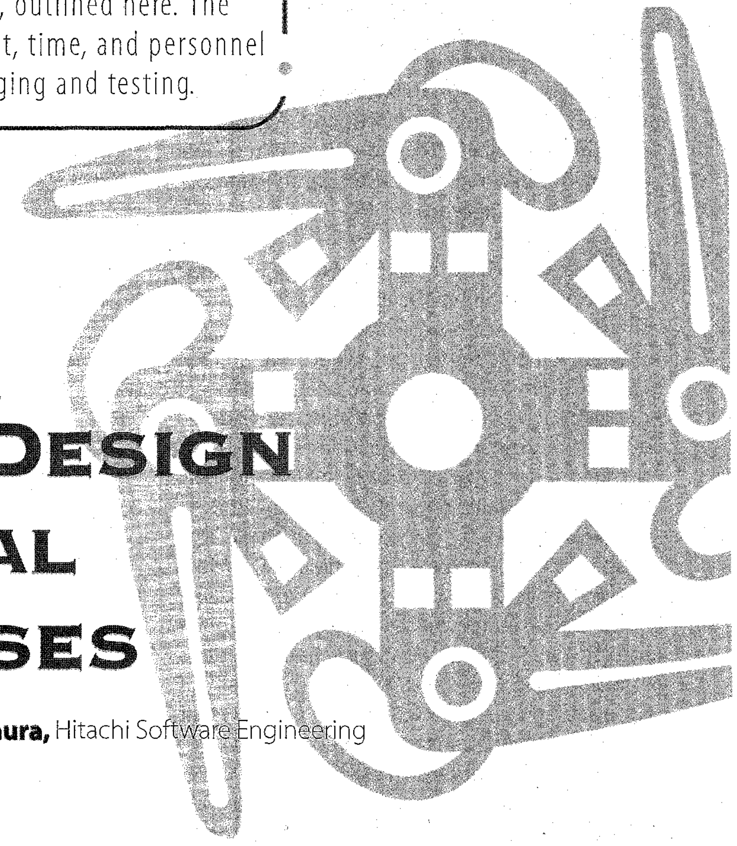
Programmers can build high-quality tests by following certain basic steps, outlined here. The author also discusses the cost, time, and personnel resources required for debugging and testing.



How To Design Practical Test Cases

Tsuneo Yamaura, Hitachi Software Engineering

At Hitachi Software, our software has attained such high quality that only 0.02 percent of all bugs in a software program emerge at the user's site.[1] If a typical project—10 engineers working for 12 months to develop 100,000 lines of code—contains 1,000 bugs, at most one will surface at the user's site. We do not use sophisticated tools or state-of-the-art methodology—we simply test programs and fix the bugs detected.

Our secret is that we document all test cases before we start debugging and testing. The written test cases provide significant advantages, and all our quality assurance activities start from this point.

Figure 1 shows the percentage of defects detected at each stage in a 1990 Hitachi software development project. (More recent data, still being collected, will not show drastic changes.) Of all the bugs detected since the project's inception, only 0.02 percent came out at the customer's site. Such high quality could never have been attained if we had not employed written test cases.

