorough enough that you would have found this bug during testing?*

For example, if a bug in line 7000 crashes the program, **B** would not be the cost of adding *one* test case that miraculously checks this line (plus the cost of fixing the line). **B** would be

- the cost of strengthening the testing so that line 7000's bug is found *in the normal course of testing*, or
- the cost of changing the design and programming practices in a way that would have prevented this bug (*and others like it*) in the first place.

Coming back to the coverage question, it seems clear that you can prevent the crash-on-line-7000 bug by making sure that you at least execute every line in the program. This is *line coverage*.

Line coverage measures the number / percentage of lines of code that have been executed. But some lines contain *branches*—the line tests a variable and does different things depending on the variable's value. To achieve complete *branch coverage*, you check each line, and each branch on multi-branch lines. To achieve complete *path coverage*, you must test every path through the program, an impossible task.[5]

The argument made at the start of this article would have us estimate **B** as the cost of achieving complete line coverage. *Is that the right estimate of what it would cost a reasonable software company to find this bug? I don't think so.*

Line coverage is just one narrow type of coverage of the program. Yes, complete line coverage would catch a syntax error on line 7000 that crashes the program, but what about all the other bugs that wouldn't show up under this simple testing? Suppose that it would cost an extra $50,000 to achieve complete line coverage. If you had an extra $50,000 to spend on testing, is line coverage what you would spend it on? *Probably not.*

Most traditional coverage measures look at the simplest building blocks of the program (lines of code) and the flow of control from one line to the next. These are easy and obvious measures to create, but they can miss important bugs.

A great risk of a measurement strategy is that it is too tempting to pick a few convenient measures and then ignore anything else that is more subtle or harder to measure. When people talk of *complete coverage* or *100% coverage*, they are using terribly misleading language. Many bugs will not be detected even if there is complete line coverage, complete branch coverage, or even if there were complete path coverage.

If you spend all of your extra money trying to achieve complete line coverage, you are spending none of your extra money looking for the many bugs that won't show up in the simple tests that can let you achieve line coverage quickly. Here are some examples:

- A key characteristic of object-oriented programming is that each object can deal with any type of data (integer, real, string, etc.) that you pass to it. Suppose that you pass data to an object that it wasn't designed to accept. The program might crash or corrupt memory when it tries to deal with it.. Note that you won't run into this problem by checking every line of code, because the failure is that the program doesn't expect this situation, therefore it supplies no relevant lines for you to test.

  There is an identifiable population of tests that can reveal this type of problem. If you pass every type of data to every object in your product, you will find every error that involves an object that doesn't properly handle a type of data that is passed to it. You can count the number of possible tests involved here, and you can track the number you've actually run. Therefore, we can make a coverage measure here.

- A Windows program might fail when printing.[6] You achieve complete coverage of printer compatibility tests (across printers) if you use the set of all Windows-supported printers, using all Windows printer drivers

---

[5] G. Myers, *The Art of Software Testing*, Wiley, 1979.

[6] I've been asked by software quality workers who are not familiar with mass-market issues why anyone would want to spend much effort on printer testing. Here is some relevant information. The technical support costs associated with printer incompatibilities are significant for several mass-market software companies. For example, at a presentation on "Wizards" at the *OpCon West 96 Customer Service & Support Conference*, March 18, 1996, Keith Sturdivant reported that print/merge calls on Microsoft Word had been averaging over 50 support-minutes per caller until Microsoft developed a special print troubleshooting Wizard. In my experience at other companies, this is not a surprisingly high number. In "Benchmark Report: Technical Support Cost Ratios," *Soft•letter*, Vol. 10, #10, p. 1, September 21, 1993, Jeffrey Tarter reported an average tech support call cost of $3 per minute. At this rate, the 50

available for each of these printers. These drivers are part of the operating system, not part of your program, but your program can fail or cause a system failure when working with them. The critical test case is not whether a particular line of code is tested, but whether it is tested in conjunction with a specific driver.

- Suppose that you test a desktop publishing program. One effective way to find bugs and usability failures is to use the program to create interesting documents. This approach is particularly effective if you use a stack of existing documents and try to make exact copies of them with your program. To create your stack, perhaps you'll use all the sample files and examples that come with PageMaker, Quark, FrameMaker, and one or two other desktop publishers. In this case, you achieve complete coverage if you recreate all of the samples provided by all available desktop publishers.

The Appendix to this article lists 101 measures of testing coverage. Line coverage is just one of many. There are too many possible tests for you to achieve complete coverage for every type of coverage in the list.

