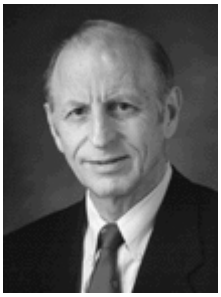
Bugs or Defects?

Watts S. Humphrey



One of the things that really bothers me is the common software practice of referring to software defects by the term “bugs.” In my view, they should be called “defects.” Any program with one or more defects should be recognized as defective. When I say this, most software engineers roll their eyes and feel I am out of my mind. But stick with me and you will see that I am not being unrealistic.

To explain why the term “bug” bothers me, I need to pose three questions. First, do defects really matter? Second, why not just worry about the important defects? And third, even if we have to worry about all the defects, why worry about what we call them?

Do defects really matter?

To answer this question, we first need to find out if defects are or can be serious. To do that, we must define what we mean by “serious.” Here, while there will be wide differences of opinion, there is almost certainly a basis for general agreement. First, if a defect kills or seriously injures someone, we would all likely agree that it was a serious defect. Further, if a software defect caused severe financial disruption, we would all probably agree that it too was a serious defect.

Lastly, there are lots of less significant problems, such as inconvenience, inefficiency, and just plain annoyance. Here, the question of seriousness obviously depends on your point of view. If you are the one being inconvenienced, and if it happens often, you would likely consider this serious. On the other hand, if you are the supplier of such software, and the inconveniences do not cause people to sue you or go to your competitors, you would probably not judge these to be serious defects. However, if these defects significantly affected the bottom line of your business, you would again agree that these too were serious defects.

So we are left with the conclusion that if defects cause loss of life or limb, result in major economic disruption to our customers or users, or affect the profitability of our businesses, these are serious defects.

Do they matter to you?

The real question, however, is not whether defects matter in a theoretical sense but whether they matter to you. One way to think about this would be in terms of paying the costs of dealing with defects. Suppose you had to pay the discovery, recovery, reporting, repairing, redistribution, and reinstallation costs for every defect a customer reported for your programs. At IBM in my day, these costs averaged around $20,000 per valid unique defect. And there were thousands of these defects every year.

Of course, the problem is not who made the mistake but why it wasn't caught by the process. Thus, instead of tracing customer-discovered defects back to the engineers who injected them, we should concentrate on fixing the process. Regardless of their causes, however, defect costs can be substantial, and if you had to personally pay these costs, you would almost certainly take defects pretty seriously.

Why not just worry about the serious defects?

At this point, I think most people would agree that there are serious defects, and in fact that the reports of serious defects have been increasing. Now that we agree that some defects are serious, why not just worry about the few that are really serious? This question leads to a further question: Is there any way to differentiate between the serious defects and all the others? If we could do this, of course, we could just concentrate on the serious problems and continue to handle all the others as we do at present.

To identify the serious defects, however, we must differentiate them from all the others. Unfortunately, there is no evidence that this is possible. In my experience, some of the most trivial-seeming defects can have the most dramatic consequences. In one example, an executive of a major manufacturer told me that the three most expensive defects his organization had encountered were an omitted line of code, two characters interchanged in a name, and an incorrect initialization. These each caused literally millions of dollars of damage. In my work, some of my most troublesome defects were caused by trivial typing mistakes.

How many defects are serious?

Surprisingly, the seriousness of a defect does not relate to the seriousness of the mistake that caused it. While major design mistakes can have serious consequences, so can minor coding errors. So it appears that some percentage of all the defects we inject will likely have serious consequences. But just how many defects is this? And do we really need to worry that much about this presumably small number?

Based on my work with the Personal Software Process[SM] (PSP[SM]), experienced programmers inject one defect in about every 10 lines of code (LOC) [Humphrey 95]. While these numbers vary widely from engineer to engineer, and they include all the defects, even those found in desk checking or by the compiler, there are still lots of defects. And even for a given engineer, the numbers of defects will vary substantially from one program to the next. For example, when I have not written any programs for just a few weeks, I find that my error rate is substantially higher than it was when I was writing pretty much full time. I suspect this is true of other programmers as well. So engineers inject a large number of defects in their programs, even when they are very experienced.

Won't the compiler find them?