0740-7459/98/$10.00 © 1998 IEEE

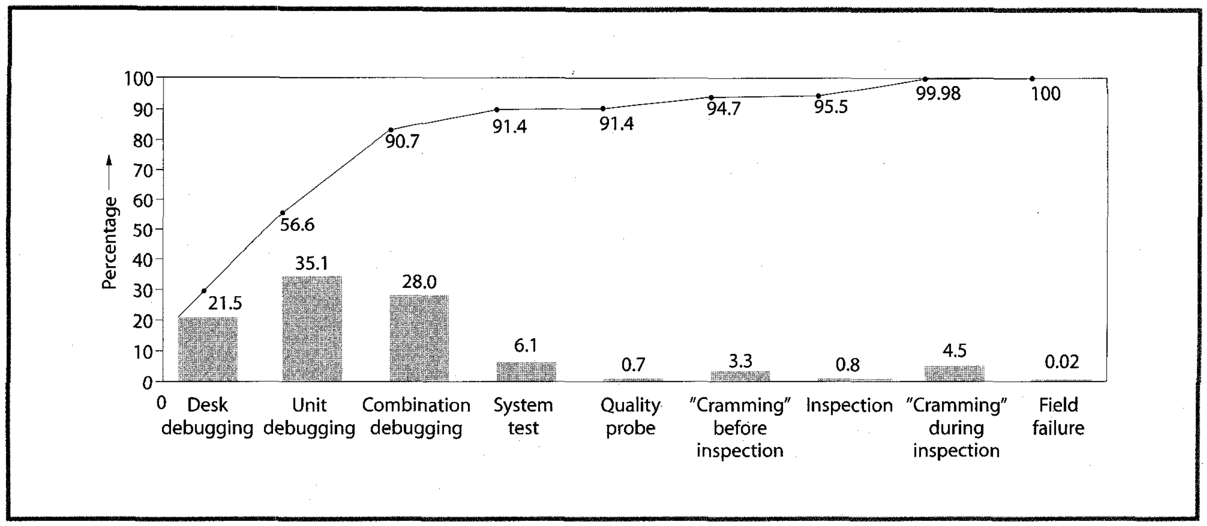


FIGURE 1. Bug detection during the development phases of a 1990 project.

Documenting Test Cases

Documented test cases can benefit your development process in several key ways:

♦ Designing test cases gives you a chance to analyze the specification from a different angle.

♦ You can repeat the same test cases.

♦ Somebody else can execute the test cases for you.

♦ You can easily validate the quality of the test cases.

♦ You can estimate the quality of the target software early on.

A new viewpoint

Test cases provide another rendition of the functional specification. Designing the test cases will give you pause: "Aha, I didn't consider such and such conditions. Was that specification really correct?" The bugs revealed in this way would be harder to detect and would require enormous time and money to fix later, such as in the system integration phase. Roughly speaking, you can catch 10 percent of a system's bugs when you are designing the test cases—a significant advantage.

Repeating test scenarios

You can easily reiterate the same test cases if everything is documented. Reusing test cases lets you reproduce bugs, which helps ensure that detected bugs are fixed properly.

Passing the test along

If you specify the test conditions, input data set, and expected outcome, you can ask somebody else to execute the test—which can prove particularly valuable on a project running late. Adding programmers to a project that is behind schedule frequently causes more delays because the project engineers must spare precious time to educate the new personnel. If the test cases are properly documented, however, the new staff can run the test cases as written.

Validating test case quality

You must, of course, test the test cases to ensure that they visit all the features implemented in the software. Check whether they are well-balanced among normal, abnormal, and boundary cases, and evaluate their overall sufficiency. Ad lib or random testing will never suffice—if you do not document the test cases clearly, you cannot precisely measure their quality metrics and success.

