<ch1 toc ch3>

Chapter 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.

A reasonable metric for bug importance is:

```
importance($) = frequency*(correction_cost + installation_cost + consequential_cost)
```

Frequency tends not to depend on application or environment, but correction, installation, and consequential costs do. As designers, testers, and QA workers, you must be interested in bug importance, not raw frequency. Therefore you must create your own importance model. This chapter will help you do that.

2.2. How Bugs Affect Us—Consequences

Bug consequences range from mild to catastrophic. Consequences should be measured in human rather than machine terms because it is ultimately for humans that we write programs. If you answer the question, "What are the consequences of this bug?" in machine terms by saying, for example, "Bit so-and-so will be set instead of reset," you're avoiding responsibility for the bug. Although it may be difficult to do in the scope of a subroutine, programmers should try to measure the consequences of their bugs in human terms. Here are some consequences on a scale of one to ten:

1. *Mild*—The symptoms of the bug offend us aesthetically; a misspelled output or a misaligned printout.
2. *Moderate*—Outputs are misleading or redundant. The bug impacts the system's performance.
3. *Annoying*—The system's behavior, because of the bug, is dehumanizing. Names are truncated or arbitrarily modified. Bills for $0.00 are sent. Operators must use unnatural command sequences and must

trick the system into a proper response for unusual bug-related cases.
4. *Disturbing*—It refuses to handle legitimate transactions. The automatic teller machine won't give you money. My credit card is declared invalid.
5. *Serious*—It loses track of transactions: not just the transaction itself (your paycheck), but the fact that the transaction occurred. Accountability is lost.
6. *Very Serious*—Instead of losing your paycheck, the system credits it to another account or converts deposits into withdrawals. The bug causes the system to do the wrong transaction.
7. *Extreme*—The problems aren't limited to a few users or to a few transaction types. They are frequent and arbitrary instead of sporadic or for unusual cases.
8. *Intolerable*—Long-term, unrecoverable corruption of the data base occurs and the corruption is not easily discovered. Serious consideration is given to shutting the system down.
9. *Catastrophic*—The decision to shut down is taken out of our hands because the system fails.
10. *Infectious*—What can be worse than a failed system? One that corrupts other systems even though it does not fail in itself; that erodes the social or physical environment; that melts nuclear reactors or starts wars; whose influence, because of malfunction, is far greater than expected; a system that kills.

Any of these consequences could follow from that wrong bit. Programming is a serious business, and testing is more serious still. It pays to have nightmares about undiscovered bugs once in a while (SHED80). When was the last time one of your bugs violated someone's human rights?

2.3. Flexible Severity Rather Than Absolutes

Many programmers, testers, and quality assurance workers have an absolutist attitude toward bugs. "Everybody *knows* that a program must be *perfect* if it's to work: if there's a bug, it *must* be fixed." That's untrue, of course, even though the myth continues to be foisted onto an unwary public. Ask the person in the street and chances are that they'll parrot that myth of ours. That's trouble for us because we can't do it now and never could. It's *our* myth because we, the computer types, created it and continue to perpetuate it. Software never was perfect and won't get perfect. But is that a license to create garbage? The missing ingredient is our reluctance to quantify quality. If instead of saying that software has either 0 quality (there is at least one bug) or 100% (perfect quality and no bugs), we recognize that quality can be measured on some scale, say from 0 to 10. Quality can be measured as a combination of factors, of which the number of bugs and their severity is only one component. The details of how this is done is the subject of another book; but it's enough to say that many organizations have designed and use satisfactory, quantitative, quality metrics. Because bugs and their symptoms play a significant role in such metrics, as testing progresses you can see the quality rise from next to zero to some value at which it is deemed safe to ship the product.

Examining these metrics closer, we see that how the parts are weighted depends on environment, application, culture, and many other factors.

Let's look at a few of these:

Correction Cost—The cost of correcting a bug has almost nothing to do with symptom severity. Catastrophic, life-threatening bugs could be trivial to fix, whereas minor annoyances could require major rewrites to correct.
Context and Application Dependency—The severity of a bug, for the same bug with the same symptoms, depends on context. For example, a roundoff error in an orbit calculation doesn't mean much in a spaceship video game but it matters to real astronauts.
Creating Culture Dependency—What's important depends on the creators of the software and their cultural aspirations. Test tool vendors are more sensitive about bugs in their products than, say, games software vendors.
User Culture Dependency—What's important depends on the user culture. An R&D shop might accept a bug for which there's a workaround; a banker would go to jail for that same bug; and naive users of PC software go crazy over bugs that pros ignore.
The Software Development Phase—Severity depends on development phase. Any bug gets more

severe as it gets closer to field use and more severe the longer it's been around—more severe because of the dramatic rise in correction cost with time. Also, what's a trivial or subtle bug to the designer means little to the maintenance programmer for whom all bugs are equally mysterious.

2.4. The Nightmare List and When to Stop Testing

In George Orwell's novel, *1984*, there's a torture chamber called "room 101"—a room that contains your own special nightmare. For me, sailing through 4-foot waves, the boat heeled over, is exhilarating; for my seasick passengers, that's room 101. For me, rounding Cape Horn in winter, with 20-foot waves in a gale is a room 101 but I've heard round-the-world sailboat racers call such conditions "bracing."

The point about bugs is that you or your organization must define your own nightmares. I can't tell you what they are, and therefore I can't ascribe a severity to bugs. Which is why I treat all bugs as equally as I can in this book. And when I slip and express a value judgment about bugs, recognize it for what it is because I can't completely rid myself of my own values.

How should you go about quantifying the nightmare? Here's a workable procedure:

1. List your worst software nightmares. State them in terms of the symptoms they produce and how your user will react to those symptoms. For end users and the population at large, the categories of Section 2.2 above are a starting point. For programmers the nightmare may be closer to home, such as: "I might get a bad personal performance rating."
2. Convert the consequences of each nightmare into a cost. Usually, this is a labor cost for correcting the nightmare, but if your scope extends to the public, it could be the cost of lawsuits, lost business, or nuclear reactor meltdowns.
3. Order the list from the costliest to the cheapest and then discard the low-concern nightmares with which you can live.
4. Based on your experience, measured data (the best source to use), intuition, and published statistics postulate the kinds of bugs that are likely to create the symptoms expressed by each nightmare. Don't go too deep because most bugs are easy. This is a bug design process. If you can "design" the bug by a one-character or one statement change, then it's a good target. If it takes hours of sneaky thinking to characterize the bug, then either it's an unlikely bug or you're worried about a saboteur in your organization, which could be appropriate in some cases. Most bugs are simple goofs once you find and understand them.
5. For each nightmare, then, you've developed a list of possible causative bugs. Order that list by decreasing probability. Judge the probability based on your own bug statistics, intuition, experience, etc. The same bug type will appear in different nightmares. The importance of a bug type is calculated by multiplying the expected cost of the nightmare by the probability of the bug and summing across all nightmares:

$$\text{importance of bug type } i = \sum_{\text{all nightmares}} C_{ij} P_{(\text{bug type } i \text{ in nightmare } j)}$$

6. Rank the bug types in order of decreasing importance to you.
7. Design tests (based on your knowledge of test techniques) and design your quality assurance inspection process by using the methods that are most effective against the most important bugs.
8. If a test is passed, then some nightmares or parts of them go away. If a test is failed, then a nightmare is possible, but upon correcting the bug, it too goes away. Testing, then, gives you information you can use to revise your estimated nightmare probabilities. As you test, revise the probabilities and reorder the nightmare list. Taking whatever information you get from testing and working it back through the exercise leads you to revise your subsequent test strategy, either on this project if it's big enough or long enough, or on subsequent projects.
9. Stop testing when the probability of all nightmares has been shown to be inconsequential as a result of hard evidence produced by testing.

