APPENDIX BUG TAXONOMY AND STATISTICS 460

CITED REFERENCES/BIBLIOGRAPHY 477

GLOSSARY/INDEX 515

Original

1

INTRODUCTION

1. THE PURPOSE OF TESTING

1.1. What We Do

Testing consumes at least half of the labor expended to produce a working program (BOEH75C, BROW73, GOOD79, RADA81, WOLV75).* Few programmers like testing and even fewer like test design—especially if test design and testing take longer than program design and coding. This attitude is understandable. Software is ephemeral: you can't point to something physical. I think, deep down, most of us don't believe in software—at least not the way we believe in hardware. If software is insubstantial, then how much more insubstantial does software testing seem? There isn't even some debugged code to point to when we're through with test design. The effort put into testing seems wasted if the tests don't reveal bugs.

There's another, deeper, problem with testing that's related to the reason we do it (MILL78B, MYER79). It's done to catch bugs. There's a myth that if we were really good at programming, there would be no bugs to catch. If only we could really concentrate, if everyone used structured programming, top-down design, decision tables, if programs were written in SQUISH, if we had the right silver bullets, then there would be no bugs. So goes the myth. There are bugs, the myth says, because we are bad at what we do; and if we are bad at it, we should feel guilty about it. Therefore, testing and test design amount to an admission of failure, which instills a goodly dose of guilt. And the tedium of testing is just punishment for our errors. Punishment for what? For being human? Guilt

* The numbers vary, but most of the apparent variability results from creative accounting. A debugger spends much time testing hypothesized causes of symptoms; so does a maintenance programmer. If we include all testing activities, of whatever they are a part, the total time spent testing by all parties ranges from 30% to 90%. If we only count formal tests conducted by an independent test group, the range is 10%–25%.

for what? For not achieving inhuman perfection? For not distinguishing between what another programmer thinks and what he says? For not being telepathic? For not solving human communication problems that have been kicked around by philosophers and theologians for 40 centuries?

The statistics show that programming, done well, will still have one to three bugs per hundred statements (AKIY71, ALBE76, BOEH75B, ENDR75, RADA81, SHOO75, THAY76, WEIS85B).* Certainly, if you have a 10% error rate, then you either need more programming education or you deserve reprimand *and* guilt.** There are some persons who claim that they can write bug-free programs. There's a saying among sailors on the Chesapeake Bay, whose sandy, shifting bottom outdates charts before they're printed, "If you haven't run aground on the Chesapeake, you haven't sailed the Chesapeake much." So it is with programming and bugs: I have them, you have them, we all have them—and the point is to do what we can to prevent them and to discover them as early as possible, but not to feel guilty about them. Programmers! Cast out your guilt! Spend half your time in joyous testing and debugging! Thrill to the excitement of the chase! Stalk bugs with care, methodology, and reason. Build traps for them. Be more artful than those devious bugs and taste the joys of guiltless programming! Testers! Break that software (as you must) and drive it to the ultimate—but don't enjoy the programmer's pain.

1.2. Productivity and Quality in Software

Consider the manufacture of a mass-produced widget. Whatever the design cost, it is a small part of the total cost when amortized over a large production run. Once in production, every manufacturing stage is subjected to quality control and testing from component source inspection to final testing before shipping. If flaws are discovered at any stage, the widget or part of it will either be discarded or cycled back for rework and correction. The assembly line's productivity is measured by the sum of the costs of the materials, the rework, and the discarded components, and the cost of quality assurance and testing. There is a trade-off between quality-assurance costs and manufacturing costs. If insufficient effort is

* Please don't use that 1% rate as a standard against which to measure programmer effectiveness. There are big variations (from 0.01 to 10) which can be explained by complexity and circumstances alone. Also, lines-of-code (usually, *k*-lines-of-code) is one of the worst complexity measures there is. See Chapter 7.

** The worst I ever saw was a 500-instruction assembly language routine with an average of 2.2 bugs per instruction after syntax checking by the assembler. That person didn't belong in programming.

spent in quality assurance, the reject rate will be high and so will the net cost. Conversely, if inspection is so good that all faults are caught as they occur, inspection costs will dominate, and again net cost will suffer. The manufacturing process designers attempt to establish a level of testing and quality assurance that minimizes net cost for a given quality objective. Testing and quality-assurance costs for manufactured items can be as low as 2% in consumer products or as high as 80% in products such as spaceships, nuclear reactors, and aircraft, where failures threaten life.

The relation between productivity and quality for software is very different from that for manufactured goods. The "manufacturing" cost of a software copy is trivial: the cost of the tape or disc and a few minutes of computer time. Furthermore, software "manufacturing" quality assurance is automated through the use of check sums and other error-detecting methods. Software costs are dominated by development. Software maintenance is unlike hardware maintenance. It is not really "maintenance" but an extended development in which enhancements are designed and installed and deficiencies corrected. The biggest part of software cost is the cost of bugs: the cost of detecting them, the cost of correcting them, the cost of designing tests that discover them, and the cost of running those tests. The main difference then between widget productivity and software productivity is that for hardware quality is only one of several productivity determinants, whereas for software, quality and productivity are almost indistinguishable.

1.3. Goals for Testing

Testing and test design, as parts of quality assurance, should also focus on bug prevention. To the extent that testing and test design do not prevent bugs, they should be able to discover symptoms caused by bugs. Finally, tests should provide clear diagnoses so that bugs can be easily corrected. Bug prevention is testing's first goal. A prevented bug is better than a detected and corrected bug because if the bug is prevented, there's no code to correct. Moreover, no retesting is needed to confirm that the correction was valid, no one is embarrassed, no memory is consumed, and prevented bugs can't wreck a schedule. More than the act of testing, the act of *designing* tests is one of the best bug preventers known. The thinking that must be done to create a useful test can discover and eliminate bugs before they are coded—indeed, test-design thinking can discover and eliminate bugs at every stage in the creation of software, from conception to specification, to design, coding, and the rest. For this reason, Dave Gelperin and Bill Hetzel (GELP87) advocate "Test, then code." The ideal test activity would be so successful at bug prevention

that actual testing would be unnecessary because all bugs would have been found and fixed during test design.*

Unfortunately, we can't achieve this ideal. Despite our effort, there will be bugs because we are human. To the extent that testing fails to reach its primary goal, *bug prevention*, it must reach its secondary goal, *bug discovery*. Bugs are not always obvious. A bug is manifested in deviations from expected behavior. A test design must document expectations, the test procedure in detail, and the results of the actual test—all of which are subject to error. But knowing that a program is incorrect does not imply knowing the bug. Different bugs can have the same manifestations, and one bug can have many symptoms. The symptoms and the causes can be disentangled only by using many small detailed tests.

