Measuring the Field Quality of Wide-Distribution Commercial Software

Garrison Q. Kenney
Department D19, Building 656
IBM Corporation
Research Triangle Park, NC 27709

Mladen A. Vouk
Computer Science Department, Box 8206
North Carolina State University
Raleigh, NC 27695-8206

Abstract

The problem of quantifying the field quality of wide-distribution commercial software is addressed. We argue that for this type of software the proper quality metric is an estimate of the number of defects remaining in the code. We observe that the apparent number of defects remaining in this type of software is a function of the number of users as well as the number of actual defects in the code. New releases of commercial software normally consist of some code from prior releases and some new or modified code. We continue to discover new defects in the code from prior releases even after a new release is in the field. In fact, the new release code appears to stimulate discovery of defects latent in the "old" code and cause a "next release effect." Field defect data from several releases of a widely distributed commercial software product are shown to demonstrate this effect.

Introduction

Much has been researched and written on the task of estimating software reliability growth and the number of defects remaining in software. The majority of this work has been directed toward estimating the operational reliability of the software from data obtained during system test. This estimate gives us a hard quality number to use along with the hard schedule date to decide on the best time to stop testing and deliver the product for operational use with an acceptable level of quality. (Actually, the concept of software quality extends beyond the attribute of performance according to specifications without failure, which is being discussed here. One might also consider the question of whether or not the specifications meet the customer requirements as another, equally important, aspect of software quality.) Since the real test of how well the development and test process did in producing quality software rests in how well it performs in the field under operational conditions, we need to take a close look at the software performance during the operational period. Only then can we validate how well the system test model projects the actual field experience. This is not too hard to do where the software is used on a limited number of systems and we can directly monitor the reliability. However, when the subject software is widely distributed commercial software that is used by thousands of customers in varied configurations, not only is direct measurement not possible, we find that there are new and interesting phenomena that need to be explained.

Wide-distribution commercial software

When we talk about any software attribute, we really need to qualify what type of software it is that we are talking about. For example, Boehm [1] classified software into three classes for the purpose of estimating cost and schedule. His Organic mode corresponds to in-house development of applications, Semidetached mode corresponds to medium complexity software developed under contract for a specific application, and Embedded mode corresponds to the most rigidly specified and scheduled software. In a like manner, we should classify software for the purpose of measuring quality in operational use. We have all come to make the distinction between mission critical and non-mission critical software. There is also a need to distinguish between software that runs on a single system or at most a limited number of systems with a homogeneous hardware configuration, and software that runs on thousands of systems and possibly on widely differing hardware configurations. How we measure the field quality of such software is strongly influenced by this large and disparate set of users.

The software we consider in this paper falls into the class of "wide-distribution commercial" software, i.e., software that is developed for the purpose of selling on the open market as opposed to contract software for a specific customer use, or in-house software developed for in-house use. From a user's view this type of software has been termed commercial off-the-shelf (COTS) [2] software. By definition, we do not know who the customers of "wide-distribution commercial" software are or how much they
actually use the software. Although we do (should) know how many licenses there are, we don't know how much each copy is used. Software for single-user systems may be purchased and installed but rarely, if ever, used. Software for multi-user systems may have a few users or thousands of users depending on the size of the system. In fact, with site licensing, we do not even know how many copies of the software are really in use in many cases.

Wide-distribution commercial software lives in a world of multiple releases, and so for whatever method we come up with to measure field quality, it must accommodate multiple releases. For example, current releases (as of 3/31/92) of some examples of such software are Word-Perfect release 5.1, Microsoft Windows release 3.1, Apple Macintosh's System 7, IBM's VTAM release 4.1, DEC's Ultrix 4.1, etc.

It is well known that when a new release of a software product is made available for customer use, the reporting of unique field problems initially rises to a peak and then tails off as time goes on [3][4]. This is generally attributed to the reporting of failures caused by defects that were inserted into the new code for the release. However, if we classify the defects into which release they originated, it turns out that this rise may not be totally due to defects inserted into the new release. The new release may somehow stimulate discovery of latent defects that were inserted into prior release code but never discovered. This phenomenon aggravates the stability of a new release in its early days, and makes predictive modeling difficult.

