REVIEW: OBJECTVISION GOES PRO

PRODUCT AWARDS

COMPUTER LANGUAGE

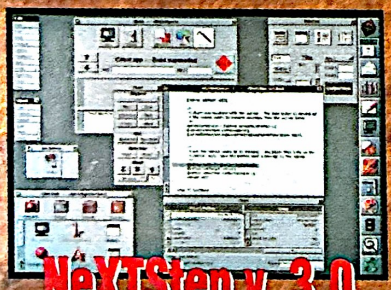
VOLUME 10, NUMBER 4

APRIL 1993

U.S. $3.95 Canada $4.95

1992's BEST TOOLS

•COMPILERS•UTILITIES
•CASE•LIBRARIES•BOOKS



NeXTStep v. 3.0

When will it ship? Defect management

FAST MATH AND THE TABLE

MIGRATION OF CHANGE

REVIEW

ObjectVision: Portable Database Programming for OS/2 and Windows

0 71896 48944 6 04

CONTENTS

APRIL 1993

Cover Story

On the cover: Feel that sugar and caffeine rush through your veins as you enjoy a cold Jolt cola and one of our award-winning products.

Screen shot: NeXTStep v. 3.0 from NeXT Computer

Carter Dow Photography

Articles

Single product review

Departments

Determining Software Quality

With a little simple math, you can implement a realistic crystal ball to predict the number of defects in your program.

by James Walsh

One of the great mysteries of any software development project is how many bugs are left in the program. While determining the number of known bugs in a software program is simply a matter of bookkeeping, estimating the number of bugs that have yet to be detected remains something of a black art.

While no technique can determine the number of remaining bugs with absolute accuracy, the systematic measurement of the rate at which new bugs are being discovered over time gives a valuable clue concerning software quality. The basic principle of this technique is an intuitively appealing one. The more defects there are in a software program, the easier they are to discover.

This theory implies that in a given amount of testing time, more bugs will be discovered when the software is buggiest than will be discovered when the software is more fully debugged. In other words, the defect discovery rate—the number of new bugs discovered per unit of testing time—tends to fall off with time as the defects in a program are discovered and fixed. By measuring the rate of falloff in defect discovery over the course of time in a project, the number of undetected bugs remaining in the program can be inferred with a high degree of confidence.

Finding bugs

Knowing the number of undetected bugs in a software program and the rate at which these new bugs will be discovered provides valuable information when deciding when the product will be ready to ship. While schedule pressures usually make it impossible to fix every defect in a commercial software program, finding every defect—or nearly every defect—sometimes is practical.

In this case, knowing your defect discovery rate and its rate of change allows you to predict how much additional time and effort will be needed to find the last defect. If schedule pressures force shipment on a certain date, knowing your defect discovery rate and its rate of decrease at that time will tell you how many unknown bugs will be shipping along with your product. This helps in estimating the incremental customer-support and maintainance costs to be expected due to customers discovering the new bugs at the predicted rate.

On its own, the defect discovery rate is a valuable quality metric, since it tells you how long a customer can use your product before encountering a bug. Sometimes ship decisions are made based on the defect discovery rate falling below a certain predetermined value and remaining there for some period of time.

Robert Grady and Deborah Caswell, for example, report an economically based criteria for ship decisions that revolves around the defect discovery rate. When the defect discovery rate becomes low enough that it would be more expensive to find the next defect through internal testing than it would be to upgrade the product in the field, the product is shipped.[1] The guideline they employ is that the defect discovery rate must remain below the threshold value for two weeks of active testing before the code is considered shipable.

The probable number of remaining bugs is also a very valuable quality metric. Numerically, most bugs in a well-written program are minor cosmetic problems. Defect severities tend to follow the 80/20 rule, with 80% of defects being noncritical and 20% actually causing program crashes, potential loss-of-data, or other critical condi-

LISTING 1.

```
#!/bin/csh -f
set VERSION=10ab
setenv  PROD_BIN        'rconfig ROSE_DEV'/main
onintr INTERRUPTED
set TIME="'time $PROD_BIN/rose_e $* )& /dev/tty'"
INTERRUPTED:
echo 'date' $USER "$TIME" )) 'rconfig ROSE_DEV'/rose_$VERSION.log
exit 0
# End of script.
```

FIGURE 1.