1.4. Phases in a Tester's Mental Life

1.4.1. Why Testing?

What's the purpose of testing? There's an attitudinal progression characterized by the following five phases:

PHASE 0—There's no difference between testing and debugging. Other than in support of debugging, testing has no purpose.

PHASE 1—The purpose of testing is to show that the software works.

PHASE 2—The purpose of testing is to show that the software doesn't work.

PHASE 3—The purpose of testing is not to prove anything, but to reduce the perceived risk of not working to an acceptable value.

PHASE 4—Testing is not an act. It is a mental discipline that results in low-risk software without much testing effort.

1.4.2. Phase 0 Thinking

I called the inability to distinguish between testing and debugging "phase 0" because it denies that testing matters, which is why I denied it the grace of a number. See Section 2.1 in this chapter for the difference between testing and debugging. If phase 0 thinking dominates an organization, then there can be no effective testing, no quality assurance, and no quality. Phase 0 thinking was the norm in the early days of software development and dominated the scene until the early 1970s, when testing emerged as a discipline.

* I think that's what good programmers do—they test at every opportunity. "Test early and often" is their motto. It's not that they have fewer bugs, but that the habit of continual testing keeps their bugs private, and therefore cheaper.

Phase 0 thinking was appropriate to an environment characterized by expensive and scarce computing resources, low-cost (relative to hardware) software, lone programmers, small projects, and throwaway software. Today, this kind of thinking is the greatest cultural barrier to good testing and quality software. But phase 0 thinking is a problem for testers and developers today because many software managers learned and practiced programming when this mode was the norm—and it's hard to change how you think.

1.4.3. Phase 1 Thinking—The Software Works

Phase 1 thinking represented progress because it recognized the distinction between testing and debugging. This thinking dominated the leading edge of testing until the late 1970s when its fallacy was discovered. This recognition is attributed to Myers (MYER79) who observed that it is self-corrupting. It only takes one failed test to show that software doesn't work, but even an infinite number of tests won't prove that it does. The objective of phase 1 thinking is unachievable. The process is corrupted because the probability of showing that the software works *decreases* as testing increases; that is, the more you test, the likelier you are to find a bug. Therefore, if your objective is to demonstrate a high probability of working, that objective is best achieved by not testing at all! Although this conclusion may seem silly to the conscious, rational mind, it is the kind of syllogism that our unconscious mind loves to implement.

1.4.4. Phase 2 Thinking—The Software Doesn't Work

When, as testers, we shift our goal to phase 2 thinking we are no longer working in cahoots with the designers, but against them. The difference between phase 1 and 2 thinking is illustrated by analogy to the difference between bookkeepers and auditors. The bookkeeper's goal is to show that the books balance, but the auditor's goal is to show that despite the appearance of balance, the bookkeeper has embezzled. Phase 2 thinking leads to strong, revealing tests.

While one failed test satisfies the phase 2 goal, phase 2 thinking also has limits. The test reveals a bug, the programmer corrects it, the test designer designs and executes another test intended to demonstrate another bug. Phase 2 thinking leads to a never-ending sequence of ever more

diabolical tests. Taken to extremes, it too never ends, and the result is reliable software that never gets shipped. The trouble with phase 2 thinking is that we don't know when to stop.

1.4.5. Phase 3 Thinking—Test for Risk Reduction

Phase 3 thinking is nothing more than accepting the principles of statistical quality control. I say "accepting" rather than "implementing" because it's not obvious how statistical quality control should be applied to software. To the extent that testing catches bugs and to the extent that those bugs are fixed, testing does improve the product. If a test is passed, then the product's quality does not change, but our perception of that quality does. Testing, pass or fail, reduces our perception of risk about a software product. The more we test, the more we test with harsh tests, the more confidence we have in the product. We'll risk release when that confidence is high enough.*

1.4.6. Phase 4 Thinking—A State of Mind

The phase 4 thinker's knowledge of what testing can and can't do, combined with knowing what makes software testable, results in software that doesn't need much testing to achieve the lower-phase goals. Testability is the goal for two reasons. The first and obvious reason is that we want to reduce the labor of testing. The second and more important reason is that testable code has fewer bugs than code that's hard to test. The impact on productivity of these two factors working together is multiplicative. What makes code testable? One of the main reasons to learn test techniques is to answer that question.

1.4.7. Cumulative Goals

The above goals are cumulative. Debugging depends on testing as a tool for probing hypothesized causes of symptoms. There are many ways to break software that have nothing to do with the software's functional requirements: phase 2 tests alone might never show that the software does what it's supposed to do. It's impractical to break software until the easy demonstrations of workability are behind you. Use of statistical

* It would be nice if such statistical methods could be applied as easily to software as they are to widgets. Today, statistical quality control can only be applied to large software products with long histories such as 20 million lines of code and 10 years of use. Application to small products or components as a means to determine when the component can be released is dangerous.

methods as a guide to test design, as a means to achieve good testing at acceptable risks, is a way of fine-tuning the process. It should be applied only to large, robust products with few bugs. Finally, a state of mind isn't enough: even the most testable software must be debugged, must work, and must be hard to break.