I hope that the list helps you make priority decisions consciously and communicate them explicitly. The tradeoffs will differ across applications—in one case you might set an objective of 85% for line coverage,[7] 100% for data coverage, but only 5% for printer / driver compatibility coverage. For a different program whose primary benefit is beautiful output, you would assign printer coverage a much higher weight.

If you had an extra $50,000 to spend, would you focus your efforts on increasing line coverage or increasing some of the others? Surely, the answer should depend on the nature of your application, the types of risks involved in your application, and the probable effectiveness of the different types of tests. The most desirable strategy will be the one that is most likely to find the most bugs, or to find the most serious bugs.

The legal (negligence) test for the coverage tradeoffs that you make is *reasonability*. No matter what tradeoffs you make, and no matter how much money you spend on testing, you will miss some bugs.[8] Whether or not those bugs are products of negligence in the testing process depends on your reasonability, not on your luck in selecting just the right tests.

Your task is to prioritize among tests in the way that a reasonably careful company would—and to me that means to select the test strategy that you rationally believe is the most likely to find the most bugs or the most serious bugs.

There is no magic talisman in coverage that you can use blindly and be free of negligence liability. Being reasonable in your efforts to safeguard your customer requires careful thought and analysis. Achieving complete (line, branch, whatever) coverage will not insulate you. The plaintiff's attorney will just ask you why you spent all that money on line coverage, at the expense of, say, interrupt coverage. Try to assign your weights sensibly, in a way that you can explain and justify.

The same reasoning applies to customer satisfaction in general. If your approach will control the risks, you've done your job. But if you can identify gaps that leave an unreasonable degree of risk to customer safety or satisfaction, there is no reasonable alternative to addressing those risks.

As a final note, I hope that you'll take a moment to appreciate the richness, multidimensionality, and complexity of what we do as testers. Sometimes we hear that only programmers should be testers, or that all testing should be driven from a knowledge of the workings of the code. This list highlights the degree to which that view is mistaken. Programming skills and code knowledge are essential for glass box testing tasks, but as we explore the full range of black box testing approaches, we find that we also need skills and knowledge in:

- the application itself (subject matter experts)
- safety and hazard analysis
- usability, task analysis, human error (human factors analysis)
- hardware (modems, printers, etc.)

---

minute printer call cost is $150 – for a product that is bundled free with a computer or sold at retail for as little as $99. The cost and aggravation to the customers is also very high.

[7] It can be *very* expensive and difficult to achieve 100% line or branch coverage. Grady reports that values of 80-85% are reasonably achievable. R.B. Grady, *Practical Software Metrics for Project Management and Process Improvement*, PTR Prentice Hall (Hewlett-Packard Professional Books), 1992, p. 171.

[8] See Chapter 2 of C. Kaner, J. Falk, and H.Q. Nguyen, *Testing Computer Software* (2nd. Ed.), Van Nostrand Reinhold, 1993.

- customer relations.

A person who has these skills but who can't program may be an invaluable member of a black box testing team.

SOFTWARE MALPRACTICE

Malpractice? Can you be sued for malpractice? I've heard alarmist talk about software malpractice – the main point of this section is to say, "Calm down."

A person commits malpractice if she provides professional services that don't meet the level that would be provided by a reasonable member of the profession in this community. For example, if you hire a lawyer to draft a contract, and the lawyer makes mistakes that any reasonable lawyer would never make, then you can sue the lawyer for malpractice.

Many people in the software community sell services rather than products. We sell our programming or testing skills on a contract basis, doing work that is defined by our customer. What if you do a terrible job? In any contract for services, the service provider is under a duty to perform the services in a reasonable and workmanlike manner.[9] If you hand back badly written code that has never been tested, you might face claims for breach of contract, misrepresentation, negligence, or malpractice:

- You're probably liable for breach of contract if you promised working code and you delivered garbage.
- You're probably liable for misrepresentation if you said that you knew how to write this type of application, but you've never done one before.
- If the only harm caused by your product is economic (it cost your customer money), then there's an excellent chance that the courts will refuse to allow a negligence-based suit to go forward. If you did a lousy job of programming and testing, your customer should sue you for breach of contract.[10]

If the suit does go forward as a negligence suit, then your customer may have to prove that no reasonable person would have done as bad a job as you. This population of "reasonable persons who sell software services" includes hobbyists, recent university graduates, high school students, and a variety of other odd characters. Proving negligence can be a big challenge.