Excerpt from a log file

```
Wed Nov 4 11:04:41 PST 1992 pete 7.3u 8.1s 3:37 7% 0+3084k 24+41o 653pf+0w
Fri Nov 6 10:10:31 PST 1992 jdart 25.7u 15.7s 39:46 1% 0+2532k 61+211o 746pf+0w
Fri Nov 6 10:36:34 PST 1992 jdart 24.9u 17.1s 19:24 3% 0+2836k 32+211o 650pf+0w
Fri Nov 6 11:39:26 PST 1992 andrea 7.8u 10.5s 5:19 5% 0+2564k 46+01o 848pf+0w
Fri Nov 6 14:39:37 PST 1992 jimr 0.0u 0.1s 0:01 11% 0+112k 12+01o 21pf+0w
Fri Nov 6 16:21:39 PST 1992 sjl 47.9u 43.0s 44:19 3% 0+2096k 155+31o 1957pf+0w
```

tions. In addition, for many applications only about 20% of the total feature content is critical, in the sense that errors in these features impair the program's fitness for use.

The remaining 80% of all errors tend to be in areas of noncritical functionality or in an area with redundant means of accomplishing the same function. The probability, then, that a randomly selected bug will be a critical bug occurring in a critical functional area is roughly 20% of 20%, or 4%.

In other words, the chances are about 96% that a given bug is minor. If you leave N randomly selected bugs in your product at shipment, the chances that all N of them will be minor are on the order of 0.96^N. This probability becomes small quite rapidly; for example, if you leave 17 bugs in your product at ship, the odds that all of them will be minor is only 50%. This argues for leaving a small number of bugs in your program and making sure the bugs left are not randomly selected, giving critical areas of functionality a disproportionate amount of testing and bug fixing.

As a practical matter, measuring a project's defect discovery rate involves two factors:

- Knowing how many new defects were discovered in a given period of time
- Knowing how much testing was done on the software in that same period of time.

The defect discovery rate, then, is computed simply by dividing the number of bugs found by the amount of testing time. This yields the value for "defects per unit of testing time" that we refer to as the defect discovery rate. This process is repeated for each interval of interest.

Debugging Rational Rose

Our experience in the Rational Rose project, a 230,000 line, 700-class CASE tool written in C++, provides a good illustration. We used an automated defect tracking system for all bug reporting. Not only was an automated tool valuable to simplify the bookkeeping and workflow management tasks needed to resolve the bugs, it was also an important collection point for our defect discovery rate statistics.

As defects were submitted to our on-line defect tracking system by users and testers, the bug report was automatically tagged by the system with a submission date and time. While initially not all defects were reported to the tracking system as they were found, over the course of time (as the "folklore" defects were all input, and the value of having timely defect data was understood), compliance became extremely high.

Over the course of the project, we could say with confidence how many new defects were discovered on a certain day. "New defects" means valid defects that were not previously discovered; this screens out duplicates, rejected bugs, and enhancement requests. The number of new bugs attributed to a given date would sometimes vary as the bugs submitted on that date were later examined and classified. This also caused the daily defect rates to fluctuate, depending on our later disposition of that day's bugs. At this writing, our record of daily defect data stretches back for well over a year to the very early stages of the project.

If the amount of testing being done each day is uniform, the daily bug count gives a good measure of the defect discovery rate. In a UNIX multiuser environment, however, it is relatively easy to nonintrusively instrument an application so that its total execution history by all users is recorded, and this gives more accurate results.