The above prescription can be implemented as a formal part of the software development process, or it can be adopted as a guideline or philosophical point of view. The idea is not that you implement elaborate metrics (unless that's appropriate) but that you recognize the importance of the feedback that testing provides to the testing process itself and, more important, to the kinds of tests you will design.

The mature tester's problem has never been how to design tests. If you understand testing techniques, you will know how to design several different infinities of justifiable tests. The tester's central problem is how to best cull a reasonable, finite, number of tests from that multifold infinity—a test suite that, as experience and logic leads us to predict, will have a high probability of putting the nightmares to rest—that is to say, an effective, revealing, set of tests. Look at the pesticide paradox again and observe the following consequence:

Corollary to the First Law—Test suites wear out.

Yesterday's elegant, revealing, effective, test suite will wear out because programmers and designers, given feedback on their bugs, do modify their programming habits and style in an attempt to reduce the incidence of bugs they know about. Furthermore, the better the feedback, the better the QA, the more responsive the programmers are, the faster those suites wear out. Yes, the software is getting better, but that only allows you to approach closer to, or to leap over, the previous complexity barrier. True, bug statistics tell you nothing about the coming release, only the bugs of the previous release—but that's better than basing your test technique strategy on general industry statistics or on myths. If you don't gather bug statistics, organized into some rational taxonomy, you don't know how effective your testing has been, and worse, you don't know how worn out your test suite is. The consequences of that ignorance is a brutal shock. How many horror stories do you want to hear about the sophisticated outfit that tested long, hard, and diligently—sent release 3.4 to the field, confident that it was the best tested product they had ever shipped—only to have it bomb more miserably than any prior release?

3. A TAXONOMY FOR BUGS*

*I'm sticking with "bug" rather than adopt another word such as "fault," which is the current fad in publications because: (1) everybody knows what "bug" means; (2) the standards are inconsistent with one another and with themselves in the definition of "fault," "error," and "failure"; (3) according to the Oxford English Dictionary, the usage of "bug" the way we use it, contrary to popular belief, *predates* its entomological use by centuries—the first written reference to "bug" = "goblin" is from 1388, but its first use to mean a small, six-legged creature with a hard carapace dates from 1642; (4) I prefer short, strong, Anglo-Saxon words to effete Norman words. The genesis of "bug" as a computer problem being derived from a moth fried on the power bus of an early computer, thus bringing the system down, is apocryphal. "Bug" is an ancient and honorable word (Welsh *bwg*) and not newly coined jargon peculiar to the computer industry.

3.1. General

There is no universally correct way to categorize bugs. This taxonomy is not rigid. Bugs are difficult to categorize. A given bug can be put into one or another category depending on its history and the programmer's state of mind. For example, a one-character error in a source statement changes the statement, but unfortunately it passes syntax checking. As a result, data are corrupted in an area far removed from the actual bug. That in turn leads to an improperly executed function. Is this a typewriting error, a coding error, a data error, or a functional error? If the bug is in our own program, we're tempted to blame it on typewriting;** if in another programmer's code, on carelessness. And if our job is to critique the system, we might say that the fault is an inadequate internal data-validation mechanism. A detailed taxonomy is presented in the appendix. The major categories are: requirements, features and functionality, structure, data, implementation and coding, integration, system and software architecture, and testing. A first breakdown is provided in Table 2. 1, whereas in the appendix the breakdown is as fine as makes sense. Bug taxonomy, as testing, is potentially infinite. More important than adopting the "right" taxonomy is that you adopt *some* taxonomy and that you use it as a statistical framework on which to base your testing strategy. Because there's so much effort required to

develop a taxonomy, don't redo my work—you're invited to adopt the taxonomy of the appendix (or any part thereof) and are hereby authorized to copy it (with appropriate attribution) without guilt or fear of being sued by me for plagiarism. If my taxonomy doesn't turn you on, adopt the IEEE taxonomy (IEEE87B).

** Ah, for the good old days when programs were keypunched onto cards by a keypunch operator; then we could blame the operator for the bugs. Today, with code entered directly by the programmer at a terminal or PC, we can try blaming the communication interface or the local area network, which isn't as convincing.

3.2. Requirements, Features, and Functionality Bugs

3.2.1. Requirements and Specifications

Requirements and the specifications developed from them can be incomplete, ambiguous, or self-contradictory. They can be misunderstood or impossible to understand. The specification may assume, but not mention, other specifications and prerequisites that are known to the specifier but not to the designer. And specifications that don't have these flaws may change while the design is in progress. Features are modified, added, and deleted. The designer has to hit a moving target and occasionally misses.

Requirements, especially as expressed in a specification (or often, as *not* expressed because there is no specification) are a major source of expensive bugs. The range is from a few percent to more than 50%, depending on application and environment. What hurts most about these bugs is that they're the earliest to invade the system and the last to leave. It's not unusual for a faulty requirement to get through all development testing, beta testing, and initial field use, only to be caught after hundreds of sites have been installed.

3.2.2. Feature Bugs

Specification problems usually create corresponding feature problems. A feature can be wrong, missing, or superfluous. A missing feature or case is the easiest to detect and correct. A wrong feature could have deep design implications. Extra features were once considered desirable. We now recognize that "free" features are rarely free. Any increase in generality that does not contribute to reliability, modularity, maintainability, and robustness should be suspected. Gratuitous enhancements can, if they increase complexity, accumulate into a fertile compost heap that breeds future bugs, and they can create holes that can be converted into security breaches. Conversely, one cannot rigidly forbid additional features that might be a consequence of good design. Removing the features might complicate the software, consume more resources, and foster more bugs.

3.2.3. Feature Interaction

Providing clear, correct, implementable, and testable feature specifications is not enough. Features usually come in groups of related features. The features of each group and the interaction of features within each group are usually well tested. The problem is unpredictable interactions between feature groups or even between individual features. For example, your telephone is provided with call holding and call forwarding. Call holding allows you to put a new incoming call on hold while you continue talking to the first caller. Call forwarding allows you to redirect incoming calls to some other telephone number. Here are some simple feature interaction questions: How about holding a third call when there is already a call on hold? Forwarding forwarded calls (i.e., the number forwarded to is also forwarding calls)? Forwarding calls in a loop? Holding while forwarding is active? Initiating forwarding when there is a call on hold? Holding for forwarded calls when the telephone forwarded to does (doesn't) have forwarding? . . . If you think these variations are brain twisters, how about feature interactions for your income tax return, say between federal, state, and local tax laws? Every application has its peculiar set of features and a much bigger set of unspecified feature interaction potentials and therefore feature interaction bugs. We have very little statistics on these bugs, but the trend seems to be that as the earlier, simpler, bugs are removed, feature interaction bugs emerge as a major category. Other than deliberately preventing some interactions and testing the important combinations, we have no magic remedy for these problems.