1.5. Test Design

Although programmers, testers, and programming managers know that code must be designed and tested, many appear to be unaware that tests themselves must be designed and tested—designed by a process no less rigorous and no less controlled than that used for code. Too often, test cases are attempted without prior analysis of the program's requirements or structure. Such test design, if you can call it that, is just a haphazard series of ad-lib cases that are not documented either before or after the tests are executed. Because they were not formally designed, they cannot be precisely repeated, and no one is sure whether there was a bug or not. After the bug has been ostensibly corrected, no one is sure that the retest was identical to the test that found the bug. Ad-lib tests are useful during debugging, where their primary purpose is to help locate the bug, but ad-lib tests done in support of debugging, no matter how exhausting, are not substitutes for *designed* tests.

The test-design phase of programming should be explicitly identified. Instead of "design, code, desk check, test, and debug," the programming process should be described as: "design, test design, code, test code, program inspection, test inspection, test debugging, test execution, program debugging, testing." Giving test design an explicit place in the scheme of things provides more visibility to that amorphous half of the labor that often goes under the name "test and debug." It makes it less likely that test design will be given short shrift when the budget's small and the schedule's tight and there's a vague hope that maybe this time, just this once, the system will come together without bugs.

1.6. Testing Isn't Everything

This is a book on testing techniques, which are only *part* of our weaponry against bugs. Research and practice (BASI87, FAGA76, MYER78, WEIN65, WHIT87) show that other approaches to the creation of good software are possible and essential. Testing, I believe, is still our most potent weapon, but there's evidence (FAGA76) that other methods may be as effective: but you can't implement inspections, say, *instead* of

testing because testing and inspections catch or prevent different kinds of bugs. Today, if we want to prevent all the bugs that we can and catch those that we don't prevent, we must review, inspect, read, do walkthroughs, *and then test*. We don't know today the mix of approaches to use under what circumstances. Experience shows that the "best mix" *very much* depends on things such as development environment, application, size of project, language, history, and culture. The other major methods in decreasing order of effectiveness are as follows:

Inspection Methods—In this category I include walkthroughs, desk checking, formal inspections (FAGA76), and code reading. These methods appear to be as effective as testing, but the bugs caught do not completely overlap.

Design Style—By this term I mean the stylistic criteria used by programmers to define what they mean by a "good" program. Sticking to outmoded style, such as "tight" code or "optimizing" for performance destroys quality. Conversely, adopting stylistic objectives such as testability, openness, and clarity can do much to prevent bugs.

Static Analysis Methods—These methods include anything that can be done by formal analysis of source code during or in conjunction with compilation. Syntax checking in early compilers was rudimentary and was part of the programmer's "testing." Compilers have taken that job over (thank the Lord). **Strong typing** and **type checking** eliminate an entire category of bugs. There's a lot more that can be done to detect errors by static analysis. It's an area of intensive research and development. For example, much of **data-flow anomaly** detection (see Chapters 5 and 8), which today is part of testing, will eventually be incorporated into the compiler's static analysis.

Languages—The source language can help reduce certain kinds of bugs. Languages continue to evolve, and preventing bugs is a driving force in that evolution. Curiously, though, programmers find new kinds of bugs in new languages, so the bug rate seems to be independent of the language used.

Design Methodologies and Development Environment—The design methodology (that is, the development process used and the environment in which that methodology is embedded), can prevent many kinds of bugs. For example, configuration control and automatic distribution of change information prevents bugs which result from a programmer's unawareness that there were changes.

1.7. The Pesticide Paradox and the Complexity Barrier

You're a poor farmer growing cotton in Alabama and the boll weevils are destroying your crop. You mortgage the farm to buy DDT, which you spray on your field, killing 98% of the pest, saving the crop. The next year, you spray the DDT early in the season, but the boll weevils still eat your crop because the 2% you didn't kill last year were resistant to DDT. You now have to mortgage the farm to buy DDT *and* Malathion; then next year's boll weevils will resist both pesticides and you'll have to mortgage the farm yet again. That's the pesticide paradox* for boll weevils and also for software testing.

First Law: The Pesticide Paradox—Every method you use to prevent or find bugs leaves a residue of subtler bugs against which those methods are ineffectual.

That's not too bad, you say, because at least the software gets better and better. Not quite!

Second Law: The Complexity Barrier—Software complexity (and therefore that of bugs) grows to the limits of our ability to manage that complexity.

By eliminating the (previous) easy bugs you allowed another escalation of features and complexity, but this time you have subtler bugs to face, just to retain the reliability you had before. Society seems to be unwilling to limit complexity because we all want that extra bell, whistle, and feature interaction. Thus, our users always push us to the complexity barrier and how close we can approach that barrier is largely determined by the strength of the techniques we can wield against ever more complex and subtle bugs.

2. SOME DICHOTOMIES

2.1. Testing Versus Debugging

Testing and debugging are often lumped under the same heading, and it's no wonder that their roles are often confused: for some, the two words are

* The pesticide paradox for boll weevils is resolved by surrounding the cotton field with a sacrificial crop that the boll weevils prefer to cotton. Would that we could create a sacrificial subroutine in all software systems that would attract all the bugs.

synonymous; for others, the phrase "test and debug" is treated as a single word. The **purpose of testing** is to show that a program has bugs. The **purpose of debugging** is find the error or misconception that led to the program's failure and to design and implement the program changes that correct the error. Debugging usually follows testing, but they differ as to goals, methods, and most important, psychology:

1. Testing starts with known conditions, uses predefined procedures, and has predictable outcomes; only whether or not the program passes the test is unpredictable. Debugging starts from possibly unknown initial conditions, and the end cannot be predicted, except statistically.
2. Testing can and should be planned, designed, and scheduled. The procedures for, and duration of, debugging cannot be so constrained.
3. Testing is a demonstration of error or apparent correctness. Debugging is a deductive process.
4. Testing proves a programmer's failure. Debugging is the programmer's vindication.
5. Testing, as executed, should strive to be predictable, dull, constrained, rigid, and inhuman. Debugging demands intuitive leaps, conjectures, experimentation, and freedom.
6. Much of testing can be done without design knowledge. Debugging is impossible without detailed design knowledge.
7. Testing can often be done by an outsider. Debugging must be done by an insider.
8. Although there is a robust theory of testing that establishes theoretical limits to what testing can and can't do, debugging has only recently been attacked by theorists—and so far there are only rudimentary results.
9. Much of test execution and design can be automated. Automated debugging is still a dream.