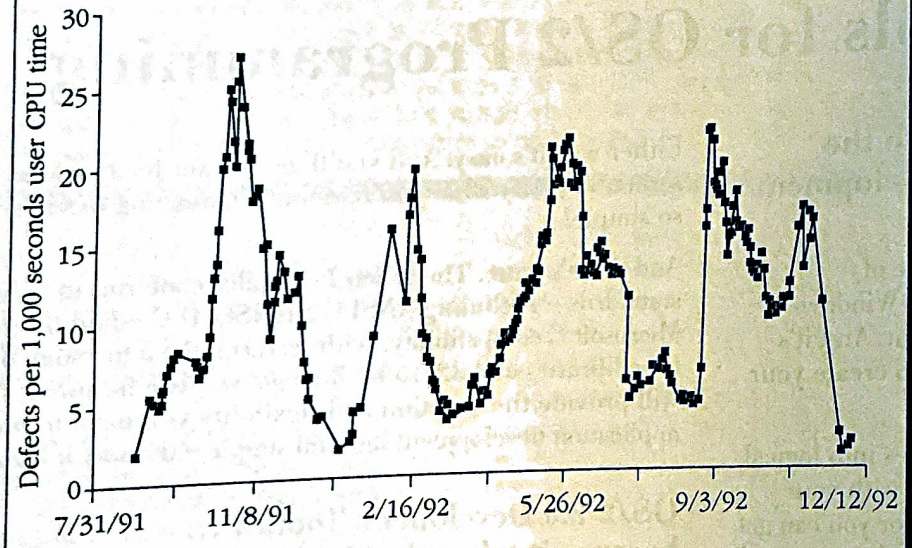
To record the total execution

LISTING 2.

```
YYMMDD UserCPU SysCPU clock #defects
921102 0.0 sec 0.0 sec 00:00:00 1
921103 42.7 sec 44.6 sec 01:07:47 3
921104 63.2 sec 65.9 sec 17:14:24 1
921105 5.9 sec 11.8 sec 00:27:50 0
921106 233.5 sec 201.8 sec 32:59:05 0
921107 0.0 sec 0.0 sec 00:00:00 0
```

FIGURE 2.

Plot of project defect discovery rate



time for our software, we wrote a simple shell script, shown in Listing 1, around the binary. This script spawned a sub-shell, using the UNIX *time(1)* command to execute the program under test and record the total CPU and elapsed clock time consumed by each invocation.

This script also recorded other useful data, such as the ID of the user invoking the program, a timestamp for each program invocation, and appended the record of the current use to a log file for later analysis, as shown in Figure 1. In this way, by always using the logging shell script to invoke the program under test, the actual total execution time—clock or CPU—of our software could easily be determined for any time interval.

By using our automated defect tracking system to determine how many bugs were submitted on a given day and our logging script to track how much our application had been used on that day, we had the elements in place to determine our defect discovery rate. A simple script correlated the defect and execution time data, logging for each day and for all users the date, total seconds of user CPU time, total seconds of system CPU time, total elapsed clock time, and number of new bugs reported that day, as shown in Listing 2.

Our results showed very little correlation between clock and CPU times for our application, other than that the clock times tended to cluster in 24-hour intervals as some users occasionally left the application up semi-idle on their screens for several days at a time. In general, for application programs there is very little correlation between the number of hours a program is displayed on screen and the amount of time its functionality is actually being exercised.

Correlating results

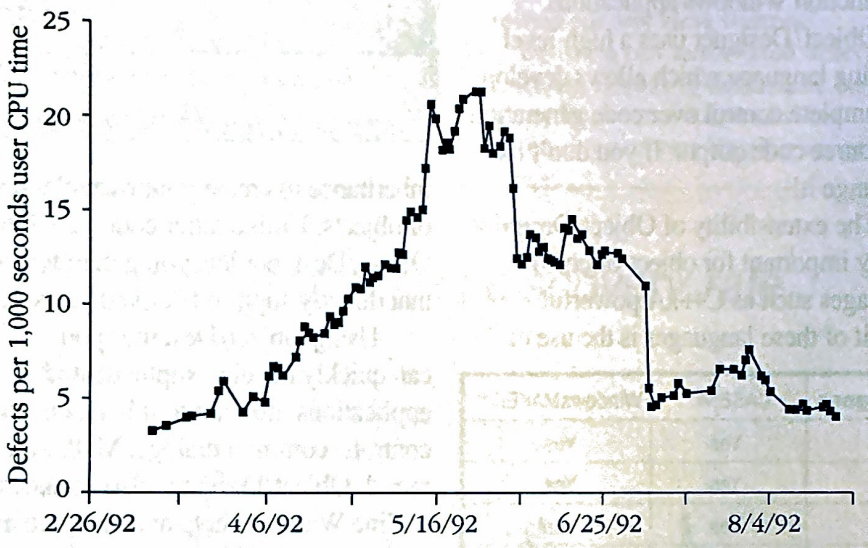
To compensate, we used CPU time in all our defect rate calculations, because CPU time is only consumed during the active use of an application. In addition, we used only the user portion of the CPU time, on the theory that it more accurately reflected use of code we had written than the total or system CPU time would, since the latter is involved primarily with the execution of library functions.