3.2.4. Specification and Feature Bug Remedies

Most feature bugs are rooted in human-to-human communication problems. One solution is the use of high-level, formal specification languages or systems (BELF76, BERZ85, DAVI88A, DAV18813, FISC79, HAYE85, PROG88, SOFT88, YEHR80). Such languages and systems provide short-term support but, in the long run, do not solve the problem.

Short-Term Support—Specification languages (we'll call them all "languages" hereafter, even though some may be interactive dialogue systems) facilitate formalization of requirements and (partial)* inconsistency and ambiguity analysis. With formal specifications, partially to fully automatic test case generation is possible. Generally, users and developers of such products have found them to be cost-effective.

---

*"Partial" rather than "complete" because total consistency and completeness analysis is a known unsolvable problem.

Long-Term Support—Assume that we have a great specification language and that it can be used to create unambiguous, complete specifications with unambiguous, complete tests and consistent test criteria. A specification written in that language could theoretically be compiled into object code (ignoring efficiency and practicality issues). But this is just programming in HOL squared. The specification problem has been shifted to a higher level but not eliminated. Theoretical considerations aside, given a system which can generate functional tests from specifications, the likeliest impact is a further complexity escalation facilitated by the reduction of another class of bugs (the complexity barrier law).
The long-term impact of formal specification languages and systems will probably be that they will influence the design of ordinary programming languages so that more of *current* specification can be formalized. This approach will reduce, but not eliminate, specification bugs. The pesticide paradox will work again to eliminate the kinds of specification bugs we now have (simple ambiguities and contradictions), leaving us a residue of tougher specification bugs that will need an even higher order specification system to expose.

3.2.5. Testing Techniques

Most **functional test techniques**—that is, those techniques which are based on a behavioral description of software, such as **transaction flow testing** (Chapter 4), **syntax testing** (Chapter 9), **domain testing** (Chapter 6), **logic testing** (Chapter 10), and **state testing** (Chapter 11) are useful in testing functional bugs. They are also useful in testing for requirements and specification bugs to the extent that the requirements can be expressed in terms of the model on which the technique is based.

3.3. Structural Bugs

3.3.1. Control and Sequence Bugs

Control and sequence bugs include paths left out, unreachable code, improper nesting of loops, loop-back or loop-termination criteria incorrect, missing process steps, duplicated processing, unnecessary processing, rampaging GOTO's, ill-conceived switches, **spaghetti code**, and worst of all, **pachinko code**.

Although much of testing and software design literature focuses on control flow bugs, they are not as common in new software as the literature might lead one to believe. One reason for the popularity of control-flow problems in the literature is that this area is amenable to theoretical treatment. Fortunately, most control-flow bugs (in new code) are easily tested and caught in unit testing.

Another source of confusion and therefore research concern is that novice programmers working on toy problems do tend to have more control-flow bugs than experienced programmers. A third reason for concern with control-flow problems is that dirty old code, especially assembly language and COBOL code, can be dominated by control-flow bugs. In fact, a good reason to rewrite an application from scratch is that the old control structure has become so complicated and so arbitrary after decades of rework that no one dare modify it further and, further, it defies testing.

Control and sequence bugs at all levels are caught by testing, especially structural testing, more specifically, path testing (Chapter 3), combined with a bottom-line functional test based on a specification. These bugs are partially prevented by language choice (e.g., languages that restrict control-flow options) and style, and most important, lots of memory. Experience shows that many control-flow problems result directly from trying to "squeeze" 8 pounds of software into a 4-pound bag (i.e., 8K object into 4K). Squeezing for short execution time is as bad.

3.3.2. Logic Bugs

Bugs in logic, especially those related to misunderstanding how case statements and logic operators behave singly and in combinations, include nonexistent cases, improper layout of cases, "impossible" cases that are not impossible, a "don't-care" case that matters, improper negation of a boolean expression (for example, using "greater than" as the negation of "less than"), improper simplification and combination of cases, overlap of exclusive cases, confusing "exclusive OR" with "inclusive OR."

Another problematic area concerns misunderstanding the semantics of the order in which a boolean expression is evaluated for specific compilers, especially in the context of deeply nested IF-THEN-ELSE constructs. For example, the truth or falsity of a logical expression is determined after evaluating a few terms, so evaluation of further terms (usually) stops, but the programmer expects that further terms will be evaluated. In other words, although the boolean expression appears as a single statement, the programmer does not understand that its components will be evaluated sequentially. See index entries on **predicate coverage** for more information.

If these bugs are part of logical (i.e., boolean) processing not related to control flow, then they are categorized as processing bugs. If they are part of a logical expression (i.e., **control-flow predicate**) which is used to direct the control flow, then they are categorized as control-flow bugs.

Logic bugs are not really different in kind from arithmetic bugs. They are likelier than arithmetic bugs because programmers, like most people, have less formal training in logic at an early age than they do in arithmetic. The best defense against this kind of bug is a systematic analysis of cases. Logic-based testing (Chapter 10) is helpful.

3.3.3. Processing Bugs

Processing bugs include arithmetic bugs, algebraic, mathematical function evaluation, algorithm selection, and general processing. Many problems in this area are related to incorrect conversion from one data representation to another. This is especially true in assembly language programming. Other problems include ignoring overflow, ignoring the difference between positive and negative zero, improper use of greater-than, greater-than-or-equal, less-than, less-than-or-equal, assumption of equality to zero in floating point, and improper comparison between different formats as in ASCII to binary or integer to floating point.

Although these bugs are frequent (12%), they tend to be caught in good unit testing and also tend to have localized effects. Selection of covering test cases, especially domain-testing methods (Chapter 6) are the testing remedies for this kind of bug.

3.3.4. Initialization Bugs

Initialization bugs are common, and experienced programmers and testers know they must look for them. Both improper and superfluous initialization occur. The latter tends to be less harmful but can affect performance. Typical bugs are as follows: forgetting to initialize working space, registers, or data areas before first use or assuming that they are initialized elsewhere; a bug in the first value of a loop-control parameter; accepting an initial value without a validation check; and initializing to the wrong format, data representation, or type.

The remedies here are in the kinds of tools the programmer has. The source language also helps. Explicit declaration of all variables, as in Pascal, helps to reduce some initialization problems. Preprocessors, either built into the language or run separately, can detect some, but not all, initialization problems. The test methods of Chapter 5 are helpful for test design and for debugging initialization problems.

3.3.5. Data-Flow Bugs and Anomalies

Most initialization bugs are a special case of data-flow anomalies. A **data-flow anomaly** occurs when there is a path along which we expect to do something unreasonable with data, such as using an uninitialized variable, attempting to use a variable before it exists, modifying data and then not storing or using the result, or initializing twice without an intermediate use. Although part of data-flow anomaly detection can be done by the compiler based on information known at compile time, much can be detected only by execution and therefore is a subject for testing. It is generally recognized today that data-flow anomalies are as important as control-flow anomalies. The methods of Chapters 5 and 12 will help you design tests aimed at data-flow problems.

3.4. Data Bugs

3.4.1. General

Data bugs include all bugs that arise from the specification of data objects, their formats, the number of such objects, and their initial values. Data bugs are at least as common as bugs in code, but they are often treated as if they did not exist at all. Underestimating the frequency of data bugs is caused by poor bug accounting. In some projects, bugs in data declarations are just not counted, and for that matter, data declaration statements are not counted as part of the code. The separation of code and data is, of course, artificial because their roles can be interchanged at will. At the extreme, one can write a twenty-instruction program that can simulate any computer (a Turing machine) and have all “programs” recorded as data and manipulated as such. Furthermore, this can be done in any language on any computer—but who would want to?