- The malpractice case is much clearer and much more dangerous. The care you took in providing your services is compared to the level of care that is to be expected from a professional software service provider. If your work doesn't live up to that standard, and your customer loses money as a result, then it doesn't matter whether a hobbyist would have done the job more sloppily than you. You acted as a professional. You provided sub-professional services. You lose.

There's just a small problem in this malpractice case – what professional standards should your work be compared to?

It's relatively easy to compare lawyers' work to professional standards because there *are* professional standards. I belong to the California Bar. Can't be a lawyer in California without being a member of the State Bar. To be admitted to the Bar, I had to take a three-day exam on basic legal knowledge that every practicing lawyer is expected to have. And I had to take another tough exam on professional responsibility – my professional and ethical obligations to my clients and to the public. These exams illustrate a standard of knowledge that is accepted by the entire legal community. As an attorney, I am required by my profession's Code of Professional Responsibility to competently represent my clients. If I fail to follow this Code, I can be hauled before a State Bar Court, prosecuted for violating the profession's code of ethics, and severely punished.

We have nothing comparable in the software community. Some groups have attempted to certify programmers but none has won general (let alone universal) acceptance. Similarly, there are no widely accepted standards for software testers.[11] There is also no enforceable code of ethics.

---

[9] This is clearly discussed in R.T. Nimmer *The Law of Computer Technology: Rights, Licenses, Liabilities* (2nd Ed.) Warren, Gorham & Lamont, 1992 (supplemented 1994). See Chapter 9 generally, especially section 9.16.

[10] This is a complex issue for lawyers, that probably shouldn't be a complex issue. The new Article 2B of the Uniform Commercial Code will clean up most or all of it by making this a clearly defined contract issue. For more on the drafting of Article 2B, see C. Kaner, "Uniform Commercial Code Article 2B: A New Law of Software Quality," in press, *Software QA*, Vol. 3, No. 2, 1996. To read the latest draft of Article 2B, check the *Uniform Commercial Code Article 2B Revision Home Page*, at **http://www.law.uh.edu/ucc2b.**

Someone who advertises their services in terms of an industry certification is inviting an evaluation against a higher standard. My letterhead advertises that I am an ASQC-Certified Quality Engineer (ASQC-CQE). This implicitly invites any (technical, rather than legal) client of mine to judge me against a standard of knowledge that one would expect from an ASQC-CQE, which is a more precisely defined standard than that of a "reasonable person who does quality-related work." It therefore makes a claim of malpractice easier to argue (against me) in my case than in the general case. You can choose to expose yourself to that risk, or you can choose not to.

NEGLIGENT CERTIFICATION

An independent software testing company can get itself into all sorts of interesting trouble. Under the right circumstances, the lab can get itself into the same trouble as any other software service provider (see the discussions of

---

[11] The American Society for Quality Control recently developed a Certification for Software Quality Engineers (CSQE). I contributed to this effort by working in the meeting that developed the final "body of knowledge" to be used in the certification exam. As an ASQC-Certified Quality Engineer, I generally support the ASQC's certification efforts, and I think that there is genuine personal value to be gained from the study and effort required for a CSQE.

I don't want the following comments to be taken as an attack on the CSQE designation or process. They are not an attack. They are a statement of limitations, made by someone who was part of the process, and I am making them to help avoid the confusion that could result in someone trying to identify the CSQE Body of Knowledge as a legally enforceable community standard.

The CSQE Body of Knowledge questionnaire was given to a select group of people in the software community. It went to some ASQC members (I never received one, even though I am moderately active in ASQC, a member of ASQC's software division, and reasonably active in the software testing community. Nor did Hung Nguyen, even though he co-authored *Testing Computer Software*, is a Senior Member of the ASQC, and is very active in the San Francisco ASQC chapter.) My understanding is that it was sent to a random sample of ASQC members. The questionnaire also went to several SPIN members and to attendees at one or two testing-related conferences. It missed several societies. For example, it never went to the Association for Computing Machinery membership, even though this is the largest computer-related professional society in the US. It never went to the Human Factors & Ergonomics Society, even though that organization includes many members who are directly involved in software quality (designing, testing and evaluating software user interfaces for safety, usability, etc.) I know several software testers who are not active in ASQC.

Survey results were contained in the working document, *The Profession of Software Quality Engineering: Results from a Survey of Tasks and Knowledge Areas for the Software Quality Engineer – A Job Analysis Conducted on Behalf of the American Society for Quality Control* by Scott Wesley, Ph.D. and Michael Rosenfeld, Ph.D. of Educational Testing Services. This document notes than only 18.1% of the recipients of the survey responded, compared to 36%, 38%, and 48% response rates for surveys for other ASQC designations.