2.2. Function Versus Structure

Tests can be designed from a functional or a structural point of view. In **functional testing** the program or system is treated as a black box. It is subjected to inputs, and its outputs are verified for conformance to specified behavior. The software's user should be concerned only with functionality and features, and the program's implementation details should not matter. Functional testing takes the user's point of view.

Structural testing does look at the implementation details. Such things as programming style, control method, source language, database design, and coding details dominate structural testing; but the boundary between function and structure is fuzzy. Good systems are built in layers—from the outside to the inside. The user sees only the outermost layer, the layer of pure function. Each layer inward is less related to the system's functions and more constrained by its structure: so what is structure to one layer is function to the next. For example, the user of an online system doesn't know that the system has a memory-allocation routine. For the user, such things are structural details. The memory-management routine's designer works from a specification for that routine. The specification is a definition of "function" at that layer. The memory-management routine uses a link-block subroutine. The memory-management routine's designer writes a "functional" specification for a link-block subroutine, thereby defining a further layer of structural detail and function. At deeper levels, the programmer views the operating system as a structural detail, but the operating system's designer treats the computer's hardware logic as the structural detail.

Most of this book is devoted to models of programs and the tests that can be designed by using those models. A given model, and the associated tests may be first introduced in a structural context but later used again in a functional context, or vice versa. The initial choice of how to present a model was based on the context that seemed most natural for that model and in which it was likeliest that the model would be used for test design. Just as you can't clearly distinguish function from structure, you can't fix the utility of a model to structural tests or functional tests. If it helps you design effective tests, then use the model in whatever context it seems to work.

There's no controversy between the use of structural versus functional tests: both are useful, both have limitations, both target different kinds of bugs. Functional tests can, in principle, detect all bugs but would take infinite time to do so. Structural tests are inherently finite but cannot detect all errors, even if completely executed. The art of testing, in part, is in how you choose between structural and functional tests.

2.3. The Designer Versus the Tester

If testing were wholly based on functional specifications and independent of implementation details, then the designer and the tester could be completely separated. Conversely, to design a test plan based only on a system's structural details would require the software designer's knowledge,

and hence only she could design the tests. The more you know about the design, the likelier you are to eliminate useless tests, which, despite functional differences, are actually handled by the same routines over the same paths; but the more you know about the design, the likelier you are to have the same misconceptions as the designer. Ignorance of structure is the independent tester's best friend and worst enemy. The naive tester has no preconceptions about what is or is not possible and will, therefore, design tests that the program's designer would never think of—and many tests that never should be thought of. Knowledge, which is the designer's strength, brings efficiency to testing but also blindness to missing functions and strange cases. Tests designed and executed by the software designers are by nature biased toward structural considerations and therefore suffer the limitations of structural testing. Tests designed and executed by an independent tester are bias-free and can't be finished. Part of the artistry of testing is to balance knowledge and its biases against ignorance and its inefficiencies.

In this book I discuss the "tester," "test-team member," or "test designer" in contrast to the "programmer" and "program designer," as if they were distinct persons. As one goes from **unit testing** to **unit integration**, to **component testing** and integration, to **system testing**, and finally to formal **system feature testing**, it is increasingly more effective if the "tester" and "programmer" are different persons. The techniques presented in this book can be used for all testing—from unit to system. When the technique is used in system testing, the designer and tester are probably different persons; but when the technique is used in unit testing, the tester and programmer merge into one person, who sometimes acts as a programmer and sometimes as a tester.

You must be a constructive schizophrenic. Be clear about the difference between your role as a programmer and as a tester. The tester in you must be suspicious, uncompromising, hostile, and compulsively obsessed with destroying, utterly destroying, the programmer's software. The tester in you is your Mister Hyde—your Incredible Hulk. He must exercise what Gruenberger calls "low cunning." (HETZ73) The programmer in you is trying to do a job in the simplest and cleanest way possible, on time, and within budget. Sometimes you achieve this by having great insights into the programming problem that reduce complexity and labor and are almost correct. And with that tester/Hulk lurking in the background of your mind, it pays to have a healthy paranoia about bugs. Remember, then, that when I refer to the "test designer" and "programmer" as separate persons, the extent to which they are separated depends on the testing level and the context in which the technique is applied. This

saves me the effort of writing about the same technique twice and you the tedium of reading it twice.

2.4. Modularity Versus Efficiency

Both tests and systems can be modular. A **module** is a discrete, well-defined, small component of a system. The smaller the component, the easier it is to understand; but every component has interfaces with other components, and *all* interfaces are sources of confusion. The smaller the component, the likelier are interface bugs. Large components reduce external interfaces but have complicated internal logic that may be difficult or impossible to understand. Part of the artistry of software design is setting component size and boundaries at points that balance internal complexity against interface complexity to achieve an overall complexity minimization.

Testing can and should likewise be organized into modular components. Small, independent test cases have the virtue of easy repeatability. If an error is found by testing, only the small test, not a large component that consists of a sequence of hundreds of interdependent tests, need be rerun to confirm that a test design bug has been fixed. Similarly, if the test has a bug, only that test need be changed and not a whole test plan. But microscopic test cases require individual setups and each such setup (e.g., data, inputs) can have bugs. As with system design, artistry comes into test design in setting the scope of each test and groups of tests so that test design, test debugging, and test execution labor are minimized without compromising effectiveness.

2.5. Small Versus Large

I often write small analytical programs of a few hundred lines that, once used, are discarded. Do I use formal test techniques, quality assurance, and all the rest I so passionately advocate? Of course not, and I'm not a hypocrite. I do what everyone does in similar circumstances: I design, I code, I test a few cases, debug, redesign, recode, and so on, much as I did 30 years ago. I can get away with such (slovenly) practices because I'm programming for a very small, intelligent, forgiving, user population—me. It's the ultimate of small programs and it is most efficiently done by intuitive means and complete lack of formality.