Having available a daily record of the number of new, valid defects reported and the total amount of CPU time consumed by our application each day, computing our daily instantaneous defect discovery rate was simply a matter of dividing the two numbers. The instantaneous defect discovery rate, however, is highly variable, depending on the type of use and program features exercised on a given day and whether or not the bugs discovered were logged on the same date. In fact, we would occasionally have situations where bugs were reported on a day with no CPU use, giving infinities in our instantaneous rates.

To cope with this variability and more easily spot trends, we averaged our data, computing our defect rate for each day by dividing the number of bugs discovered in the previous three weeks by the total amount of user CPU time consumed in that period of time. Averaging, of course, introduces a

FIGURE 3.

Plot of defect discovery rate as a function of date



degree of inertia to the data, masking short-term effects. However, for our project we found a three-week averaging interval provided adequate smoothing and still revealed changes fairly quickly.

We also excluded from our defect data the CPU time consumed by our automated regression testing tool. This tool was used to execute automated test suites, and, at peak times, drove the application under test around the clock for days at a time on several workstations simultaneously. Though a vital element of the software quality assurance process, this form of testing—because of its repetitive nature—tends to find few new bugs per unit CPU time.

In any case, it is not representative of the type of use the application would receive in the field. For these reasons, the scripts that accumulated utilization data excluded execution times corresponding to runs of the automated regression tests. New defects found by these utilities were included in the bug counts, however. In addition, bugs submitted against the software's documentation were excluded from the bug counts; only bugs in the software were counted for purposes of computing the defect discovery rate.

Making it make sense

Once our defect discovery data was processed, it was graphed. For our application, we found it convenient to use units of "defects per 1,000 seconds of CPU time" to graph our data because the Y values fell quickly between 0 and 100. These graphs formed a revealing picture of the quality level of our product throughout the project life cycle, as shown in Figure 2.

During the course of development, our product passed through four major iterations, each of which corresponds to a peak on the defect finding rate curve, as shown in Figure 3. In each iteration, the defect discovery rate curve has a leading edge during which the discovery rate is rising, a plateau during which the rate is roughly constant, and a trailing edge during which the defect rate rapidly declines. These three periods correspond, respectively, to the product's integration phase, the alpha test phase, and the beta test or customer ship phase.

During the integration phase of a project, where previously written code fragments or subsystems are being brought together to work in combination for the first time, the defect discovery rate—paradoxically—starts out being low. Over the course of time, as integration proceeds, the discovery rate rises. Though initially counterintuitive, there is a good reason for this phenomena and even a name: defect masking.