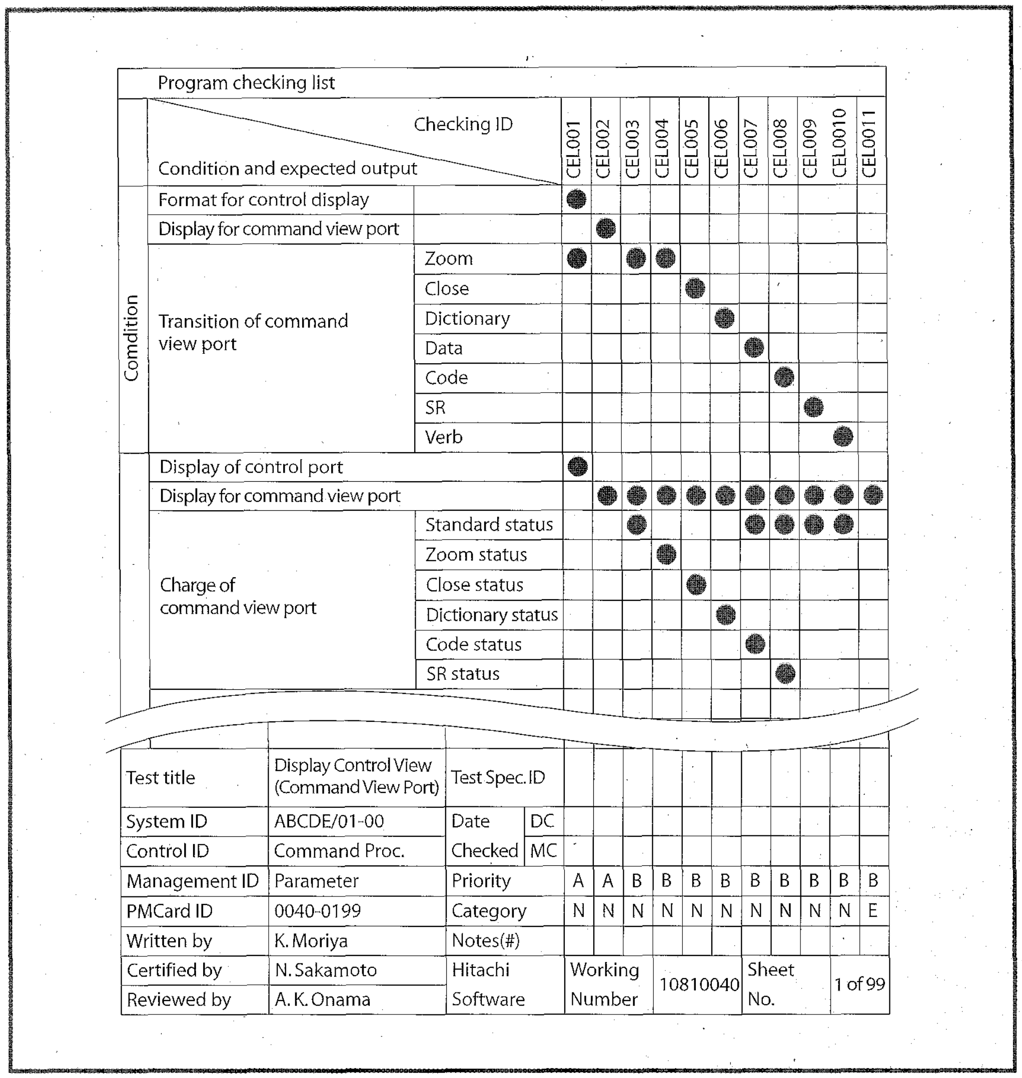
Estimating software quality

If the test cases are properly developed, you can easily estimate the quality of the target software in the midst of debugging by applying the fish-in-the-pond analogy: If you detect four bugs after executing 100 test cases out of 1,000, common sense says the software will carry about 40 bugs altogether. State-of-the-art software reliability theory is not that simple, but this quick and easy estimation gives you a rough idea of the software's quality.

Schedule, Cost, and Personnel

You see how useful it is to document the test cases. There is no free lunch, however—you must invest time, money, and personnel to enjoy the advantages. The question is how much you will need.

Here is a rough idea, with some statistical data.



Program checking list													
	Checking ID / Condition and expected output		CEL001	CEL002	CEL003	CEL004	CEL005	CEL006	CEL007	CEL008	CEL009	CEL0010	CEL0011
Comdition	Format for control display		●										
	Display for command view port			●									
	Transition of command view port	Zoom	●		●	●							
		Close					●						
		Dictionary						●					
		Data							●				
		Code								●			
		SR									●		
		Verb										●	
	Display of control port		●										
	Display for command view port			●	●	●	●	●	●	●	●	●	●
	Charge of command view port	Standard status			●				●	●	●	●	
		Zoom status				●							
		Close status					●						
		Dictionary status						●					
		Code status							●				
		SR status								●			

Test title	Display Control View (Command View Port)	Test Spec. ID												
System ID	ABCDE/01-00	Date	DC											
Control ID	Command Proc.	Checked	MC											
Management ID	Parameter	Priority		A	A	B	B	B	B	B	B	B	B	B
PMCard ID	0040-0199	Category		N	N	N	N	N	N	N	N	N	N	E
Written by	K. Moriya	Notes(#)												
Certified by	N. Sakamoto	Hitachi Software		Working Number			10810040			Sheet No.			1 of 99	
Reviewed by	A. K. Onama													

FIGURE 2. A matrix-based test sheet.