Let's up the scale to a larger package. I'm still the only programmer and user, but now, the package has thirty components averaging 750 statements each, developed over a period of 5 years. Now I *must* create and

maintain a data dictionary and do thorough unit testing. But I'll take my own word for it and not bother to retain all those test cases or to exercise formal configuration control.

You can extrapolate from there or draw on your experiences. **Programming in the large** (DERE76) means constructing programs that consist of many components written by many different persons. **Programming in the small** is what we do for ourselves in the privacy of our own offices or as homework exercises in an undergraduate programming course. Size brings with it nonlinear scale effects, which are imperfectly understood today. Qualitative changes occur with size and so must testing methods and quality criteria. A primary example is the notion of **coverage**—a measure of test completeness. Without worrying about exactly what these terms mean, 100% coverage is essential for unit testing, but we back off this requirement as we deal with ever larger software aggregates, accept 75%–85% for most systems, and possibly as low as 50% for huge systems of 10 million lines of code or so.

2.6. The Builder Versus the Buyer

Most software is written and used by the same organization. Unfortunately, this situation is dishonest because it clouds accountability. Many organizations today recognize the virtue of independent software development and operation because it leads to better software, better security, and better testing. Independent software development does not mean that all software should be bought from software houses or consultants but that the software developing entity and the entity that pays for the software be separated enough to make accountability clear. I've heard of cases where the software development group and the operational group within the same company negotiate and sign formal contracts with one another—with lawyers present. If there is no separation between builder and buyer, there can be no accountability. If there is no accountability, the motivation for software quality disappears and with it any serious attempt to do proper testing.

Just as programmers and testers can merge and become one, so can builder and buyer. There are several other persons in the software development cast of characters who, like the above, can also be separated or merged:

1. The **builder,** who designs for and is accountable to
2. The **buyer,** who pays for the system in the hope of profits from providing services to
3. The **user,** the ultimate beneficiary or victim of the system. The user's interests are guarded by
4. The **tester,** who is dedicated to the builder's destruction and
5. The **operator,** who has to live with the builder's mistakes, the buyer's murky specifications, the tester's oversights, and the user's complaints.

3. A MODEL FOR TESTING

3.1. The Project

Testing is applied to anything from subroutines to systems that consist of millions of statements. The archetypical system is one that allows the exploration of all aspects of testing without the complications that have nothing to do with testing but affect any very large project. It's medium-scale programming. Testing the interfaces between different parts of your own mind is very different from testing the interface between you and other programmers separated from you by geography, language, time, and disposition. Testing a one-shot routine that will be run only a few times is very different from testing one that must run for decades and may be modified by some unknown future programmer. Although all the problems of the solitary routine occur for the routine that is embedded in a system, the converse is not true: many kinds of bugs just can't exist in solitary routines. There is an implied context for the test methods discussed in this book—a real-world context characterized by the following model project:

Application—The specifics of the application are unimportant. It is a real-time system that must provide timely responses to user requests for services. It is an online system connected to remote terminals.

Staff—The programming staff consists of twenty to thirty programmers—big enough to warrant formality, but not too big to manage—big enough to use specialists for some parts of the system's design.

Schedule—The project will take 24 months from the start of design to formal acceptance by the customer. Acceptance will be followed by a 6-month cutover period. Computer resources for development and testing will be almost adequate.

Specification—The specification is good. It is functionally detailed without constraining the design, but there are undocumented "understandings" concerning the requirements.

Acceptance Test—The system will be accepted only after a formal acceptance test. The application is not new, so part of the formal test already exists. At first the customer will intend to design the acceptance test, but later it will become the software design team's responsibility.

Personnel—The staff is professional and experienced in programming and in the application. Half the staff has programmed that computer before and most know the source language. One-third, mostly junior programmers, have no experience with the application. The typical programmer has been employed by the programming department for 3 years. The climate is open and frank. Management's attitude is positive and knowledgeable about the realities of such projects.

Standards—Programming and test standards exist and are usually followed. They understand the role of interfaces and the need for interface standards. Documentation is good. There is an internal, semiformal, quality-assurance function. The database is centrally developed and administered.

Objectives—The system is the first of many similar systems that will be implemented in the future. No two will be identical, but they will have 75% of the code in common. Once installed, the system is expected to operate profitably for more than 10 years.

Source—One-third of the code is new, one-third extracted from a previous, reliable, but poorly documented system, and one-third is being rehosted (from another language, computer, operating system—take your pick).

History—One programmer will quit before his components are tested. Another programmer will be fired before testing begins: excellent work, but poorly documented. One component will have to be redone after unit testing: a superb piece of work that defies integration. The customer will insist on five big changes and twenty small ones. There will be at least one nasty problem that nobody—not the customer, not the programmer, not the managers, nor the hardware vendor—suspected. A facility and/or hardware delivery problem will delay testing for several weeks and force second- and third-shift work. Several important milestones will slip but the delivery date will be met.

Our model project is a typical well-run, successful project with a share of glory and catastrophe—neither a utopian project nor a slice of hell.

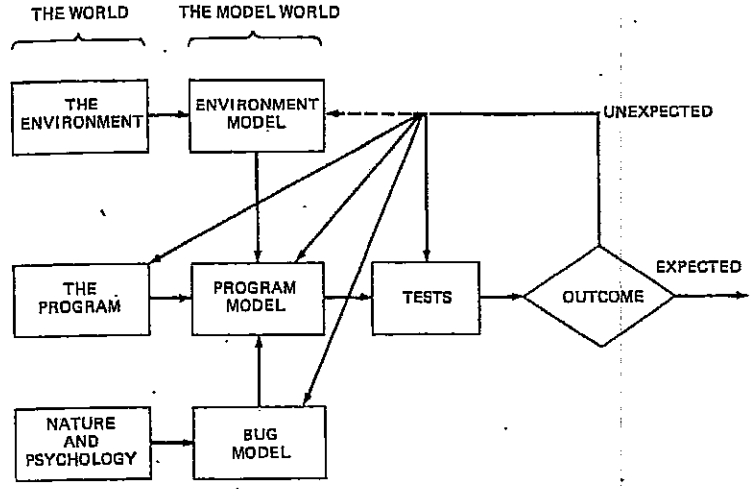

Figure 1.1. A Model of Testing.