**Counting defects as a measure of code quality**

Reporting on the estimated number of defects remaining in the code rather than the reliability of the code is a more appropriate measure of the field quality of wide-distribution commercial software. We realize that this can be controversial, but we think it is important when we are talking about the operational quality of software that runs on thousands of systems. Although the reliability of the software is of primary interest to an individual user, it is not a very informative way to characterize the overall quality of the software when there are thousands of users.

Much of the current literature concentrates on measuring and predicting the individual system reliability or mean time to failure (MTTF). This is an important measure to an individual user of the software -- how long he can expect to get productive work out of the software until it fails in some way that disrupts his use. However, when we are talking about wide-distribution commercial software, we have many, many users, and each experiences his own level of reliability. One user may run for months without a disruption and another may run for only a few hours before experiencing a failure. A user's reliability experience depends as much on how he uses the software and on how wide a range of features he exercises as it does on the inherent quality of the code. Consider a commercial word processing program. For a user who simply uses it to type routine one-page letters, the software may work very reliably. On the other hand, if another person uses graphics, equations, tables, footnotes and endnotes, varying font types and sizes, etc., in a document that is 100 pages long, he is much more likely to run into defects in the code that cause failures, and so the reliability of the software for this user may be much lower. Which reliability number is the right one to characterize quality of the software as a whole? Obviously, it doesn't make sense to use either one of them, although a range composed of the two extremes may be acceptable. An alternative is to characterize the software as having some number of defects left in the code. One user may happen to run into those defects while another may not. By estimating the number of defects remaining in the code, it is possible to distill the reliability experience of thousands of users down to one number that characterizes the software product as a whole.

Another problem with using reliability as a measure of quality for commercial software is that the users don't generally keep good records on how much execution time passes between failures. Although reliability of a software system is important to customers of commercial software products, they do not report on their reliability experience. What they do report to the software development organization is the occurrence of a specific failure, with the expectation of getting the underlying defect fixed so that the failure does not reoccur. This is another reason why many commercial software development organizations focus on the number of remaining defects rather than reliability or mean time to failure as the measure of software quality.

This argument for counting defects in the code instead of using individual system reliability as the measure of software quality may be open to the criticism that not all defects are the same severity, and that we need to somehow separate the defects that cause a severe hardship to the user from those that cause only a minor inconvenience. There is no question that some defects are "worse" than others, but we choose to lump them all together in the same way that failures in software reliability calculations are lumped together. In reliability calculations, the software either failed or it didn't. We don't generally assign a severity to the failure and somehow work this into the calculation of reliability. Perhaps the biggest difference is that the measurement of reliability is based on the user's report that the software failed (and failed badly
enough so that in the user's opinion this particular failure counts in the calculation of reliability), whereas counting the number of defects remaining in the software depends on some user thinking that it failed badly enough to report the problem to the development team. In counting remaining defects, if any user reports the problem, it's severe enough to count as a defect.

**Apparent number of defects is a function of the number of users**

The apparent quality of software is a function of the number of users of the software. If we take a piece of software, and a user runs it operationally on one system for some long period of time, we will discover some number of unique defects that were latent in the software. If a thousand users operate that same software on a thousand different systems, we will find that it apparently has significantly more defects.

Of course, the defects were there all along; it's just that we didn't find them with one user running on one system because the operation of the software became "grooved" to a specific machine running a certain class of data by this restricted set of users, and so the defects were never encountered. On the other hand, software that is used on thousands of systems by thousands of users has the opportunity to meet up with many different machines, each with its own unique timings and configuration of peripheral equipment; and with thousands of individual users, each with his own unique set of data and operational habits. In other words, new users on new hardware configurations with new data and new operational habits find new bugs. So whenever we measure the operational quality of software, we must somehow take this phenomenon of the number of users into account.