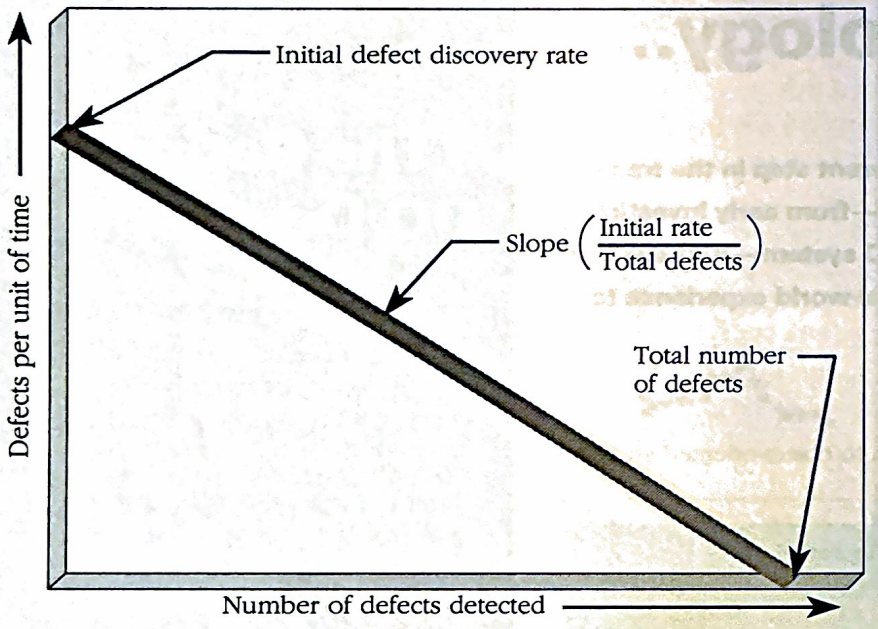
The defect discovery rate is initially low because there are so many severe bugs in the code that they hide or mask each other. For example, if the application immediately crashes whenever its File menu is accessed—as may well happen during the earliest phases of integration—none of the options on that menu can be exercised.

Thus, a high-level, severe bug prevents the exercise of other program features, masking other bugs that may be in them. This gives a low defect-discovery rate because no matter how long the program is exercised, only a small set of bugs will be found, as little of the program's functionality is accessible.

As integration progresses, more and more of the programs' functionality becomes accessible. This makes the process of finding defects more efficient, since more test scenarios can be completely executed, and the defect discovery rate rises until the product be-

FIGURE 4.

Musa's basic model



comes fully testable. At the time all the product's functionality becomes fully testable, we say that the integration phase has ended, and the alpha test phase has begun. Indeed, we used the leveling off of the defect rate as an objective measure of when integration was actually completed.

Alpha testing is characterized initially by a plateau in the defect discovery rate. This is because, though the system is now fully testable, it still contains a large number of bugs and defects that are being reported as fast as they can be found and logged into the defect tracking system. An upper limit on the defect finding rate is imposed not by the number of bugs in the software, but rather by the test team's efficiency in finding and reporting such bugs.

Even if the software contained an infinite number of bugs, they could not be logged any faster than the efficiency the finding and reporting process permits. This is why the peak defect discovery rate for each iteration of our software is roughly constant. Our peak efficiency finding and reporting of bugs is somewhere on the order of six bugs per clock hour of testing. It is the breadth of the plateau rather than its height that is the indicator of how buggy a given release is.

Keep testing

Until the number of bugs reported begins to approach the actual number of bugs in the software, the defect discovery rate will remain roughly constant. During this frustrating phase of product development, there is no way of distinguishing between a finite number of defects in the program under test and an infinite number; both would show the same defect discovery rate. Once a significant fraction of the defects remaining in the software has been reported, the defect rate begins to fall off because new defects become harder to find.

More testing is required to find each new defect. At this point, a sudden or systematic drop in the defect rate will be encountered, and we can begin to infer the number of undetected bugs remaining in the software.

While plotting the defect discovery rate vs. time can provide valuable project-management information, it is more useful to plot the defect rate vs. the cumulative number of defects discovered when you want to infer the number of defects remaining. The cumulative number of defects provides a nonlinear time axis that normalizes out the effects of periods when little or no testing was done.

Musa's technique

Work by John Musa and his colleagues has established a strong theoretical basis for determining the remaining defects in a program from the rate of change of the defect discovery rate with respect to the number of defects discovered, as shown in Figure 4.[2] Musa's basic model suggests that if a straight line is fitted through the curve describing the decline in defect rate, this line will intercept the X axis at a point corresponding to the total number of defects in the program.

Subtracting the number of currently known defects from the total number of defects predicted in the program gives the number of currently undetected defects. Musa also describes a more sophisticated exponential model, but we will focus on the basic model here.