Software is evolving toward programs in which more and more of the control and processing functions are stored in tables. I call this the third law:

Third Law—Code migrates to data.

Because of this law there is an increasing awareness that bugs in code are only half the battle and that data problems should be given equal attention. The bug statistics of Table 2.1 support this concept; that is, structural bugs and data bugs each have frequencies of about 25%. If you examine a piece of contemporary source code, you may find that half of the statements are data declarations. Although these statements do not result in executable code, because they are specified by humans, they are as subject to error as operative statements. If a program is designed under the assumption that a certain data object will be set to zero and it isn’t, the operative statements of the program are not at fault. Even so, there is still an initialization bug, which, because it is in a data statement, could be harder to find than if it had been a bug in executable code.

This increase in the proportion of the source statements devoted to data definition is a direct consequence of two factors: (1) the dramatic reduction in the cost of main memory and disc storage, and (2) the high

cost of creating and testing software. Generalized software controlled by tables is not efficient. Computer costs, especially memory costs, have decreased to the point where the inefficiencies of generalized table-driven code are not usually significant. The increasing cost of software as a percentage of system cost has shifted the emphasis in the software industry away from single-purpose, unique software to an increased reliance on prepackaged, generalized programs. This trend is evident in the computer manufacturers' software, in the existence of a healthy proprietary software industry, and in the emergence of languages and programming environments that support code reusability (e.g., object-oriented languages). Generalized packages must satisfy a wide range of options, host configurations, operating systems, and computers. The designer of a generalized package achieves generality, in part, by making many things parametric, such as array sizes, memory partition, and file structure. It is not unusual for a big application package to have several hundred parameters. Setting the parameter values particularizes the program to the specific installation. The parameters are interrelated, and errors in those relations can cause illogical conditions and, therefore, bugs.

Another source of database complexity increase is the use of control tables in lieu of code. The simplest example is the use of tables that turn processing options on and off. A more complicated form of control table is used when a system must execute a set of closely related processes that have the same control structure but are different in details. An early example is found in telephony, where the details of controlling a telephone call are table-driven. A generalized call-control processor handles calls from and to different kinds of lines. The system is loaded with a set of tables that corresponds to the protocols required for that telephone exchange. Another example is the use of generalized device-control software which is particularized by data stored in device tables. The operating system can be used with new, undefined devices, if those devices' parameters can fit into a set of very broad values. The culmination of this trend is the use of complete, internal, transaction-control languages designed for the application. Instead of being coded as computer instructions or language statements, the steps required to process a transaction are stored as a sequence of constants in a transaction-processing table. The state of the transaction, that is, the current processing step, is stored in a transaction-control block. The generalized transaction-control processor uses the combination of transaction state and the control tables to direct the transaction to the next step. The transaction-control table is actually a program which is processed interpretively by the transaction-control processor. That program may contain the equivalent of addressing, conditional branch instructions, looping statements, case statements, and so on. In other words, a **hidden programming language** has been created. It is an effective design technique because it enables fixed software to handle many different transaction types, individually and simultaneously. Furthermore, modifying the control tables to install new transaction types is usually easier than making the same modifications in code.

In summary, current programming trends are leading to the increasing use of undeclared, internal, specialized programming languages. These are languages—make no mistake about that—even if they are simple compared to normal programming languages; but the syntax of these languages is rarely debugged. There's no compiler for them and therefore no source syntax checking. The programs in these languages are inserted as octal or hexadecimal codes—as if we were programming back in the early days of UNIVAC-I. Large, low-cost memory will continue to strengthen this trend and, consequently, there will be an increased incidence of code masquerading as data. Bugs in this kind of hidden code are at least as difficult to find as bugs in normal code. The first step in the avoidance of data bugs—whether the data are used as pure data, as parameters, or as hidden code—is the realization that *all* source statements, including data declarations, must be counted, and that all source statements, whether or not they result in object code, are bug-prone.

The categories used for data bugs are different from those used for code bugs. Each way of looking at data provides a different perspective. These categories for data bugs overlap and are no stricter than the categories used for bugs in code.

3.4.2. Dynamic Versus Static

Dynamic data are transitory. Whatever their purpose, they have a relatively short lifetime, typically the processing time of one transaction. A storage object may be used to hold dynamic data of different types, with different formats, attributes, and residues. Failure to initialize a shared object properly can lead to data-dependent bugs caused by residues from a previous use of that object by another transaction. Note that the culprit transaction is long gone when the bug's symptoms are discovered. Because the effect of corruption of dynamic data can be arbitrarily far removed from the cause, such bugs are among the most difficult to catch. The design remedy is complete documentation of all shared-memory structures, defensive code that does thorough data-validation checks, and centralized-resource managers.

The basic problem is leftover garbage in a shared resource. This can be handled in one of three ways: (1) cleanup after use by the user, (2) common cleanup by the resource manager, and (3) no cleanup. The latter is the method usually used. Therefore, resource users must program under the assumption that the resource manager gives them garbage-filled resources. Common cleanup is used in very secure systems where subsequent users of a resource must never be able to read data left by a previous user in another security or privacy category.

Static data are fixed in form and content. Whatever their purpose, they appear in the source code or data base, directly or indirectly, as, for example, a number, a string of characters, or a bit pattern. Static data need not be explicit in the source code. Some languages provide **compile-time processing**, which is especially useful in general-purpose routines that are particularized by interrelated parameters. Compile-time processing is an effective measure against parameter-value conflicts. Instead of relying on the programmer to calculate the correct values of interrelated parameters, a program executed at compile time (or assembly time) calculates the parameters' values. If compile-time processing is not a language feature, then a specialized preprocessor can be built that will check the parameter values and calculate those values that are derived from others. As an example, a large commercial telecommunications system has several hundred parameters that dictate the number of lines, the layout of all storage media, the hardware configuration, the characteristics of the lines, the allowable user options for those lines, and so on. These are processed by a site-adapter program that not only sets the parameter values but builds data declarations, sizes arrays, creates constants, and inserts processing routines from a library. A bug in the site adapter, or in the data given to the site adapter, can result in bugs in the static data used by the object programs for that site.

Another example is the postprocessor used to install many personal computer software packages. Here the configuration peculiarities are handled by generalized table-driven software, which is particularized at run (actually, installation) time.

Any preprocessing (or postprocessing) code, any code executed at compile or assembly time or before, at load time, at installation time, or some other time can lead to faulty static data and therefore bugs—even though such code (and the execution thereof) does not represent object code at run time. We tend to take compilers, assemblers, utilities, loaders, and configurators for granted and do not suspect them to be bug sources. This is not a bad assumption for standard utilities or translators. But if a highly parameterized system uses site-adapter software or preprocessors or compile-time/assembly-time processing, and if such processors and code are developed concurrently with the working software of the application—watch out!

Software used to produce object code is suspect until validated. All new software must be rigorously tested even if it isn't part of the application's mainstream. Static data can be just as wrong as any other kind and can have just as many bugs. Do not treat a routine that creates static data as "simple" because it "just stuffs a bunch of numbers into a table." Subject such code to the same testing rigor that you apply to running code.[*]

---

[*]And the winner for consistently bad software of this ilk is PC installation software. Clean, robust, "friendly," operational software (e.g., word processing) is saddled with a hostile but overly tender installation package

whose operation is closer to clearing mine fields than to processing.