Now, even though there are lots of defects, engineers generally feel that the compiler will find all the trivial ones and that they just need to worry about the design mistakes. This, unfortunately, is not the case. Many of my programming mistakes were actually typing errors. Unfortunately, some of these mistakes produced syntax-like defects that were not flagged by the compiler. This was because some of my mistakes resulted in syntactically correct programs. For example, typing a ")" instead of a "}" in Pascal could extend a comment over a code fragment. Similarly, in C, typing "=" instead of "= =" can cause an assignment instead of a comparison. From my personal data, I found that in Pascal, 8.6% of my syntax defects were really syntax like, and in C++, this percentage was 9.4%.

Some syntax-like defects can have serious consequences and be hard to find. If, as in my case, there are about 50 syntax defects per 1000 lines of code (KLOC), and if about 10% of them are actually syntax like, then about 5 or so defects per KLOC of this type will not be detected during compilation. In addition there are 30 to 50 or so design-type defects that will not be detected during compilation. So we are left with a large number of defects, some of which are logical and some of which are entirely random. And these defects are sprinkled throughout our programs.

How about exhaustive testing?

Next, most engineers seem to think that testing will find their defects. First, it is an unfortunate fact that programs will run even when they have defects. In fact, they can have a lot of defects and still pass a lot of tests. To find even a large percentage of the defects in a program, we would have to test almost all the logical paths and conditions. And to find all of the defects in even small programs, we would have to run an exhaustive test.

To judge the practicality of doing this, I examined a small program of 59 LOC. I found that an exhaustive test would involve a total of 67 test cases, 368 path tests, and a total of 65,536 data values. And this was just for a 59 LOC program. This is obviously impractical. While this was just one small program, if you think that exhaustive testing is generally possible, I suggest you examine a few of your own small programs to see what an exhaustive test would involve. You will likely be surprised at what you find.

Just how many defects are there?

So now, assuming you agree that exhaustive testing is impossible, and that some of these defects are likely to be serious, what next? First, we find that the programs that engineers produce have a lot of defects, at least before compilation and unit test. Also, we find that compilation and unit testing cannot find all of the defects. So a modest percentage of the defects are left after unit testing. But how many are likely left, and do these few defects really matter?

Here again, the data are compelling. From PSP data, we find that engineers typically find between 20 to 40 defects per KLOC when they first test their programs. From my personal experience, I also find that unit testing typically finds only about 45% to 50% of the defects in a program. This means that after unit test, programs will typically still have around 20 or more defects per KLOC remaining. This is an extraordinary number. This means that after the first unit testing, programs will typically have a defect every 50 or so LOC! And after integration and product test, there would still be 5 to 10 or more defects left per KLOC. Remember, all of these defects must be found in system testing, or they will be left for the customer.

Consider some data

Just so you know that this is not an exaggeration, consider some data from the Jet Propulsion Laboratory (JPL). This organization develops the spacecraft that NASA sends to explore the solar system. One of their internal reports listed all the defects found in the system testing of three spacecraft [Nikora 91]. These software systems all had about 20 KLOC, and they were each tested for two or more years. The cumulative defects per KLOC found during this testing by week are shown in the figure. As you can see, they all had from 6.5 to nearly 9 defects per KLOC found in system test, and that does not guarantee that all the defects were found. In fact, from the figure, it seems pretty obvious that some defects remained. Note that these data were taken after the programs were developed, compiled, unit tested, and integration tested.

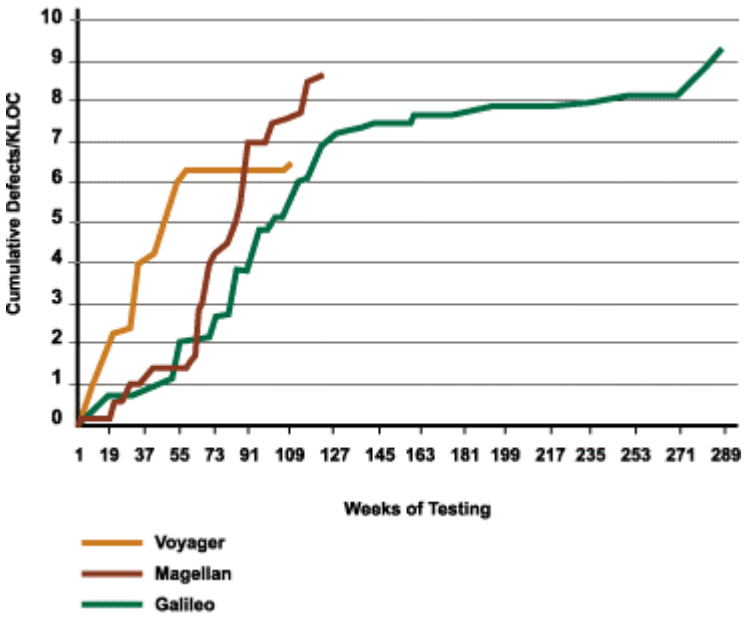


Figure 1: Spacecraft system test defects/KLOC

This, by the way, should not be taken as critical of JPL. Their programmers are highly skilled, but they followed traditional software practices. Normal practice is for programmers to first produce the code and then find and fix the defects while compiling and testing.