3.2. Overview

Figure 1.1 is a model of the testing process. The process starts with a program embedded in an environment, such as a computer, an operating system, or a calling program. We understand human nature and its susceptibility to error. This understanding leads us to create three models: a model of the environment, a model of the program, and a model of the expected bugs. From these models we create a set of tests, which are then executed. The result of each test is either expected or unexpected. If unexpected, it may lead us to revise the test, our model or concept of how the program behaves, our concept of what bugs are possible, or the program itself. Only rarely would we attempt to modify the environment.

3.3. The Environment

A **program's environment** is the hardware and software required to make it run. For online systems the environment may include communications lines, other systems, terminals, and operators. The environment also includes all programs that interact with—and are used to create—the pro-

gram under test, such as operating system, loader, linkage editor, compiler, utility routines.

Programmers should learn early in their careers that it's not smart to blame the environment (that is, hardware and firmware) for bugs. Hardware bugs are rare. So are bugs in manufacturer-supplied software. This isn't because logic designers and operating system programmers are better than application programmers, but because such hardware and software is stable, tends to be in operation for a long time, and most bugs will have been found and fixed by the time programmers use that hardware or software.* Because hardware and firmware are stable, we don't have to consider all of the environment's complexity. Instead, we work with a simplification of it, in which only the features most important to the program at hand are considered. Our model of the environment includes our *beliefs* regarding such things as the workings of the computer's instruction set, operating system macros and commands, and what a higher-order language statement will do. If testing reveals an unexpected result, we may have to change our beliefs (our model of the environment) to find out what went wrong. But sometimes the environment could be wrong: the bug could be in the hardware or firmware after all.

3.4. The Program

Most programs are too complicated to understand in detail. We must simplify our concept of the program in order to test it. So although a real program is exercised on the test bed, in our brains we deal with a simplified version of it—a version in which most details are ignored. If the program calls a subroutine, we tend not to think about the subroutine's details unless its operation is suspect. Similarly, we may ignore processing details to focus on the program's control structure or ignore control structure to focus on processing. As with the environment, if the simple model of the program does not explain the unexpected behavior, we may have to modify that model to include more facts and details. And if that fails, we may have to modify the program.

3.5. Bugs

Bugs are more insidious than ever we expect them to be. Yet it is convenient to categorize them: initialization, call sequence, wrong variable, and so on. Our notion of what is or isn't a bug varies. A bad specification may lead us to mistake good behavior for bugs, and vice versa. An unexpected

* But new operating systems and firmware are as buggy as new application software.

test result may lead us to change our notion of what a bug is—that is to say, our model of bugs.

While we're on the subject of bugs, I'd like to dispel some optimistic notions that many programmers and testers have about bugs. Most programmers and testers have beliefs about bugs that express a naivete that ranks with belief in the tooth fairy. If you hold any of the following beliefs, then disabuse yourself of them because as long as you believe in such things you will be unable to test effectively and unable to justify the dirty tests most programs need.

Benign Bug Hypothesis—The belief that bugs are nice, tame, and logical. Only weak bugs have a logic to them and are amenable to exposure by strictly logical means. Subtle bugs have no definable pattern—they are wild cards.

Bug Locality Hypothesis—The belief that a bug discovered within a component affects only that component's behavior; that because of structure, language syntax, and data organization, the symptoms of a bug are localized to the component's designed domain. Only weak bugs are so localized. Subtle bugs have consequences that are arbitrarily far removed from the cause in time and/or space from the component in which they exist.

Control Bug Dominance—The belief that errors in the control structure of programs dominate the bugs. While many easy bugs, especially in components, can be traced to **control-flow** errors, **data-flow** and data-structure errors are as common. Subtle bugs that violate data-structure boundaries and data/code separation can't be found by looking only at control structures.

Code/Data Separation—The belief, especially in HOL programming, that bugs respect the separation of code and data.* Furthermore, in real systems the distinction between code and data can be hard to make, and it is exactly that blurred distinction that permit such bugs to exist.

Lingua Salvator Est—The hopeful belief that language syntax and semantics (e.g., structured coding, strong typing, complexity hiding) eliminates most bugs. True, good language features do help prevent the simpler component bugs but there's no statistical evidence to support the notion that such features help with subtle bugs in big systems.

* Think about it. How do most programs crash? Either by executing data or by inadvertent modifications of code—in either case, a supposedly impossible violation of code/data separation.

Corrections Abide—The mistaken belief that a corrected bug remains corrected. Here's a generic counterexample. A bug is believed to have symptoms caused by the interaction of components A and B but the real problem is a bug in C, which left a residue in a data structure used by both A and B. The bug is "corrected" by changing A and B. Later, C is modified or removed and the symptoms of A and B recur. Subtle bugs are like that.

Silver Bullets—The mistaken belief that X (language, design method, representation, environment—name your own) grants immunity from bugs. Easy-to-moderate bugs may be reduced, but remember the pesticide paradox.

Sadism Suffices—The common belief, especially by independent testers, that a sadistic streak, low cunning, and intuition are sufficient to extirpate most bugs. You only catch easy bugs that way. Tough bugs need methodology and techniques, so read on.

Angelic Testers—The ludicrous belief that testers are better at test design than programmers are at code design.*

3.6. Tests

Tests are formal procedures. Inputs must be prepared, outcomes predicted, tests documented, commands executed, and results observed; all these steps are subject to error. There is nothing magical about testing and test design that immunizes testers against bugs. An unexpected test result is as often cause by a test bug as it is by a real bug.* Bugs can creep into the documentation, the inputs, and the commands and becloud our observation of results. An unexpected test result, therefore, may lead us to revise the tests. Because the tests are themselves in an environment, we also have a mental model of the tests, and instead of revising the tests, we may have to revise that mental model.