The design remedy for the preprocessing situation is in the source language. If the language permits compile-time processing that can be used to particularize parameter values and data structures, and if the syntax of the compile-time statements is identical to the syntax of the rest of the language, then the code will be subjected to the same validation and syntax checking as ordinary code. Such language facilities eliminate the need for most specialized preprocessors, table generators, and site adapters. For postprocessors, there is no magic, other than to recognize that users judge developers by the entire picture, installation software included.

3.4.3. Information, Parameter, and Control

Static or dynamic data can serve in one of three roles, or in a combination of roles: as a parameter, for control, or for information. What constitutes control or information is a matter of perspective and can shift from one processing level to another. A scheduler receives a request to start a process. To the scheduler the identity of the process is information to be processed, but at another level it is control. My name is used to generate a hash code that will be used to access a disc record. My name is information, but to the disc hardware its translation into an address is control (e.g., move to track so-and-so).

Information is usually dynamic and tends to be local to a single transaction or task. As such, errors in information (when data are treated as information, that is) may not be serious bugs. The bug, if any, is in the lack of protective data-validation code or in the failure to protect the routine's logic from out-of-range data or data in the wrong format. The only way we can be sure that there is data-validation code in a routine is to put it there. Assuming that the other routine will validate data invites latent bugs and maintenance problems. The program evolves and changes, and it is forgotten that the modified routine did the data validation for several other routines. *If* a routine is vulnerable to bad data, the only sane thing to do is to block such data within the routine; but it's even better to redesign it so that it is no longer vulnerable.

Inadequate data validation often leads to finger pointing. The calling routine's author is blamed, the called routine's author blames back, they both blame the operators. This scenario leads to a lot of ego confrontation and guilt. "If only the other programmers did their job correctly," you say, "we wouldn't need all this redundant data validation and defensive code. I have to put in this extra junk because I'm surrounded by slobs!" This attitude is understandable, but not productive. Furthermore, if you really feel that way, you're likely to feel guilty about it. Don't blame your fellow programmer and don't feel guilt. Nature has conspired against us but given us a scapegoat. One of the unfortunate side effects of large-scale integrated circuitry stems from the use of microscopic logic elements that work at very low energy levels. Modern circuitry is vulnerable to electronic noise, electromagnetic radiation, cosmic rays, neutron hits, stray alpha particles, and other noxious disturbances. No kidding—alpha-particle hits that can change the value of a bit are a serious problem, and the semiconductor manufacturers are spending a lot of money and effort to reduce the random modification of data by alpha particles. Therefore, even if your fellow programmers did thorough, correct data validation, dynamic data, static data, parameters, and code can be corrupted. Program without rancor and guilt! Put in the data-validation checks and blame the necessity on sun spots and alpha particles![*]

[*]There are always two sides to the coin. The best routine accepts any kind of input garbage and returns with an "invalid input data" code, whereas the worst routine just crashes. How, when, and where data validation should be done, and how defensive low-level routines should be, are architecture and integration issues. Locally defensive code—that is, code that is defensive in accordance with local notions of correctness—may block or reject good transactions and therefore create integration problems. What it comes down to is that defensiveness should be (but isn't usually) a part of the routine's functional specification and should be attributed to routines in accordance with a global plan.

3.4.4. Contents, Structure, and Attributes

Data specifications consist of three parts:

> **Contents**—The actual bit pattern, character string, or number put into a data structure. Content is a pure bit pattern and has no meaning unless it is interpreted by a hardware or software processor. All data bugs result in the corruption or misinterpretation of content.
> **Structure**—The size and shape and numbers that describe the data object, that is, the memory locations used to store the content (e.g., 16 characters aligned on a word boundary, 122 blocks of 83 characters each, bits 4 through 14 of word 17). Structures can have substructures and can be arranged into superstructures. A hunk of memory may have several different structures defined over it—e.g., a two-dimensional array treated elsewhere as *N* one-dimensional arrays.
> **Attributes**—The specification of meaning, that is, the semantics associated with the contents of a data object (e.g., an integer, an alphanumeric string, a subroutine).

The severity and subtlety of bugs increases as we go from content to attributes because things get less formal in that direction. Content has been dealt with earlier in this section. Structural bugs can take the form of declaration bugs, but these are not the worst kind of structural bugs. A serious potential for bugs occurs when data are used with different structures. Here is a piece of clever design. The programmer has subdivided the problem into eight cases and uses a 3-bit field to designate the case. Another programmer has four different cases to consider and uses a 2-bit field for the purpose. A third programmer is interested in the combination of the other two sets of cases and treats the whole as a 5-bit field that leads to thirty-two combined cases. We cannot judge, out of context, whether this is a good design or an abomination, but we can note that there is a different structure in the minds of the three programmers and therefore a potential for bugs. The practice of interpreting a given memory location under several different structures is not intrinsically bad. Often, the only alternative would be increased memory and many more data transfers.

Attributes of data are the meanings we associate with data. Although some bugs are related to misinterpretation of integers for floating point and other basic representation problems, the more subtle attribute-related bugs are embedded in the application. Consider a 16-bit field. It could represent, among other things, a number, a loop-iteration count, a control code, a pointer, or a link field. Each interpretation is a different attribute. There is no way for the computer to know that it is proper or improper to add a control code to a link field to yield a loop count. We have used the same data with different meanings. In modern parlance, we have changed the **data type**. It is generally incorrect to logically or arithmetically combine objects whose types are different. Conversely, it is almost impossible to create an efficient system without doing so. Shifts in interpretation usually occur at interfaces, especially the human interface that is behind every software interface. See GANN76 for a summary of **type bugs**.

The preventive measures for data-type bugs are in the source language, documentation, and coding style. Explicit documentation of the contents, structure, and attributes of all data objects is essential. The database documentation should be centralized. All alternate interpretation of a given data object should be listed along with the identity of all routines that have access to that object. A proper **data dictionary** (which is what the database documentation is called) can be as large as the narrative description of the code. The data dictionary and the database it represents must also be designed. This design is done by a high-level design process, which is as important as the design of the software architecture. My point of view here is dogmatic. Routines should not be administratively treated as if they have their "own" data declarations.[*] All data structures should be globally defined and centrally administered. Exceptions, such as a private work area, should be individually justified. Such private data structures must never be used by any other routine but the structure must still be documented in the data dictionary.

---

[*]I've had more flack and misinterpretations of this position than almost anything else I've written. It seems to

> fly in the face of contemporary programming trends and language advances. To support the reasoning, I'll cite what I call the Fourth Law:
> *Fourth Law*—Local migrates to global.
> There is a tendency, over a period of time (years), for previously local data objects to become global because there are intermediate (local) results that some other programmer can use to "fix" a maintenance problem or to add new functionality. Although the originating programmer may have intended object X to be local, own data, 20 years later enough of these have been migrated to a more global scope to cause serious problems. For this reason all data objects should be administratively treated as if they were global from the very start. The issue is centralized administration of data structures and data. Where the language requires data declarations within the body of the routine the desired effect can be achieved by use of macros, by preprocessors, etc. The point is that even if the language permits or requires private data objects, administratively they should be treated as if they are global—i.e., in the data dictionary. An alternate, and possibly concurrent position, is to enforce locality rigidly and permanently. That means making the global use of local data as heinous a crime as say, modifying the computer's backplane wiring. This is part of what's behind object-oriented programming. We'll have to wait for a decade, preferably two, to see if locality is retained over time and if the Fourth Law does or does not hold (in practice, rather than in theory) for object-oriented software.