The survey published several types of demographic information about respondents. It didn't analyze the responses in terms of software market segment, but my sense from looking at the other data was that relatively few came from the mass-market software industry. It also appeared that relatively few people from the mass-market software industry contributed to the development of the questionnaire. I have limited expertise in some other areas, so I am less confident in identifying other software markets as under-represented.

My sense is that software development and testing are not homogenous. I believe that we approach problems differently for mass-market consumer software than for life-critical diagnostic systems or for life-critical embedded systems. The differences are not merely in thoroughness or degree to which they are systematic. The approaches are qualitatively different. For example, some groups in Microsoft have what looks like a mature development and testing process, but it is not an SEI-mature process. The ASQC-Certified Quality Engineer approach recognizes the diversity of the Quality Control community across industries. The ASQC-Certified Software Quality Engineer approach does not appear to recognize this diversity. The ASQC-CSQE Body of Knowledge appears to treat the field as homogenous.

The ASQC-CQE Body of Knowledge requires background knowledge in the history of the quality control movement. The ASQC-CSQE Body of Knowledge appears much less scholarly to me. Couple this with a lack of university training available for software quality, and I question the degree to which we can expect as a matter of law (for malpractice purposes) that the typical software quality worker would have a thoughtful, historical insight into the techniques and approaches that she uses.

In sum, I believe that the CSQE Body of Knowledge represents an interesting body of knowledge to study. I believe that it would be good for an experienced member of the community to know the material covered by the CSQE. And I believe that a CSQE designation tells me a fair bit about the level of knowledge and commitment of a job candidate or coworker. But I do not believe that it is based on a representative study of the software quality community's practices and I do not believe that it can or should serve as a standard that is useful as evidence of community norms.

As noted in the main body, however, if someone says they are a CSQE in their advertising, they are representing themselves as knowledgeable of the CSQE Body of Knowledge and this invites comparison of *that person's* efforts to those one would expect from a CSQE.

product development negligence and malpractice above). In addition, let me mention two other well known court cases:

- *Hanberry v. Hearst Corporation*:[12] A consumer sued *Good Housekeeping* magazine for negligent endorsement of a defectively designed shoe.
- *Hempstead v. General Fire Extinguisher Corporation*:[13] A worker injured by the explosion of a fire extinguisher sued Underwriters' Laboratories for negligence in inspecting, testing, and approving the design of the extinguisher.

In both cases, the public was told that this was a product that had been evaluated and approved. The organization that had allegedly evaluated and approved the product was sued.

If the public is told that, *because it was you who tested the product*, the public should believe that a product that you tested is reliable and safe, then that product had better be reliable and safe. Otherwise, cranky and injured customers will come to your door too, saying that they bought the product because they trusted your recommendation.

APPENDIX: THE MANY TYPES OF TESTING COVERAGE

This appendix lists 101 coverage measures. *Coverage* measures the amount of testing done of a certain type. Because testing is done to find bugs, coverage is also a measure of your effort to detect a certain class of potential errors. For example, 100% line coverage doesn't just mean that you've executed every line of code; it also means that you've tested for every bug that can be revealed by simple execution of a line of code.

Please note that this list is far from complete. For example, it doesn't adequately cover safety issues.[14] Nor does it convey the richness of the tests and test strategies that you can derive from customer complaints and surveys and from tests involving controlled customer observation. And you will add measures as you analyze the application that you're testing.

1. ***Line coverage.*** Test every line of code (Or ***Statement coverage:*** test every statement).
2. ***Branch coverage.*** Test every line, and every branch on multi-branch lines.
3. ***N-length sub-path coverage.*** Test every sub-path through the program of length N. For example, in a 10,000 line program, test every possible 10-line sequence of execution.
4. ***Path coverage.*** Test every path through the program, from entry to exit. The number of paths is impossibly large to test.[15]
5. ***Multicondition or predicate coverage.***[16] Force every logical operand to take every possible value. Two different conditions within the same test may result in the same branch, and so branch coverage would only require the testing of one of them.
6. ***Trigger every assertion check in the program.*** Use impossible data if necessary.
7. ***Loop coverage.*** "Detect bugs that exhibit themselves only when a loop is executed more than once."[17]
8. ***Every module, object, component, tool, subsystem, etc.*** This seems obvious until you realize that many programs rely on off-the-shelf components. The programming staff doesn't have the source code to these components, so measuring line coverage is impossible. At a minimum (which is what is measured here), you need a list of all these components and test cases that exercise each one at least once.