To illustrate how this technique may be used in practice, let's look at the defect discovery rates for one

iteration of the UNIX version of our product, shown in Figure 5. Fitting a linear regression curve to the falling slope of the defect rate curve for this iteration predicts a total of 1,154 cumulative defects. Subtracting the 1,011 defects known at the time the code was frozen suggests that there were 143 unknown defects left in this iteration. Assuming that there were actually 1,154 known plus unknown defects in the first iteration, and knowing that there were 200,000 lines of code in this iteration gives us a defect rate of about six defects for every 1,000 lines of code.

In the next iteration, 30,000 lines of new code were added. If this code had the same bug rate, we could predict that 180 new bugs would be introduced in the next iteration on top of the 143 undetected bugs remaining from the previous iteration. We knew that about 323 bugs would be present in the next iteration of the product before testing had even begun on it. In fact, we found 433 bugs in the next iteration of the project, at which time the defect discovery rate fell rapidly to our target level. This discrepancy could either be in our original bug estimate, or it could be because the new code had a higher defect rate than the old code.

Perfecting perfection

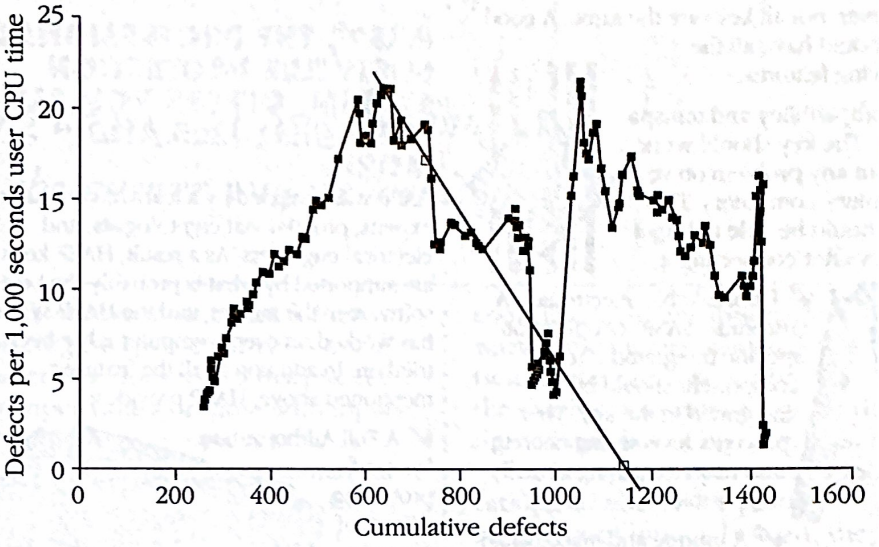
Inferring the number of defects remaining in a software program is not an exact process, even using these statistical techniques. Estimating to an accuracy of 34% is too good to be hoped for in most cases, as simply choosing the interval over which you fit the regression curves can introduce considerable variability. We attempt to be conservative in this procedure by considering upper and lower bounds for the number of remaining defects in our planning process. However, simply having a rational basis for determining upper and lower bounds on the number of remaining defects is a considerable advance in the state of the art.

Knowing how many undiscovered bugs are in the current version of your project and how many will be present in the next version is very valuable information. It allows you to more accurately estimate test and debug times, ship dates, customer satisfaction, support requirements, and more. There is no magic to the process, nor is it guaranteed to produce 100% accurate estimates.

Putting the systems in place to measure the number of defects found and how much your software has been used and using this information to measure your defect discovery rate, however, will tell you a lot about the quality of your software. Having hard data on the quality of your product instead of relying solely on subjective guesses is an essential step to making the informed decisions needed to produce a first-rate system. ■

FIGURE 5.

Plot of defect discovery rate as a function of cumulative defects



References

1. Grady, Robert B., and Deborah L. Caswell. *Software Metrics: Establishing a Company-Wide Program.* Englewood Cliffs, N.J.: Prentice-Hall Inc., 1987, p. 128.

2. Musa, John D., Anthony Iannino, and Kazuhira Okumoto. *Software Reliability: Measurement, Prediction, Application.* New York, N.Y.: McGraw-Hill Book Co., 1987, p. 32.

James F. Walsh is currently software quality manager for NeXT Computer Inc. Most recently, he was manager of software quality for the UNIX version of Rational Rose, where he performed the work described in this article.