It's impossible to properly test software of any size (say 10,000+ statements) without central database management and a configuration-controlled data dictionary. I was once faced with such a herculean challenge. My first step was to try to create the missing data dictionary preparatory to any attempt to define tests. The act of dragging the murky bottoms of a hundred minds for hidden data declarations and semiprivate space in an attempt to create a data dictionary revealed so many data bugs that it was obvious that the system would defy integration. I never did get to design tests for that project—it collapsed; and a new design was started surreptitiously from scratch.

The second remedy is in the source language. **Strongly typed languages** prevent the inadvertent mixed manipulation of data that are declared as different types. A conversion in usage from pointer type to counter type, say, requires an explicit statement that will do the conversion. Such statements may or may not result in object code. Conversion from floating point to integer, would, of course, require object code, but conversion from pointer to counter might not. Strong typing forces the explicit declaration of attributes and provides compiler facilities to check for mixed-type operations. The ability of the user to specify types, as in Pascal, is mandatory. These data-typing facilities force the specification of data attributes into the source code, which makes them more amenable to automatic verification by the compiler and to test design than when the attributes are described in a separate data dictionary. In assembly language programming, or in source languages that do not have user-defined types, the remedy is the use of **field-access macros**. No programmer is allowed to directly access a field in the database. Access can be obtained only through the use of a field-access macro. The macro code does all the extraction, stripping, justification, and type conversion necessary. If the database structure has to be changed, the affected field-access macros are changed, but the source code that uses the macros does not (usually) have to be changed. The attributes of the data are documented with the field-access macro documentation. Another advantage of this approach is that the data dictionary can be automatically produced from the specifications of the field-access macro library.

3.5. Coding Bugs

Coding errors of all kinds can create any of the other kinds of bugs. Syntax errors are generally not important in the scheme of things if the source language translator has adequate syntax checking. Failure to catch a syntax error is a bug in the translator. A good translator will also catch undeclared data, undeclared routines, dangling code, and many initialization problems. Any programming error caught by the translator (assembler, compiler, or interpreter) does not substantially affect test design and execution because testing cannot start until such errors are corrected. Whether it takes a programmer one, ten, or a hundred passes before a routine can be tested should concern software management (because it is a programming productivity issue) but not test design (which is a quality-assurance issue). But if a program has many source-syntax errors, we should expect many logic and coding bugs—because a slob is a slob is a slob.

Given good source-syntax checking, the most common pure coding errors are typographical, followed by errors caused by not understanding the operation of an instruction or statement or the by-products of an instruction or statement. Coding bugs are the wild cards of programming. Unlike logic or process bugs, which have their own perverse rationality, wild cards are arbitrary.

The most common kind of coding bug, and often considered the least harmful, are documentation bugs (i.e., erroneous comments). Although many documentation bugs are simple spelling errors or the result of poor writing, many are actual errors—that is, misleading or erroneous comments. We can no longer afford to discount such bugs because their consequences are as great as "true" coding errors. Today, programming labor is dominated by maintenance. This will increase as software becomes even longer-lived. Documentation bugs lead to incorrect maintenance actions and therefore cause the insertion of other bugs. Testing techniques have nothing to offer for these bugs. The solution lies in inspections, QA, automated data dictionaries, and specification systems.

3.6. Interface, Integration, and System Bugs

3.6.1. External Interfaces

The external interfaces are the means used to communicate with the world. These include devices, actuators, sensors, input terminals, printers, and communication lines. Often there is a person on the other side of the interface. That person may be ingenious or ingenuous, but is frequently malevolent. The primary design criterion for an interface with the outside world should be **robustness**. All external interfaces, human or machine, employ a protocol. Protocols are complicated and hard to understand. The protocol itself may be wrong, especially if it's new, or it may be incorrectly implemented. Other external interface bugs include: invalid timing or sequence assumptions related to external signals; misunderstanding external input and output formats; and insufficient tolerance to bad input data. The test design methods of Chapters 6, 9, and 11 are suited to testing external interfaces.

3.6.2. Internal Interfaces

Internal interfaces are in principle not different from external interfaces, but there are differences in practice because the internal environment is more controlled. The external environment is fixed and the system must adapt to it but the internal environment, which consists of interfaces with other components, can be negotiated. Internal interfaces have the same problems external interfaces have, as well as a few more that are more closely related to implementation details: protocol-design bugs, input and output format bugs, inadequate protection against corrupted data, wrong subroutine call sequence, call-parameter bugs, misunderstood entry or exit parameter values.

To the extent that internal interfaces, protocols, and formats are formalized, the test methods of Chapters 6, 9, and 11 will be helpful. The real remedy is in the design and in standards. Internal interfaces should be standardized and not just allowed to grow. They should be formal, and there should be as few as possible. There's a trade-off between the number of different internal interfaces and the complexity of the interfaces. One universal interface would have so many parameters that it would be inefficient and subject to abuse, misuse, and misunderstanding. Unique interfaces for every pair of communicating routines would be efficient, but N programmers could lead to N^2 interfaces, most of which wouldn't be documented and all of which would have to be tested (but wouldn't be). The main objective of integration testing is to test all internal interfaces (BEIZ84).

3.6.3. Hardware Architecture

It's easy to forget that hardware exists. You can have a programming career and never see a mainframe or minicomputer. When you are working through successive layers of application executive, operating system, compiler, and other intervening software, it's understandable that the hardware architecture appears abstract and remote. It is neither practical nor economical for every programmer in a large

project to know all aspects of the hardware architecture. Software bugs related to hardware architecture originate mostly from misunderstanding how the hardware works. Here are examples: paging mechanism ignored or misunderstood, address-generation error, I/O-device operation or instruction error, I/O-device address error, misunderstood device-status code, improper hardware simultaneity assumption, hardware race condition ignored, data format wrong for device, wrong format expected, device protocol error, device instruction-sequence limitation ignored, expecting the device to respond too quickly, waiting too long for a response, ignoring channel throughput limits, assuming that the device is initialized, assuming that the device is not initialized, incorrect interrupt handling, ignoring hardware fault or error conditions, ignoring operator malice.

The remedy for hardware architecture and interface problems is two-fold: (1) good programming and testing and (2) centralization of hardware interface software in programs written by hardware interface specialists. Hardware interface testing is complicated by the fact that modern hardware has very few buttons, switches, and lights. Old computers had lots of them, and you could abuse those buttons and switches to create wonderful anomalous interface conditions that could not be simulated any other way. Today's highly integrated black boxes rarely have such controls and, consequently, considerable ingenuity may be needed to simulate and test hardware interface status conditions. Modern hardware is better and cheaper without the buttons and lights, but also harder to test. This paradox can be resolved by hardware that has special test modes and test instructions that do what the buttons and switches used to do. The hardware manufacturers, as a group, have yet to provide adequate features of this kind. Often the only alternative is to use an elaborate hardware simulator instead of the real hardware. Then you're faced with the problem of distinguishing between real bugs and hardware simulator implementation bugs.