Assume that an average 12-month project with 10 engineers develops C-based software with 100,000 lines of code. An average project might apportion 12 months among the following phases:

- ♦ requirement specification: two months
- ♦ designing: three months
- ♦ coding: two months
- ♦ debugging: three months
- ♦ testing (handled by the quality assurance team): two months

At Hitachi Software, when the project completes debugging, the product is sent to the QA department where testers spend approximately two months evaluating whether the software is shippable. Since the testers must be unbiased and understand the customer's requirements, they never reuse the programmers' test suites; they redesign all the test items from scratch.

The test-case density—"how many test cases do we need?"—is one of the most critical engineering issues in testing. Too few test cases will leave easy bugs undetected, and too many will run short of time.

Our 30-year empirical study based on trial and error found that the proper test case density converges on the standard of one test case per 10 to 15

LOC. This means that a product with 100,000 LOC needs approximately 10,000 test cases—1,000 per programmer. Of 1,000 test cases, approximately 100 will be checked in the code inspection phase, and all of them will be checked in the machine debugging phase (which means that the first 100 cases for code inspection overlap with the machine debugging). Based on our study, these figures do not look unrealistic. An average programmer takes two weeks to execute 100 test cases in the code inspection phase (10 cases per day), and two months to execute 1,000 test cases in the machine debugging phase (25 cases per day). Ten programmers will take two weeks to design the test cases (assuming 100 test cases a day per programmer), or 3.8 percent of the entire project.

Steps for Debugging and Testing

Systematic testing follows six core procedures:

1. Design a set of test cases.
2. Check the test cases.
3. Do code inspection based on the test cases.
4. Do machine debugging based on the test cases.
5. Collect quality-related data during debugging.
6. Analyze the data during debugging.

Designing test cases

There is only one rule in designing test cases: cover all the features, but do not make too many test cases. We use a matrix-based test sheet to visit all the necessary functions, and apply equivalent partitioning and boundary analysis to eliminate redundant test cases. When needed, we use other test methods based on a state transition model, decision table, or dataflow model.

Figure 2 illustrates the matrix-based test sheet we employ. The first step is to itemize all the conditions, then consider all the possible combinations. This step will reveal considerable defects, because designing the test cases in this manner means redesigning the software based on another method, namely the decision table.

Note that all the test cases in Figure 1 have the corresponding expected outputs, without which you cannot reveal bugs while designing the test cases. You also need to indicate if the test case checks a normal, abnormal, or boundary case; this is essential to evaluate the quality of the test cases. And you must specify the testing priority, which shows the testing order.

In the early 1970s, we employed natural language to describe the stepwise conditioning of the test cases. For example:

♦ *When the person in the form is 65 years or older:*

♦ *When the person's annual income is $10,000 or less:*

♦ *When the person lives in area A-1:* … (the nesting of the conditions goes deeper)

Since this approach frequently caused us to overlook various combinations of the conditions (or holes in the test items), we migrated to matrix-based test design in the mid-1970s.

Equivalent partitioning,[2] a well-known testing technique, uses a single value to represent the same domain. Suppose, for example, an admission fee varies by age, such that there is no fee for age 6 or under, a $5 fee for 12 years old or younger, $8 for 18 years or under, and $10 for 19 and above. The test cases you pick up from each domain can be "age 2" for the domain of $0 \leq$ age ≤ 6, "age 10" for $6 <$ age ≤ 12, "age 14" for $12 <$ age ≤ 18, and "age 43" for $18 <$ age. Picking up "age 43" and "age 50" is redundant unless each represents a different domain in terms of the white-box testing.

Boundary analysis[2] is another well-known testing technique whose core idea is that bugs tend to exist on the borders of domains. Thus, in the admission fee example above, you need to test the ages of −1, 0, 6, 7, 12, 13, 18, and 19.

We extracted the following test case design criteria (or lessons) based on our empirical study.

♦ As I mentioned earlier, our optimized, pragmatic density of the test cases is one per 10 to 15 LOC. A language processor generally needs more test cases (approximately one test case per eight to 12 LOC) than a batch program, and an online program requires more (one per five to 10 LOC) than a language processor.

♦ From the viewpoint of white-box testing, the number of IF statements serves as a better index than the LOC because it relates directly to the number of executable paths in the program. The most error-prone structure in software is a loop. For white-box testing, you should test the iteration of 0, 1, 2, average number, max − 1, max, and max + 1.