But why not call them "bugs"?

So, by now, you presumably agree that some defects are serious, and that there is no way to tell in advance which ones will be serious. Also, you will probably agree that there are quite a few of these defects, at least in programs that have been produced by traditional means. So now, does it matter what we call these defects? My contention is that we should stop using the term "bug." This has an unfortunate connotation of merely being an annoyance; something you could swat away or ignore with minor discomfort.

By calling them "bugs," you tend to minimize a potentially serious problem. When you have finished testing a program, for example, and say that it only has a few bugs left, people tend to think that is probably okay. Suppose, however, you used the term "land mines" to refer to these latent defects. Then, when you have finished testing the program, and say it only has a few land mines left, people might not be so relaxed. The point is that program defects are more like land mines than insects. While they may not all explode, and some of them might be duds, when you come across a few of them, they could be fatal, or at least very damaging.

We think in language

When the words we use imply that something is insignificant, we tend to treat it that way, at least if we don't know any better. For example, programmers view the mistakes they make as minor details that will be quickly found by the compiler or a few tests. If all software defects were minor, this would not be a problem. Then we could happily fix the ones we stumble across and leave the rest to be fixed when the users find them. But by now you presumably agree that some software defects are serious, that there is no way to know in advance which defects will be serious, and that there are growing numbers of these serious defects. I hope you also agree that we should stop referring to software defects as "bugs."

Finally, now that we agree, you might ask if there is anything we can do about all these defects? While that is a good question, I will address it in later columns. It should be no surprise, however, that the answer will involve the PSP and TSP[SM] (Team Software Process[SM]) [Webb 99].

Acknowledgements

In writing papers and columns, I make a practice of asking associates to review early drafts. For this column, I particularly appreciate the helpful suggestions of Dan Burton, Bob Cannon, Sholom Cohen, Frank Gmeindl, and Bill Peterson.

In closing, an invitation to readers

In these columns, I discuss software issues and the impact of quality and process on engineers and their organizations. I am, however, most interested in addressing issues you feel are important. So please drop me a note with your comments, questions, or suggestions. I will read your notes and consider them when I plan future columns.

Thanks for your attention and please stay tuned in.

Watts S. Humphrey
watts@sei.cmu.edu

References

[Humphrey 95] Humphrey, Watts S. *A Discipline for Software Engineering*. Reading, Ma.: Addison Wesley, 1995.

[Nikora 91] Nikora, Allen P. *Error Discovery Rate by Severity Category and Time to Repair Software Failures for Three JPL Flight Projects.* Software Product Assurance Section, Jet Propulsion Laboratory, 4800 Oak Grove Drive, Pasadena, CA 91109-8099, November 5, 1991.

[Webb 99] Webb, Dave & Humphrey, W.S. "Using the TSP on the TaskView Project," *CROSSTALK*, February, 1999.

About the author

Watts S. Humphrey founded the Software Process Program at the SEI. He is a fellow of the institute and is a research scientist on its staff. From 1959 to 1986, he was associated with IBM Corporation, where he was director of programming quality and process. His publications include many technical papers and six books. His most recent books are: *Managing the Software Process* (1989), *A Discipline for Software Engineering* (1995), *Managing Technical People* (1996), *and Introduction to the Personal Software Process*SM(1997). He holds five U.S. patents. He is a member of the Association for Computing Machinery, a fellow of the Institute for Electrical and Electronics Engineers, and a past member of the Malcolm Baldridge National Quality Award Board of Examiners. He holds a BS in physics from the University of Chicago, an MS in physics from the Illinois Institute of Technology, and an MBA from the University of Chicago.

The views expressed in this article are the author's only and do not represent directly or imply any official position or view of the Software Engineering Institute or Carnegie Mellon University. This article is intended to stimulate further discussion about this topic.

The Software Engineering Institute (SEI) is a federally funded research and development center sponsored by the U.S. Department of Defense and operated by Carnegie Mellon University.