3.6.4. Operating System

Program bugs related to the operating system are a combination of hardware architecture and interface bugs, mostly caused by a misunderstanding of what it is the operating system does. And, of course, the operating system could have bugs of its own. Operating systems can lull the programmer into believing that all hardware interface issues are handled by it. Furthermore, as the operating system matures, bugs in it are found and corrected, but some of these corrections may leave quirks. Sometimes the bug is not fixed at all, but a notice of the problem is buried somewhere in the documentation—if only you knew where to look for it.

The remedy for operating system interface bugs is the same as for hardware bugs: use operating system interface specialists, and use explicit interface modules or macros for all operating system calls. This approach may not eliminate the bugs, but at least it will localize them and make testing easier.

3.6.5. Software Architecture

Software architecture bugs are often the kind that are called "interactive." Routines can pass unit and integration testing without revealing such bugs. Many of them depend on load, and their symptoms emerge only when the system is stressed. They tend to be the most difficult kind of bug to find and exhume. Here is a sample of the causes of such bugs: assumption that there will be no interrupts, failure to block or unblock interrupts, assumption that code is reentrant or not reentrant, bypassing data interlocks, failure to close or open an interlock, assumption that a called routine is resident or not resident, assumption that a calling program is resident or not resident, assumption that registers or memory were initialized or not initialized, assumption that register or memory location content did not change, local setting of global parameters, and global setting of local parameters.

The first line of defense against these bugs is the design. The first bastion of that defense is that there *be* a design for the software architecture. Not designing a software architecture is an unfortunate but common disease. The most elegant test techniques will be helpless in a complicated system whose architecture "just growed" without plan or structure. All test techniques are applicable to the discovery of software architecture bugs, but experience has shown that careful integration of modules and subjecting

the final system to a brutal stress test are especially effective (BEIZ84).*

*Until the stress test wears out, that is.

3.6.6. Control and Sequence Bugs

System-level control and sequence bugs include: ignored timing; assuming that events occur in a specified sequence; starting a process before its prerequisites are met (e.g., working on data before all the data have arrived from disc); waiting for an impossible combination of prerequisites; not recognizing when prerequisites have been met; specifying wrong priority, program state, or processing level; missing, wrong, redundant, or superfluous process steps.

The remedy for these bugs is in the design. Highly structured sequence control is helpful. Specialized, internal, sequence-control mechanisms, such as an internal job control language, are useful. Sequence steps and prerequisites stored in tables and processed interpretively by a sequence-control processor or dispatcher make process sequences easier to test and to modify if bugs are discovered. **Path testing** as applied to **transaction flowgraphs**, as discussed in Chapter 4, is especially effective at detecting system-level control and sequence bugs.

3.6.7. Resource Management Problems

Memory is subdivided into dynamically allocated resources such as buffer blocks, queue blocks, task control blocks, and overlay buffers. Similarly, external mass storage units such as discs, are subdivided into memory-resource pools. Here are some resource usage and management bugs: required resource not obtained (rare); wrong resource used (common, if there are several resources with the same structure or different kinds of resources in the same pool); resource already in use; race condition in getting a resource; resource not returned to the right pool; fractionated resources not properly recombined (some resource managers take big resources and subdivide them into smaller resources, and Humpty Dumpty isn't always put together again); failure to return a resource (common); **resource deadlock** (a type A resource is needed to get a type B, a type B is needed to get a type C, and a type C is needed to get a type A); resource use forbidden to the caller; used resource not returned; resource linked to the wrong kind of queue; forgetting to return a resource.

A design remedy that prevents bugs is always preferable to a test method that discovers them. The design remedy in resource management is to keep the resource structure simple: the fewest different kinds of resources, the fewest pools, and no private resource management.

Complicated resource structures are often designed in a misguided attempt to save memory and not because they're essential. The software has to handle, say, large-, small-, and medium-length transactions, and it is reasoned that memory will be saved if three different-sized resources are implemented. This reasoning is often faulty because:

1. Memory is cheap and getting cheaper.
2. Complicated resource structures and multiple pools need management software; that software needs memory, and the increase in program space could be bigger than the expected data space saved.
3. The complicated scheme takes additional processing time, and therefore all resources are held in use a little longer. The size of the pools will have to be increased to compensate for this additional holding time.
4. The basis for sizing the resource is often wrong. A typical choice is to make the buffer block's length equal to the length required by an average transaction—usually a poor choice. A correct analysis (see BEIZ78, pp. 301-302) shows that the optimum resource size is usually proportional

to the square root of the transaction's length. However, square-root laws are relatively insensitive to parameter changes and consequently the waste of using many short blocks for long transactions or large blocks to store short transactions isn't as bad as naive intuition suggests.

The second design remedy is to centralize the management of all pools, either through centralized resource managers, common resource-management subroutines, resource-management macros, or a combination of these.

I mentioned resource loss three times—it was not a writing bug. Resource loss is the most frequent resource-related bug. Common sense tells you why programmers lose resources. You need the resource to process—so it's unlikely that you'll forget to get it; but when the job is done, the successful conclusion of the task will not be affected if the resource is not returned. A good routine attempts to get resources as soon as possible at a common point and also attempts to return them at a common point; but strange paths may require more resources, and you could forget that you're using several resource units instead of one. Furthermore, an exception-condition handler that responds to system-threatening illogical conditions may bypass the normal exit and jump directly to an executive level—and there goes the resource. The design remedies are to centralize resource fetch-and-return within each routine and to provide macros that return all resources rather than just one. Resource-loss problems are exhumed by path testing (Chapter 3), by transaction-flow testing (Chapter 4), data-flow testing (Chapter 5), and by stress testing (BEIZ84).

3.6.8. Integration Bugs

Integration bugs are bugs having to do with the integration of, and with the interfaces between, presumably working and tested components. Most of these bugs result from inconsistencies or incompatibilities between components. All methods used to transfer data directly or indirectly between components and all methods by which components share data can host integration bugs and are therefore proper targets for integration testing. The communication methods include data structures, call sequences, registers, semaphores, communication links, protocols, and so on. Integration strategies and special testing considerations are discussed in more detail in BEIZ84. While integration bugs do not constitute a big bug category (9%) they are an expensive category because they are usually caught late in the game and because they force changes in several components and/or data structures, often during the height of system debugging. Test methods aimed at interfaces, especially domain testing (Chapter 6), syntax testing (Chapter 9), and data-flow testing when applied across components (Chapter 5), are effective contributors to the discovery and elimination of integration bugs.

3.6.9. System Bugs

System bugs is a catch-all phrase covering all kinds of bugs that cannot be ascribed to components or to their simple interactions, but result from the totality of interactions between many components such as programs, data, hardware, and the operating system. System testing as a discipline is discussed in BEIZ84. The only test technique that applies obviously and directly to system testing is transaction-flow testing (Chapter 4); but the reader should keep in mind two important facts: (1) all test techniques can be useful at all levels, from unit to system, and (2) there can be no meaningful system testing until there has been thorough component and integration testing. System bugs are infrequent (1.7%) but very important (expensive) because they are often found only after the system has been fielded and because the fix is rarely simple.