♦ The basic and normal cases must constitute less than 60 percent of the test case set, with the boundary and limitation cases at least 10 percent, the error cases 15 percent, and the environmental test cases (whether the program runs on different

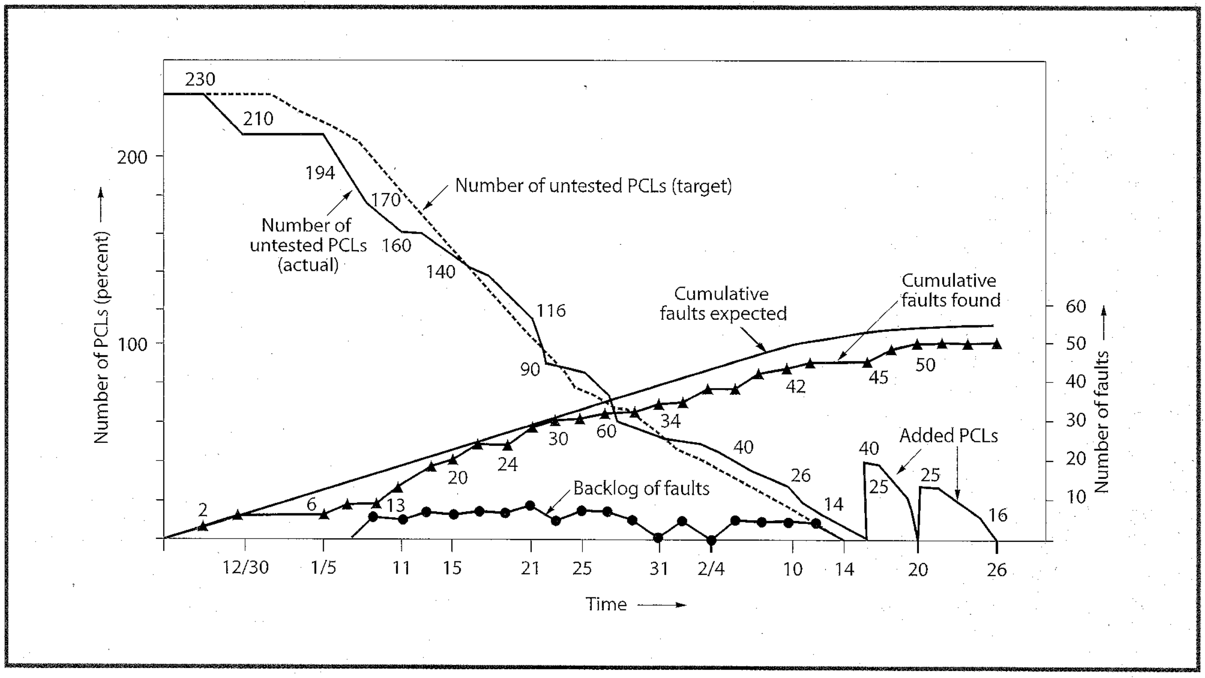


FIGURE 3. An example of test execution and bug detection.

operating systems, and performance requirements) 15 percent.

♦ As a finishing touch, we run a 48-hour continuous-operation test. All you have to provide is a test suite that unlimitedly reiterates the same basic functions. This test reveals many bugs related to a memory leakage, deadlock, and connection time-out.

Several tips can help you design successful and effective test cases:

♦ Do not design too many test cases, in particular for syntax testing. If you consider the error combinations of half a dozen parameters, you will easily end up with thousands of test cases, which may take several months to execute. The reality is, somebody—most likely you—must run the test. Design it so it can be completed within the planned period.

♦ Refer to previous projects that developed similar software. If they have test suites, by all means get them—you may not be able to reuse them, but they will give you insight.

♦ Indicate which test cases you will execute in the code inspection phase, and which in machine checking. It is advisable to use approximately 10 percent of test cases for code inspection; this can help you start tracing the software's main streets.

♦ A software engineer tends to make too many test cases for the features that he or she understands well and too few for unfamiliar functions. Compare the number of test cases with the LOC or number of IFs to reveal such an anomaly.