3.7. Testing and Levels

We do three distinct kinds of testing on a typical software system: **unit/component testing, integration testing,** and **system testing.** The objectives

* Not universal. For mature organizations with an established, sophisticated design and independent testing process in place, bug rates are approximately equal, where we can (roughly) equate one line of source code to one subtest. An immature process, characterized by inadequate component tests, appears to show programmer bug rates that are about 10 times higher than independent testers' bug rates (in tests). As programmers get better at component testing and inspections, their bug rates go down and eventually approximates the testers' bug rates. Tester bug rates also drop, but not as fast. Actually, we should expect programmer bug rates to be slightly lower than tester bug rates because there are more aspects of test design that cannot be automated than for comparable programming activities. Supporting evidence for comparisons of tester and programmer bug rates is sparse and mostly anecdotal.

of each class is different and therefore, we can expect the mix of test methods used to differ. They are:

Unit, Unit Testing—A **unit** is the smallest testable piece of software, by which I mean that it can be compiled or assembled, linked, loaded, and put under the control of a **test harness** or **driver.** A **unit** is usually the work of one programmer and it consists of several hundred or fewer, lines of source code. **Unit testing** is the testing we do to show that the unit does not satisfy its functional specification and/or that its implemented structure does not match the intended design structure. When our tests reveal such faults, we say that there is a **unit bug.**

Component, Component Testing—A **component** is an **integrated aggregate** of one or more units. A unit is a component, a component with subroutines it calls is a component, etc. By this (recursive) definition, a component can be anything from a unit to an entire system. **Component testing** is the testing we do to show that the component does not satisfy its functional specification and/or that its implemented structure does not match the intended design structure. When our tests reveal such problems, we say that there is a **component bug.**

Integration, Integration Testing—**Integration** is a *process* by which components are aggregated to create larger components. **Integration testing** is testing done to show that even though the components were individually satisfactory, as demonstrated by successful passage of component tests, the combination of components are incorrect or inconsistent. For example, components A and B have both passed their component tests. Integration testing is aimed as showing inconsistencies between A and B. Examples of such inconsistencies are improper call or return sequences, inconsistent data validation criteria, and inconsistent handling of data objects. Integration testing should not be confused with testing integrated objects, which is just higher level component testing. Integration testing is specifically aimed at exposing the problems that arise from the combination of components. The sequence, then, consists of component testing for components A and B, integration testing for the combination of A and B, and finally, component testing for the "new" component (A,B).*

* I'm indebted to one of my seminar students for the following elegant illustration of component and integration testing issues. Consider a subroutine A, which calls itself recursively. Initial component testing does not include the called subcomponents; therefore the recursive call of A by A is not tested. Integration testing is the test of the A call and return. The new, integrated component is clearly a different kind of component because it invokes the recursive call support mechanisms (e.g., the stack), which were not tested before; therefore, as a "new" component, it needs additional testing.

System, System Testing—A system is a big component. System testing is aimed at revealing bugs that cannot be attributed to components as such, to the inconsistencies between components, or to the planned interactions of components and other objects. System testing concerns issues and behaviors that can only be exposed by testing the entire integrated system or a major part of it. System testing includes testing for performance, security, accountability, configuration sensitivity, start-up, and recovery.

This book concerns component testing, but the techniques discussed here also apply to integration and system testing. There aren't any special integration and system testing techniques but the mix of effective techniques changes as our concern shifts from components to integration, to system. How and where integration and system testing will be covered is discussed in the preface to this book. You'll find comments on techniques concerning their relative effectiveness as applied to component, integration, and system testing throughout the book. Such comments are intended to guide your selection of a mix of techniques that best matches your testing concerns, be it component, integration, or system, or some mixture of the three.

3.8. The Role of Models

Testing is a process in which we create mental models of the environment, the program, human nature, and the tests themselves. Each model is used either until we accept the behavior as correct or until the model is no longer sufficient for the purpose. Unexpected test results always force a revision of some mental model, and in turn may lead to a revision of whatever is being modeled. The revised model may be more detailed, which is to say more complicated, or more abstract, which is to say simpler. The art of testing consists of creating, selecting, exploring, and revising models. Our ability to go through this process depends on the number of different models we have at hand and their ability to express a program's behavior.

4. PLAYING POOL AND CONSULTING ORACLES

4.1. Playing Pool

Testing is like playing pool. There's real pool and there's kiddie pool. In kiddie pool you hit the balls, and whatever pocket they fall into you claim as the intended pocket. It's not much of a game and, though suitable for 10-year-olds, it's no challenge for an adult. The objective of real pool is to

specify the pocket in advance; similarly for testing. There's real testing and there's kiddie testing. In kiddie testing the tester says, after the fact, that the observed outcome of the test was the expected outcome. In real testing *the outcome is predicted and documented before the test is run.* If a programmer can't reliably predict the outcome of a test before it is run, then that programmer doesn't understand how the program works or what it's supposed to be doing. The tester who can't make that kind of prediction doesn't understand the program's functional objectives. Such misunderstandings lead to bugs, either in the program or in its tests, or both.

4.2. Oracles

An **oracle** (HOWD78B) is any program, process, or body of data that specifies the expected outcome of a set of tests as applied to a tested object. There are as many different kinds of oracles as there are testing concerns. The most common oracle is an **input/outcome oracle**—an oracle that specifies the expected outcome for a specified input. When used without qualification, the term means input/outcome oracle. Other types of oracles are defined in the glossary and will be introduced as required.

4.3. Sources of Oracles

If every test designer had to analyze and predict the expected behavior for every test case for every component, then test design would be very expensive. The hardest part of test design is predicting the expected outcome, but we often have oracles that reduce the work. Here are some sources of oracles:

Kiddie Testing—Run the test and see what comes out. No, I didn't lie and I'm not contradicting myself. It's a question of discipline. If you have the outcome in front of you, and especially if you have intermediate values of internal variables, then it's much easier to validate that outcome by analysis and show it to be correct than it is to predict what the outcome should be and validate your prediction. The problem with kiddie testing as an oracle is that it is very hard to distinguish between its use as an adjunct to prediction and as pure kiddie testing. If the personal discipline and the controls are there, and if there is a documented analysis that justifies the predicted outcomes, then does it matter if that analysis was aided by kiddie testing?