---

[12] California Appellate Reports, 2nd Series, Vol. 276, p. 680 (California Court of Appeal, 1969).

[13] Federal Supplement, Vol. 269, p. 109 (United States District Court for the District of Delaware, applying Virginia law, 1969).

[14] N. Leveson, *Safeware: System Safety and Computers*, Addison-Wesley, 1995.

[15] See G. Myers, *The Art of Software Testing*, Wiley, 1979, and Chapter 2 of C. Kaner, J. Falk, and H.Q. Nguyen, *Testing Computer Software* (2nd. Ed.), Van Nostrand Reinhold, 1993.

[16] G. Myers, *The Art of Software Testing*, Wiley, 1979 (multicondition coverage) and B. Beizer, *Software Testing Techniques* (2nd Ed.), Van Nostrand Reinhold, 1990.

[17] B. Marick, *The Craft of Software Testing*, Prentice Hall, 1995, p. 146.

9. ***Fuzzy decision coverage.*** If the program makes heuristically-based or similarity-based decisions, and uses comparison rules or data sets that evolve over time, check every rule several times over the course of training.

10. ***Relational coverage.*** "Checks whether the subsystem has been exercised in a way that tends to detect off-by-one errors" such as errors caused by using < instead of <=.[18] This coverage includes:

    - ***Every boundary on every input variable.***[19]
    - ***Every boundary on every output variable.***
    - ***Every boundary on every variable used in intermediate calculations.***

11. ***Data coverage.*** At least one test case for each data item / variable / field in the program.

12. ***Constraints among variables:*** Let **X** and **Y** be two variables in the program. **X** and **Y** constrain each other if the value of one restricts the values the other can take. For example, if **X** is a transaction date and **Y** is the transaction's confirmation date, **Y** can't occur before **X**.

13. ***Each appearance of a variable.*** Suppose that you can enter a value for **X** on three different data entry screens, the value of **X** is displayed on another two screens, and it is printed in five reports. Change **X** at each data entry screen and check the effect everywhere else **X** appears.

14. ***Every type of data sent to every object.*** A key characteristic of object-oriented programming is that each object can handle any type of data (integer, real, string, etc.) that you pass to it. So, pass every conceivable type of data to every object.

15. ***Handling of every potential data conflict.*** For example, in an appointment calendaring program, what happens if the user tries to schedule two appointments at the same date and time?

16. ***Handling of every error state.*** Put the program into the error state, check for effects on the stack, available memory, handling of keyboard input. Failure to handle user errors well is an important problem, partially because about 90% of industrial accidents are blamed on human error or risk-taking.[20] Under the legal doctrine of *foreseeable misuse*,[21] the manufacturer is liable in negligence if it fails to protect the customer from the consequences of a reasonably foreseeable misuse of the product.

17. ***Every complexity / maintainability metric against every module, object, subsystem, etc.*** There are many such measures. Jones[22] lists 20 of them.[23] People sometimes ask whether any of these statistics are grounded in a theory of measurement or have practical value.[24] But it is clear that, in practice, some organizations find them an effective tool for highlighting code that needs further investigation and might need redesign.[25]

18. ***Conformity of every module, subsystem, etc. against every corporate coding standard.*** Several companies believe that it is useful to measure characteristics of the code, such as total lines per module, ratio of lines of comments to lines of code, frequency of occurrence of certain types of statements, etc. A module that doesn't fall

---

[18] B. Marick, *The Craft of Software Testing*, Prentice Hall, 1995, p. 147.

[19] Boundaries are classically described in numeric terms, but any change-point in a program can be a boundary. If the program works one way on one side of the change-point and differently on the other side, what does it matter whether the change-point is a number, a state variable, an amount of disk space or available memory, or a change in a document from one typeface to another, etc.? See C. Kaner, J. Falk, and H.Q. Nguyen, *Testing Computer Software* (2nd. Ed.), Van Nostrand Reinhold, 1993, p. 399-401.

[20] B.S. Dhillon, *Human Reliability With Human Factors*, Pergamon Press, 1986, p. 153.

[21] This doctrine is cleanly explained in S. Brown (Ed.) *The Product Liability Handbook: Prevention, Risk, Consequence, and Forensics of Product Failure*, Van Nostrand Reinhold, 1991, pp. 18-19.

[22] C. Jones, *Applied Software Measurement*, McGraw-Hill, 1991, p. 238-341.