Checking test case quality

Test cases, of course, must be tested. When your test case design is complete, evaluate its properness and correctness based on

♦ whether the test cases cover all features;

♦ the balance between normal, abnormal, boundary, and environment test cases;

♦ the balance between code inspection (for checking hard-to-provide conditions and for enabling the detection of bugs that can be easily and effectively fixed) and machine execution (which eats up testing time);

♦ the balance between black-box and white-box testing; and

♦ the balance between functional tests and performance tests.

Code inspection

Our empirical study indicates that 21.5 percent of all bugs are detected in this phase. Of this number, I roughly estimate that code inspection reveals half, and test case designing reveals the rest. I recommend assigning 25 to 33 percent of the debugging time to check 10 percent of the test cases as code inspection.

Record the defects detected during this phase. This will tell you what bugs are left unfixed, where and of what type the bugs will tend to be, what module carries more defects, and so on. When you execute a test case successfully, put the completed date on the test case sheet.

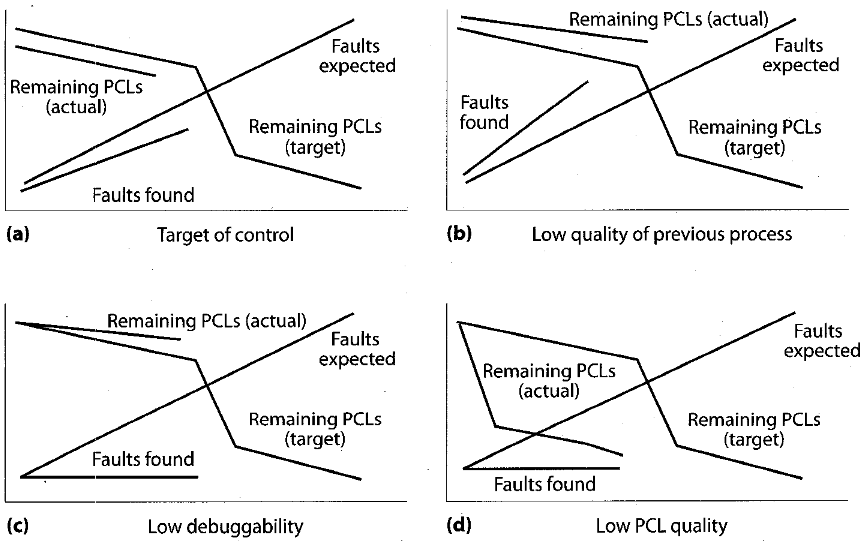


FIGURE 4. Diagnostics of test progress.

Machine debugging

This step is simple—just run a test case on the actual machine (or simulator) and keep a record of the result: passed or failed. When a test case reveals a defect, fill out a bug report to record information such as who revealed what, when, the symptom of the bug, and the test case ID. Without such information, you may end up thinking you have already fixed the bug—until it resurfaces.

Before starting machine debugging, implement the test cases as a test suite if possible. A test suite is a program that automatically runs test cases and compares the actual outputs with the expected ones to determine "pass" or "fail." This provides many advantages. Although the test suite implementation takes time (the test suite engine is small—more than 95 percent of the work goes to providing execution commands and defining the expected results), you save enormous time once it is built by letting your computer debug the programs while you are away. Stupid but costly typing errors can be completely eliminated, and you can use the test suite as a functional degradation checker in the final stage of software development when you have to make the scheduled shipment date but the program still needs modification and bug fixing.

You must remember one important thing when implementing the test suite: make sure it reinitializes all data when it goes on to the next test case. If a test case cannot be executed because a bug in the previous case terminates the test suite, the early bug may overshadow the consequent bugs, and these will not be revealed unless the early bug is completely fixed.

Collect data for quality analysis

To maximize the usefulness of the documented test cases, keep a record of all defects detected: their symptoms, seriousness, who caught them and when, the ID of the test case that caught them, the modules where they reside, and when they are fixed.

Also, project managers should collect the daily record of the number of test cases successfully executed and the number of bugs detected and fixed. They can plot this daily information and draw up a target plan to run further test cases, as illustrated in Figure 3.

Analyze the data

A diagram like that in Figure 3 offers useful information for controlling software quality and the expected shipment date. How should you interpret this data?