Regression Test Suites—Today, software development and testing are dominated not by the design of new software but by rework and maintenance of existing software. In such instances, most of the tests you

need will have been run on a previous version. Most of those tests should have the same outcome for the new version. Outcome prediction is therefore needed only for changed parts of components.

Purchased Suites and Oracles—Highly standardized software that (should) differ only as to implementation often has commercially available test suites and oracles. The most common examples are compilers for standard languages, communications protocols, and mathematical routines. As more software becomes standardized, more oracles will emerge as products and services.

Existing Program—A working, trusted program is an excellent oracle. The typical use is when the program is being rehosted to a new language, operating system, environment, configuration, or to some combination of these, with the intention that the behavior should not change as a result of the rehosting.

5. IS COMPLETE TESTING POSSIBLE?

If the objective of testing were to *prove* that a program is free of bugs, then testing not only would be practically impossible, but also would be theoretically impossible. Three different approaches can be used to demonstrate that a program is correct: tests based on structure, tests based on function, and formal proofs of correctness. Each approach leads to the conclusion that complete testing, in the sense of a *proof* is neither theoretically nor practically possible (MANN78).

Functional Testing—Every program operates on a finite number of inputs. Whatever pragmatic meaning those inputs might have, they can always be interpreted as a binary bit stream. A complete functional test would consist of subjecting the program to all possible input streams. For each input the routine either accepts the stream and produces a correct outcome, accepts the stream and produces an incorrect outcome, or rejects the stream and tells us that it did so. Because the rejection message is itself an outcome, the problem is reduced to verifying that the correct outcome is produced for every input. But a 10-character input string has 2^{80} possible input streams and corresponding outcomes. So complete functional testing in this sense is impractical.*

But even theoretically, we can't execute a purely functional test this way because we don't know the length of the string to which the system is responding. Let's say that the routine should respond to a 10-character string. It should be reset after the tenth character, and the next 10 characters should constitute a new test. Unknown to us, the routine has a huge buffer and is actually responding to 10,000-character strings. The bug is such that the program will appear to provide a proper outcome for every 10-character sequence the first thousand times and fail on the 1001st attempt. Without a limit to the routine's memory capacity, which is a structural concept, it is impossible even in principle to prove that the routine is correct.

* At one test per microsecond, twice the current estimated age of the universe.

There are two more problems: the input sequence generator and the outcome verifier. Should we assume that the hardware and software used to generate the inputs, to compare the real outcome to the expected outcome, and to document the expected outcome are bug-free? Pure functional testing is at best conditional on an unverifiable assumption that all test tools and test preparation tools are correct and that only the tested routine is at fault; in the real world of testing, that assumption is silly.

Structural Testing—One should design enough tests to ensure that every path through the routine is exercised at least once. Right off that's impossible, because some loops might never terminate. Brush that problem aside by observing that the universe—including all that's in it—is finite. Even so, the number of paths through a small routine can be awesome because each loop multiplies the path count by the number of times through the loop. A small routine can have millions or billions of paths, so total **path testing** is usually impractical, although it can be done for some routines. By doing it we solve the problems of unknown size that we ran into for purely functional testing; however, it doesn't solve the problem of preparing a bug-free input, a bug-free response list, and a bug-free test observation. We still need those things, because pure structural testing can never assure us that the routine is doing the right thing.

Correctness Proofs—Formal proofs of correctness rely on a combination of functional and structural concepts. Requirements are stated in a formal language (e.g., mathematics), and each program statement is examined and used in a step of an inductive proof that the routine will produce the correct outcome for all possible input sequences. The practical issue here is that such proofs are very expensive and have been applied only to numerical routines or to formal proofs for crucial software such as a system's security kernel or portions of compilers. But there are theoretical objections to formal proofs of correctness that go beyond the practical issues. How do we know that the specification is achievable? Its consistency and completeness must be proved, and in

general, that is a provably unsolvable problem. Assuming that the specification has been proved correct, then the mechanism used to prove the program, the steps in the proof, the logic used, and so on, must be proved (GOOD75). Mathematicians and logicians have no more immunity to bugs than programmers or testers have. This also leads to never-ending sequences of unverifiable assumptions.

Manna and Waldinger (MANN78) have clearly summarized the theoretical barriers to complete testing:

"We can never be sure that the specifications are correct."

"No verification system can verify every correct program."

"We can never be certain that a verification system is correct."

Not only are all known approaches to absolute demonstrations of correctness impractical, but they are impossible. Therefore, our objective must shift from an absolute proof to a suitably convincing demonstration—from a deduction to a seduction. That word "suitable," if it is to have the same meaning to everyone, implies a quantitative measure, which in turn implies a statistical measure of software reliability. Our goal, then, should be to provide enough testing to ensure that the probability of failure due to hibernating bugs is low enough to accept. "Enough" implies judgment. What is enough to a video game is insufficient to a nuclear reactor. We can expect that each application will eventually evolve its own software reliability standards. Concurrently, test techniques and reliability models will evolve so that it will be possible, based on test results, to make a quantitative prediction of the routine's reliability.

2

THE TAXONOMY OF BUGS

1. SYNOPSIS

What are the possible consequences of bugs? Bugs are categorized. Statistics and occurrence frequency of various bugs are given.

2. THE CONSEQUENCES OF BUGS

2.1. The Importance of Bugs

The importance of a bug depends on frequency, correction cost, installation cost, and consequences.

Frequency—How often does that kind of bug occur? See Table 2.1 on page 57 for bug frequency statistics. Pay more attention to the more frequent bug types.

Correction Cost—What does it cost to correct the bug after it's been found? That cost is the sum of two factors: (1) the cost of discovery and (2) the cost of correction. These costs go up dramatically the later in the development cycle the bug is discovered. Correction cost also depends on system size. The larger the system the more it costs to correct the same bug.

Installation Cost—Installation cost depends on the number of installations: small for a single-user program, but how about a PC operating system bug? Installation cost can dominate all other costs—fixing one simple bug and distributing the fix could exceed the entire system's development cost.

Consequences—What are the consequences of the bug? You might measure this by the mean size of the awards made by juries to the victims of your bug.