3.7. Test and Test Design Bugs

3.7.1. Testing

Testers have no immunity to bugs (see the footnote on page 20). Tests, especially system tests, require complicated scenarios and databases. They require code or the equivalent to execute, and consequently

they can have bugs. The virtue of independent functional testing is that it provides an unbiased point of view; but that lack of bias is an opportunity for different, and possibly incorrect, interpretations of the specification. Although test bugs are not software bugs, it's hard to tell them apart, and much labor can be spent making the distinction. Also, consider the maintenance programmer—does it matter whether she's worked 3 days to chase and fix a real bug or wasted 3 days chasing a chimerical bug that was really a faulty test specification?

3.7.2. Test Criteria

The specification is correct, it is correctly interpreted and implemented, and a seemingly proper test has been designed; but the criterion by which the software's behavior is judged is incorrect or impossible. How would you, for example, "prove that the entire system is free of bugs?" If a criterion is quantitative, such as a throughput or processing delay, the act of measuring the performance can perturb the performance measured. The more complicated the criteria, the likelier they are to have bugs.

3.7.3. Remedies

The remedies for test bugs are: test debugging, test quality assurance, test execution automation, and test design automation.

Test Debugging—The first remedy for test bugs is testing and debugging the tests. The differences between test debugging and program debugging are not fundamental. Test debugging is usually easier because tests, when properly designed, are simpler than programs and do not have to make concessions to efficiency. Also, tests tend to have a localized impact relative to other tests, and therefore the complicated interactions that usually plague software designers are less frequent. We have no magic prescriptions for test debugging—no more than we have for software debugging.

Test Quality Assurance—Programmers have the right to ask how quality in independent testing and test design is monitored. Should we implement test testers and test—tester tests? This sequence does not converge. Methods for test quality assurance are discussed in *Software System Testing and Quality Assurance* (BEIZ84).

Test Execution Automation—The history of software bug removal and prevention is indistinguishable from the history of programming automation aids. Assemblers, loaders, compilers, and the like were all developed to reduce the incidence of programmer and/or operator errors. Test execution bugs are virtually eliminated by various test execution automation tools, many of which are discussed throughout this book. The point is that "manual testing" is self-contradictory. If you want to get rid of test execution bugs, get rid of manual execution.

Test Design Automation—Just as much of software development has been automated (what is a compiler, after all?) much test design can be and has been automated. For a given productivity rate, automation reduces bug count—be it for software or be it for tests.

3.8. Testing and Design Style

This is a book on test design, yet this chapter has said a lot about programming style and design. You might wonder why the productivity of one programming group is as much as 10 times higher than that of another group working on the same application, the same computer, in the same language, and under similar constraints. It should be obvious—bad designs lead to bugs, and bad designs are difficult to test; therefore, the bugs remain. Good designs inhibit bugs before they occur and are easy to test. The two factors are multiplicative, which explains the large productivity differences. The best test techniques are useless when applied to abominable code: it is sometimes easier to redesign a bad routine than to attempt to create tests for it. The labor required to produce new code plus the test design and execution labor for the new code can be much less than the labor required to design thorough tests for an undisciplined, unstructured monstrosity. Good testing works best on good code and good designs. And no test technique can ever convert garbage into gold.

4. SOME BUG STATISTICS

The frequency of bugs taken from many different sources (see the appendix) shows approximately 2.4 bugs per thousand source statements.* But because some of the sample did not include unexecutable statements, the real value is probably lower. These data (usually) describe bugs caught in independent testing, integration testing, and system testing. The number of bugs discovered by the programmer in self-testing at the component level is unknown. The importance of Table 2.1 is not the absolute frequency of bugs (e.g., how many bugs per thousand lines of code) but the relative frequency of the bugs by category. You should examine the sources for these statistics yourself so that you can rearrange the categories to match your own taxonomy. Other references with useful statistics are: AKIY71, BELF79, BOEH75A, BOIE72, DNIE78, ELSH76B, ENDR75, GANN76, GILB77, GOEL78B, HAUG64, HOFF77, ITOH73, LITE76, REIF79A, RUBE75, SCH178, SCHN75, SCHN79A, SCHW71, SHOO75, and WAGO73.

*This is not a contradiction to the 1%–3% rate stated on page 2 in Chapter 1. The 1%–3% rate applies to all bugs, from unit testing out to the field, and typically includes all the bugs discovered by the programmer during self-testing and inspections. The 0.24% rate is dominated by our data source, which included mostly independent testing, integration testing, and system testing—after thorough component testing by the programmers. But even that shouldn't be taken seriously because there's a lot of variation in the source on that score also. What counts in Table 2.1 is the relative frequency of various bug types as a guide to selecting effective testing strategies, and not the absolute bug rate to be used as club on the programmer's head.

SIZE OF SAMPLE—6,877,000 STATEMENTS (COMMENTS INCLUDED)
TOTAL REPORTED BUGS—16,209—BUGS PER 1000 STATEMENTS—2.36

1xxx REQUIREMENTS	1317	
11xx Requirements Incorrect	649	
12xx Requirements Logic	153	
13xx Requirements, Completeness	224	
15xx Presentation, Documentation	13	
16xx Requirements Changes	278	
2xxx FEATURES AND FUNCTIONALITY	2624	1
21xx Feature/Function Correctness	456	
22xx Feature Completeness	231	
23xx Functional Case Completeness	193	
24xx Domain Bugs	778	
25xx User Messages and Diagnostics	857	
26xx Exception Condition Mishandled	79	
29xx Other Functional Bugs	30	
3xxx STRUCTURAL BUGS	4082	2
31xx Control Flow and Sequencing	2078	1
32xx Processing	2004	1
4xxx DATA	3638	2
41xx Data Definition and Structure	1805	1
42xx Data Access and Handling	1831	1
49xx Other Data Problems	2	

5xxx IMPLEMENTATION AND CODING	1601
51xx Coding and Typographical	322
52xx Style and Standards Violations	318
53xx Documentation	960
59xx Other Implementation	1
6xxx INTEGRATION	1455
61xx Internal Interfaces	859
62xx External Interfaces, Timing, Throughput	518
69xx Other Integration	78
7xxx SYSTEM, SOFTWARE ARCHITECTURE	282
71xx O/S Call and Use	47
72xx Software Architecture	139
73xx Recovery and Accountability	4
74xx Performance	64
75xx Incorrect Diagnostics, Exceptions	16
76xx Partitions, Overlays	3
77xx Sysgen, Environment	9
8xxx TEST DEFINITION AND EXECUTION	447
81xx Test Design Bugs	11
82xx Test Execution Bugs	355
83xx Test Documentation	11
84xx Test Case Completeness	64
89xx Other Testing Bugs	6
9xxx OTHER, UNSPECIFIED	763

Table 2.1. Sample Bug Statistics.

5. SUMMARY

1. The importance of a bug depends on its frequency, the correction cost, the consequential cost, and the application. Allocate your limited testing resources in proportion to the bug's importance.
2. Use the nightmare list as a guide to how much testing is required.
3. Test techniques are like antibiotics—their effectiveness depends on the target—what works against a virus may not work against bacteria or fungi. The test techniques you use must be matched to the kind of bugs you have.
4. Because programmers learn from their mistakes, the effectiveness of test techniques, just as antibiotics, erodes with time. *TEST SUITES WEAR OUT.*
5. A comprehensive bug taxonomy is a prerequisite to gathering useful bug statistics. Adopt a taxonomy, simple or elaborate, but adopt one and classify all bugs within it.
6. Continually gather bug statistics (in accordance to your taxonomy) to determine the kind of bugs you have and the effectiveness of your current test techniques against them.

<ch1 toc ch3>