First, analyze the differences between the target number and the actual number of successfully executed test cases, and the target and actual number of detected bugs. This will help you pinpoint the problem in the early stage of debugging. Figure 4 illustrates four such diagrams. Figure 4a shows a target of control. Figure 4b shows that many bugs have been caught even though few test cases have been executed. This likely results from previous processes, such as module designing, not being properly completed or tested. In a very serious case, you should suspend debugging to revisit the previous process. Figure 4c indicates that the programmers cannot run the test cases as planned, hence very few bugs have been caught. This occurs when programmers are not able to debug a program. In such a case, ask an expert to oversee the programmers until they become familiar with debugging. Figure 4d indicates two possibilities: the test cases do not adequately reveal bugs, or the software does not carry many bugs. The former, of course, is much more likely than the latter, and the remedy is to create more effective test cases.

Second, if you catch 10 bugs for 10 days in a row, it is unlikely that today's are the last 10 you'll see. Bug occurrence never stops suddenly but diminishes gradually. From the bug accumulative curve, you can estimate when the curve goes flat, or the date you should attain the quality goal. We use the Gompertz Curve or the Logistic Curve for more accurate estimation, but intuitive observation alone can be a powerful tool.

Third, statistical data from bug reports can show you, for example, what the most common errors are, or what modules are error-prone. This kind of software metric "during the fact" works as a quality control parameter.

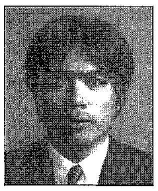
Finally, you can analyze what kind of defects your programmers tend to make—bug characteristics such as memory leakage, keeping a file unclosed, and so forth—and seek out errors that follow that pattern. This may be a project manager's task, but it also may reveal the presence of more complicated organizational and cultural issues.[3,4]

What might happen if a project does not employ the documented test cases or the QA approach based on the written test cases? Of 1,000 bugs, suppose only five percent of the bugs revealed during code inspection (215 bugs based upon the bug distribution in Figure 1) and machine debugging (631 bugs) are left undetected (43 bugs), and emerge at the end of the development life cycle, or—even worse—at the customer's site. Our data indicates that detecting and fixing each such defect requires two to three days. This means you will need an additional 86 to 130 days to raise the product quality to the desired level for shipment. We spend two weeks documenting the test cases and another two weeks executing the (written) test cases in the debugging phase—four weeks that may save four to six months. This alone justifies the usefulness of test case documentation.

References

1. A. Onoma and T. Yamaura, "Practical Steps Toward Quality Development," *IEEE Software*, Sept. 1995, pp. 68-77.
2. G.J. Myers, *The Art of Software Testing*, John Wiley & Sons, New York, 1979.
3. T. Yamaura, "Standing Naked in the Snow," *American Programmer*, Vol. 5, No. 1, 1992, pp. 2-9.
4. T. Yamaura, "Why Johnny Can't Test," *IEEE Software*, Mar./Apr. 1998, pp. 113-115.

About the Author



Tsuneo Yamaura is a senior engineer at Hitachi Software Engineering. His research interests include testing methodologies, software metrics, development paradigms, software modeling, and CASE.

Yamaura received a BS in electrical engineering from Himeji Institute of Technology and was a visiting scholar at the University of California, Berkeley. He is a member of the IEEE Computer Society and ACM.

Address questions about this article to Yamaura at Hitachi Software Engineering, 6-81 Onoe-cho, Naka-ku, Yokohama, 244 Japan; e-mail yamaur_t@soft.hitachi.co.jp.

CALL FOR ARTICLES AND REVIEWERS

Life in an Internet World: Software Security

Major sectors of the global economy like energy, transportation, and telecommunications are depending more and more on the Internet. This interconnectedness increases some security risks and creates a few new ones. We are looking for articles that address the following issues:

- How secure is online personal information?
- How vulnerable are major elements of our infrastructure (power, telephone, banking, etc.) to electronic attack?
- What is the tradeoff between easy access and security? What is technically feasible?
- How do we recover from short-term security breaches?

To submit an article or to become a reviewer, contact Angie Su at asu@computer.org. Articles must not exceed 5,400 words including tables and figures, which count for 200 words each.