[23] B. Beizer, *Software Testing Techniques* (2nd Ed.), Van Nostrand Reinhold, 1990, provides a sympathetic introduction to these measures. R.L. Glass, *Building Quality Software*, Prentice Hall, 1992, and R.B. Grady, D.L. Caswell, *Software Metrics: Establishing a Company-Wide Program*, Prentice Hall, 1987, provide valuable perspective.

[24] For example, C. Kaner, J. Falk, and H.Q. Nguyen, *Testing Computer Software* (2nd Ed.), Van Nostrand Reinhold, 1993, pp. 47-48; also R.L. Glass, *Building Quality Software*, Prentice Hall, 1992, "Software metrics to date have not produced any software quality results which are useful in practice" p. 303.

[25] R.B. Grady, D.L. Caswell, *Software Metrics: Establishing a Company-Wide Program*, Prentice Hall, 1987 and R.B. Grady, *Practical Software Metrics for Project Management and Process Improvement*, PTR Prentice Hall (Hewlett-Packard Professional Books), 1992, p. 87-90.

within the "normal" range might be summarily rejected (bad idea) or re-examined to see if there's a better way to design this part of the program.

19. ***Table-driven code.*** The table is a list of addresses or pointers or names of modules. In a traditional CASE statement, the program branches to one of several places depending on the value of an expression. In the table-driven equivalent, the program would branch to the place specified in, say, location 23 of the table. The table is probably in a separate data file that can vary from day to day or from installation to installation. By modifying the table, you can radically change the control flow of the program without recompiling or relinking the code. Some programs drive a great deal of their control flow this way, using several tables. Coverage measures? Some examples:
    - check that every expression selects the correct table element
    - check that the program correctly jumps or calls through every table element
    - check that every address or pointer that is available to be loaded into these tables is valid (no jumps to impossible places in memory, or to a routine whose starting address has changed)
    - check the validity of every table that is loaded at any customer site.
20. ***Every interrupt.*** An interrupt is a special signal that causes the computer to stop the program in progress and branch to an interrupt handling routine. Later, the program restarts from where it was interrupted. Interrupts might be triggered by hardware events (I/O or signals from the clock that a specified interval has elapsed) or software (such as error traps). Generate every type of interrupt in every way possible to trigger that interrupt.
21. ***Every interrupt at every task, module, object, or even every line.*** The interrupt handling routine might change state variables, load data, use or shut down a peripheral device, or affect memory in ways that could be visible to the rest of the program. The interrupt can happen at any time—between any two lines, or when any module is being executed. The program may fail if the interrupt is handled at a specific time. (Example: what if the program branches to handle an interrupt while it's in the middle of writing to the disk drive?)

The number of test cases here is huge, but that doesn't mean you don't have to think about this type of testing. This is path testing through the eyes of the processor (which asks, "What instruction do I execute next?" and doesn't care whether the instruction comes from the mainline code or from an interrupt handler) rather than path testing through the eyes of the reader of the mainline code. Especially in programs that have global state variables, interrupts at unexpected times can lead to very odd results.

22. ***Every anticipated or potential race.***[26] Imagine two events, **A** and **B**. Both will occur, but the program is designed under the assumption that **A** will always precede **B**. This sets up a race between **A** and **B**—if **B** ever precedes **A**, the program will probably fail. To achieve race coverage, you must identify every potential race condition and then find ways, using random data or systematic test case selection, to attempt to drive **B** to precede **A** in each case.

Races can be subtle. Suppose that you can enter a value for a data item on two different data entry screens. User 1 begins to edit a record, through the first screen. In the process, the program locks the record in Table 1. User 2 opens the second screen, which calls up a record in a different table, Table 2. The program is written to automatically update the corresponding record in the Table 1 when User 2 finishes data entry. Now, suppose that User 2 finishes before User 1. Table 2 has been updated, but the attempt to synchronize Table 1 and Table 2 fails. What happens at the time of failure, or later if the corresponding records in Table 1 and 2 stay out of synch?