Several researchers have recently noticed and reported on this phenomenon, but have not really pursued a consistent way of accounting for it. Uemura et al. [5] observed that the estimated number of latent defects increased each time that their software acquired new users. This led them to postulate that software reliability more or less depends on the characteristics of the users or groups who use it, and that this applies specifically for software packages that are produced for quantity sales and that have unspecific, mass users. Ohtera and Yamada [6] use a different model during operation than that used during test because "it is commonly known that software error detection phenomenon in the user's operation phase is different from that in the software testing phase." However, they offer no explanation for why it is different. Yamada et al. [7] offer some explanation to account for the difference between system test estimates and actual operational use.

They simply state that this difference occurs "because many unknown users use the same software" during the operational phase. Martini et al. [8] also recognize that we must take the usage rate into account in order to model the defect discovery process during operational use. Their rationale is that, "The increase in the number of running software configurations lengthens the execution time for the same calendar time."

![Figure 1 Arrival distribution of field defects for release X of a commercial software product.](image1)

![Figure 2 Arrival distribution of field defects for release X+1 of a commercial software product.](image2)

We also have observed this dependency of the rate of discovery of the remaining defects in software on the number of users. An example of this can be seen in the defect discovery rates for two releases of a wide-distribution software product. Figure 1 shows the discovery rate for release X of the product, where there were approximately 100,000 systems on which the software could run at the time it was released. Figure 2 shows the discovery
rate for a subsequent release of the same software product where there were approximately 200,000 systems on which the product could run at the time of release. (The secondary peak in the eighth quarter of Figure 2 is discussed in the next section.) Even though the subsequent release had fewer total defects, the peak discovery time was shifted from the third quarter back to the second quarter. We suspect this is because of the rapid installation of the subsequent release on twice as many systems.

These references show that there exists a link between usage and defect discovery rate. They also hint at some of the reasons for this link. Our hypothesis is that the following factors contribute to the increased discovery of defects for wide-distribution commercial software in operational use:

- **Increased usage alone** can contribute to increased discovery of defects because it will increase the execution time for the same calendar time. This is obvious. If we have a thousand identical systems running for a year, it is the same as a single system running for a thousand years in this respect, and we will find defects for this particular usage at a much accelerated rate.

- **Different users** can contribute to increased discovery of defects. This is because they expose the software to different data, and also because each user has a different set of habits as to how he deals with the software.

- **Different hardware configurations** can contribute to increased discovery of defects because of timing changes as well as interaction among different hardware components.

- **Different levels of operating system** can contribute to increased discovery of defects if the software in question is an application program that uses the services of such an operating system. A new operating system level could cause timing and component interaction changes similar to a hardware change.

These factors contribute to the field defect discovery rate in varying degrees for different types of software. For example, contract software that is developed for a specific application but is installed on several systems across the country would be exposed to some of these perils, but the effect may only be slight. However, the impact of these factors is so strong on wide-distribution commercial software that runs on thousands of systems, that we must take them into account when measuring the field quality of this type of software.

Because of these reasons, we continue to find defects in this type of software several years after it is first introduced. This is readily apparent from an area graph of several releases of the software product. Figure 3 shows the defect rate caused by each release. To bring out the trend and provide some smoothing of the arrival times, the data are shown as a running three-month average. (For example, month 5 shows the average of the defects discovered in months 4, 5 and 6.) The first customer ship of release X is at relative month zero, and the release dates of subsequent releases through X+5 are shown on the graph. (Note that, because of the averaging over three months, in some cases it appears that defects are discovered for a release before it is shipped.)

The "Next Release Effect"

An interesting observation in the arrival pattern of field defects in widely distributed commercial software is that we not only continue to find defects in "old" code following the shipment of a new release, the new release somehow stimulates the discovery of defects in the old code. We can see this somewhat from Figure 2 and Figure 3, but it is more readily apparent in Figure 4 through Figure 7. Here, the line outline is the total defect discovery rate for all releases (the total shown on Figure 3), and the bars beneath represent the contribution of the total that is caused by code inserted into a particular release. The major peak in the overall defect rate is caused by the new code for a release, but we also see minor peaks following the shipment of later releases.

This "next release effect" was not expected. The defects that cause it are evidently well hidden in the code prior to the next release, and the new release increases their likelihood of being encountered.

A possible reason why a defect remains hidden until being stimulated by a subsequent release is that there is a "window" of converging events that must all come together to expose the defect. The window is there from the beginning, and eventually the defect would have been tripped over, but it was very unlikely to occur. New code in the next release changed the instruction path lengths enough so that the window of opportunity for tripping over the defect lined up in a more likely time. For example, we are more likely to be in a certain piece of code when an interrupt occurs due to a change in code path lengths. Another possibility is that the window of opportunity is much wider now because of added (or deleted) code.

The next release effect could also be an aberration caused by the way we assign the discovery date and release of the defects. Rather than the next release stimulating the discovery rate of indigenous defects, it could be that:
A developer may assign the defect to a prior release incorrectly. The defect never would have been reported had some new function not been put into the code. Although fixed in code that had been there before the new release, the defect may have really been caused by new code that did not recognize a special case assumed in the prior release. Possibly, a particular state could never have been entered, but the new release code now caused the forbidden state to be reached. The designers and developers of the new release did not account for the special case assumed by the prior release design restriction.

Users likely do not report every failure that they come across. The effect of some failures is slight and the user may choose not to waste his time going through the documentation, reporting and reproduction process. He may think that he can live with the possibility of this failure reoccurring, or live with it until it is hopefully fixed in the next release. Then, when the next release comes and it still isn’t fixed, he may decide to take the time and report the problem.

A user may encounter a problem in the weeks just before the availability of a new release. The field support team recommends to the customer that he upgrade to the new release to possibly fix the problem; the original problem report is closed. After the customer upgrades to the new release, the problem reoccurs and a new problem is opened. Eventually, the problem is traced to the defect in the original release, fixed and closed. However, the recording of the problem was delayed from the latter days of release X into the early days of release X+1.
The developer classifies the defect into the wrong release for some other reason. This could be because he didn't understand the tracking system codes, didn't give it enough thought, or simply made a mistake in typing the release number. (He may have had his mind on other things, or he may think that keeping track of such things is a waste of time.)

We need to explore these possible reasons and report on the results later. Depending on the results, we can keep better records and make the effect go away, or we may need to incorporate the effect into our model of the field defect discovery process of wide-distribution commercial software.

Conclusions

From these observations, we conclude that the current software reliability growth models are not adequate to model the field quality of wide-distribution commercial software. We need a new model that can account for each of the following factors:

- The metric for field quality of widely distributed commercial software should be the estimated number of defects remaining in the code as opposed to the estimated reliability. This is because we want to measure an attribute of the software itself, and not an attribute of how a particular use of the software behaves. Software reliability is important in talking about an individual user or system experience, and we need to continue to use it in this sense. But a good estimate of the number of defects that remain in the code distills the reliability experience of thousands of users down to one number that characterizes the software product as a whole.
The apparent number of defects in widely distributed commercial software is strongly influenced by the number of users of that software. New users on new hardware configurations with new data and new operational habits find new bugs. When the number of users grows to the thousands, there is a lot of opportunity to find those bugs.

New releases of a software product stimulate the discovery of latent defects that were inserted into "old" code but were never previously discovered. We need to better understand this "next release effect" and account for it in our field quality models.

As an additional observation, it seems obvious from these conclusions that we can't test all the defects out of the code in the lab before we ship a software release to customers. If unique defects remain lurking in the code over several releases, only to be discovered by the confluence of some new user, system and release, there is no hope that we can simulate these conditions in the test lab. We will have to depend more and more on defect prevention to attain our software quality goals. One can see why it is practically impossible to reach zero defects in software by testing out bugs given these conditions. If we are ever to approach zero defects in any substantial software product, it will have to be as a result of defect prevention and very robust software.

References