23. ***Every time-slice setting.*** In some systems, you can control the grain of switching between tasks or processes. The size of the time quantum that you choose can make race bugs, time-outs, interrupt-related problems, and other time-related problems more or less likely. Of course, coverage is an difficult problem here because you aren't just varying time-slice settings through every possible value. You also have to decide which tests to run under each setting. Given a planned set of test cases per setting, the coverage measure looks at the number of settings you've covered.
24. ***Varied levels of background activity.*** In a multiprocessing system, tie up the processor with competing, irrelevant background tasks. Look for effects on races and interrupt handling. Similar to time-slices, your coverage analysis must specify

---

[26] Here as in many other areas, see Appendix 1 of C. Kaner, J. Falk, and H.Q. Nguyen, *Testing Computer Software* (2nd Ed.), Van Nostrand Reinhold, 1993 for a list and discussion of several hundred types of bugs, including interrupt-related, race-condition-related, etc.

- categories of levels of background activity (figure out something that makes sense) and
- all timing-sensitive testing opportunities (races, interrupts, etc.).

25. ***Each processor type and speed***. Which processor chips do you test under? What tests do you run under each processor? You are looking for:
    - speed effects, like the ones you look for with background activity testing, and
    - consequences of processors' different memory management rules, and
    - floating point operations, and
    - any processor-version-dependent problems that you can learn about.
26. ***Every opportunity for file / record / field locking***.
27. ***Every dependency on the locked (or unlocked) state of a file, record or field.***
28. ***Every opportunity for contention for devices or resources.***
29. ***Performance of every module / task / object.*** Test the performance of a module then retest it during the next cycle of testing. If the performance has changed significantly, you are either looking at the effect of a performance-significant redesign or at a symptom of a new bug.
30. ***Free memory / available resources / available stack space at every line or on entry into and exit out of every module or object.***
31. ***Execute every line (branch, etc.) under the debug version of the operating system.*** This shows illegal or problematic calls to the operating system.
32. ***Vary the location of every file.*** What happens if you install or move one of the program's component, control, initialization or data files to a different directory or drive or to another computer on the network?
33. ***Check the release disks for the presence of every file.*** It's amazing how often a file vanishes. If you ship the product on different media, check for all files on all media.
34. ***Every embedded string in the program.*** Use a utility to locate embedded strings. Then find a way to make the program display each string.

Operation of every function / feature / data handling operation under:

35. ***Every program preference setting.***
36. ***Every character set, code page setting, or country code setting.***
37. ***The presence of every memory resident utility (inits, TSRs).***
38. ***Each operating system version.***
39. ***Each distinct level of multi-user operation.***
40. ***Each network type and version.***
41. ***Each level of available RAM.***
42. ***Each type / setting of virtual memory management.***
43. ***Compatibility with every previous version of the program.***
44. ***Ability to read every type of data available in every readable input file format.*** If a file format is subject to subtle variations (e.g. CGM) or has several sub-types (e.g. TIFF) or versions (e.g. dBASE), ***test each one.***
45. ***Write every type of data to every available output file format.*** Again, beware of subtle variations in file formats—if you're writing a CGM file, full coverage would require you to test your program's output's readability by ***every one*** of the main programs that read CGM files.
46. ***Every typeface supplied with the product.*** Check all characters in all sizes and styles. If your program adds typefaces to a collection of fonts that are available to several other programs, check compatibility with the other programs (nonstandard typefaces will crash some programs).

47. ***Every type of typeface compatible with the program.*** For example, you might test the program with (many different) TrueType and Postscript typefaces, and fixed-sized bitmap fonts.

48. ***Every piece of clip art in the product***. Test each with this program. Test each with other programs that should be able to read this type of art.

49. ***Every sound / animation provided with the product.*** Play them all under different device (e.g. sound) drivers / devices. Check compatibility with other programs that should be able to play this clip-content.

50. ***Every supplied (or constructible) script*** to drive other machines / software (e.g. macros) / BBS's and information services (communications scripts).

51. ***All commands available in a supplied communications protocol.***

52. ***Recognized characteristics.*** For example, every speaker's voice characteristics (for voice recognition software) or writer's handwriting characteristics (handwriting recognition software) or every typeface (OCR software).

53. ***Every type of keyboard and keyboard driver.***

54. ***Every type of pointing device and driver at every resolution level and ballistic setting.***

55. ***Every output feature with every sound card and associated drivers.***

56. ***Every output feature with every type of printer and associated drivers at every resolution level.***

57. ***Every output feature with every type of video card and associated drivers at every resolution level.***

58. ***Every output feature with every type of terminal and associated protocols.***

59. ***Every output feature with every type of video monitor and monitor-specific drivers at every resolution level.***

60. ***Every color shade displayed or printed to every color output device (video card / monitor / printer / etc.) and associated drivers at every resolution level.*** And check the conversion to grey scale or black and white.

61. ***Every color shade readable or scannable from each type of color input device at every resolution level.***

62. ***Every possible feature interaction between video card type and resolution, pointing device type and resolution, printer type and resolution, and memory level.*** This may seem excessively complex, but I've seen crash bugs that occur only under the pairing of specific printer and video drivers at a high resolution setting. Other crashes required pairing of a specific mouse and printer driver, pairing of mouse and video driver, and a combination of mouse driver plus video driver plus ballistic setting.

63. ***Every type of CD-ROM drive, connected to every type of port (serial / parallel / SCSI) and associated drivers.***

64. ***Every type of writable disk drive / port / associated driver.*** Don't forget the fun you can have with removable drives or disks.

65. ***Compatibility with every type of disk compression software.*** Check error handling for every type of disk error, such as full disk.

66. ***Every voltage level from analog input devices.***

67. ***Every voltage level to analog output devices.***

68. ***Every type of modem and associated drivers.***

69. ***Every FAX command (send and receive operations) for every type of FAX card under every protocol and driver.***

70. ***Every type of connection of the computer to the telephone line (direct, via PBX, etc.; digital vs. analog connection and signaling); test every phone control command under every telephone control driver.***

71. ***Tolerance of every type of telephone line noise and regional variation (including variations that are out of spec) in telephone signaling (intensity, frequency, timing, other characteristics of ring / busy / etc. tones).***

72. ***Every variation in telephone dialing plans.***

73. ***Every possible keyboard combination.*** Sometimes you'll find trap doors that the programmer used as hotkeys to call up debugging tools; these hotkeys may crash a debuggerless program. Other times, you'll discover an Easter Egg (an undocumented, probably unauthorized, and possibly embarrassing feature). ***The broader coverage***

measure is every possible keyboard combination at every error message and every data entry point. You'll often find different bugs when checking different keys in response to different error messages.

74. ***Recovery from every potential type of equipment failure.*** Full coverage includes each type of equipment, each driver, and each error state. For example, test the program's ability to recover from full disk errors on writable disks. Include floppies, hard drives, cartridge drives, optical drives, etc. Include the various connections to the drive, such as IDE, SCSI, MFM, parallel port, and serial connections, because these will probably involve different drivers.
75. ***Function equivalence.*** For each mathematical function, check the output against a known good implementation of the function in a different program. Complete coverage involves equivalence testing of all testable functions across all possible input values.
76. ***Zero handling.*** For each mathematical function, test when every input value, intermediate variable, or output variable is zero or near-zero. Look for severe rounding errors or divide-by-zero errors.
77. ***Accuracy of every graph,*** across the full range of graphable values. Include values that force shifts in the scale.
78. ***Accuracy of every report.*** Look at the correctness of every value, the formatting of every page, and the correctness of the selection of records used in each report.
79. ***Accuracy of every message.***
80. ***Accuracy of every screen.***
81. ***Accuracy of every word and illustration in the manual.***
82. ***Accuracy of every fact or statement in every data file provided with the product.***
83. ***Accuracy of every word and illustration in the on-line help.***
84. ***Every jump, search term, or other means of navigation through the on-line help.***
85. ***Check for every type of virus / worm that could ship with the program.***
86. ***Every possible kind of security violation of the program, or of the system while using the program.***
87. ***Check for copyright permissions for every statement, picture, sound clip, or other creation provided with the program.***
88. ***Verification of the program against every program requirement and published specification.***
89. ***Verification of the program against user scenarios.*** Use the program to do real tasks that are challenging and well-specified. For example, create key reports, pictures, page layouts, or other documents events to match ones that have been featured by competitive programs as interesting output or applications.
90. ***Verification against every regulation (IRS, SEC, FDA, etc.) that applies to the data or procedures of the program.***

Usability tests of:

91. ***Every feature / function of the program.***
92. ***Every part of the manual.***
93. ***Every error message.***
94. ***Every on-line help topic.***
95. ***Every graph or report provided by the program.***

Localizability / localization tests:

96. ***Every string.*** Check program's ability to display and use this string if it is modified by changing the length, using high or low ASCII characters, different capitalization rules, etc.
97. ***Compatibility with text handling algorithms under other languages (sorting, spell checking, hyphenating, etc.)***
98. ***Every date, number and measure in the program.***

99. ***Hardware and drivers, operating system versions, and memory-resident programs that are popular in other countries.***

100. ***Every input format, import format, output format, or export format that would be commonly used in programs that are popular in other countries.***

101. ***Cross-cultural appraisal of the meaning and propriety of every string and graphic shipped with the program.***