Watts New

A Personal Quality Strategy

Watts S. Humphrey

This is the fourth column in a four-column series on software quality and security. The first column discussed the software quality problem, why customers don't care about quality, and how bad software quality really is. The second column described why testing alone cannot produce much higher quality software than we get today. In the third column, I described the quality attitude and how software professionals and their managers must change their view of quality if we are to make much headway improving software quality and security. This column discusses the stages of quality and strategies for addressing the quality problems at each stage. It also describes practices to consider as you improve your personal performance as a software developer. Finally, I comment on the challenges ahead and how the strategies described here can help you to address them.

Quality Stages

There are many ways to look at quality, but, from a software development perspective, we can think about quality as having five stages.

Stage 1 – Basic code quality: syntax and coding constructs

Stage 2 – Detailed design: the logical construction of programs and the actions required so that these programs perform their specified functions

Stage 3 – High-level design: system issues such as interfaces, compatibility, performance, security, and safety

Stage 4 – Requirements focused: determining the meaning of the requirements and particularly deducing what is written between the lines.

Stage 5 – User driven: users and what we must do to provide them with truly great products. While last in this list, user concerns must have top priority.

Stage 1 – Basic Code Quality

At the stage of basic code quality, quality is personal. Either you are fluent in the programming language or you are not. If you are fluent, your principal concerns are typos and occasional obvious mistakes. I once tracked my errors for a large data-entry job and found that I made 1.74 errors per thousand keystrokes, or 2.4 errors per hour of key entry. Since a line of my C++ code averages 17 keystrokes, my key-entry error rate alone injected 29.5 defects per 1,000 lines of code.

This was just typing. I did not consider program semantics, functions, or design. To correct this type of basic mistake, you need to track the defects and understand the errors that caused them. Until you do, you cannot prevent these types of defects or get better at finding and fixing them. Many developers object to tracking defects that the compiler could quickly find. The reason to record these defects, however, is to understand the mistakes you make no matter how they are found. Once you have defect data, analyze it. By recording your defects and analyzing the data, you are likely to cut your defect-injection rate by about 50%. Also, keep recording the defect data. If you don't, your injection rate will creep back up.

If you are not yet fluent with the programming language, take the same three steps: track your defects, analyze the data, and continue doing this as a regular part of your programming work. By tracking and analyzing my defects and using the defect data to guide my personal design and code reviews, I substantially cut the time it took me to learn a new programming language and also dramatically improved the quality of my products.

The obvious next question is, "Why bother with these Stage 1 defects, since the compiler will find most of them anyway?" The brief answer is that tools and testing will not find a significant number of your defects, and any that you miss will be very expensive for you, the testers, or the users to find and fix later. (For a more complete answer, consult my previous three columns mentioned above.) If you don't clean up the Stage 1 defects first, they will make it harder for you and your teammates to find and fix the more sophisticated defects at the higher quality stages. You will also waste a lot of test time fixing defects that you could have quickly found and fixed beforehand.

Stage 2 – Detailed-Design Quality

At the detailed-design stage, the problems are more sophisticated. Most programmers make design mistakes not because they don't know how to design, but because they never actually produced a design. They may have drawn some bubble charts or sketched a few use cases, but they never reduced these designs to a specification for writing code.

The practice of designing while coding is error prone. From data on 3,240 programs written in Personal Software Process (PSP)[SM] courses, the SEI has found that experienced developers inject fewer defects when designing (2.0 defects per hour) than when they design while coding (4.6 defects per hour). If you want low-defect designs, you must produce those designs, instead of just creating them while coding.

There are two big advantages to producing designs. First, you will make less than half as many design mistakes, and second, you will have a design that you can review and correct. However, in doing the design review, use good review methods. This is too big a subject to cover here, but it is covered in my new book *The Personal Software Process – A Discipline for Software Engineers* to be published in March 2005.

[SM] PSP and Personal Software Process are service marks of Carnegie Mellon University.

To do design reviews properly, you must spend enough time doing them. From the data on the same 3,240 PSP programs, the average defect-removal rate during design reviews was 3.3 defects per hour. Therefore, if you inject defects at the rate of 2.0 per hour and remove them at 3.3 per hour, you must spend about 36 minutes reviewing the design for every hour you spent producing it.

The detailed-design stage is particularly important because this is where you design the logic paths that must be tested. If some of the paths through your module have subtle defects and if you don't find and fix them, they will be left for the testers or users to find. Since you presumably will have tested your program before passing it on to test, the remaining defects will be the ones that did not show up in your testing. In other words, these defects do not prevent the program from working except under specialized conditions. To find them, the test must cover the right paths and it must have the proper parameter or variable values.

Assuming that your programs have about the same proportion of branch instructions as my C++ programs, your typical 500 line-of-code (LOC) module would have about 50 branch instructions. From the data in Table 1, your program would then have about 3,400 possible test paths, any one of which could contain a design defect that you missed. (For more on the testing maze, see my previous column in news@sei.) That is why the strategy of testing in quality takes so long and produces defective products.

Table 1. Testing Paths

Maze Size	Branches	Paths	Maze Size	Branches	Paths
1	1	2	11	121	705,432
2	4	6	12	144	2,704,156
3	9	20	13	169	10,400,600
4	16	70	14	196	40,116,600
5	25	252	15	225	1.55E+08
6	36	924	16	256	6.01E+08
7	49	3,432	17	289	2.33E+09
8	64	12,870	18	324	9.08E+09
9	81	48,620	19	361	3.53E+10
10	100	184,756	20	400	1.38E+11

The key practices at Stage 2, the detailed-design stage, are to produce complete designs, review the designs yourself, and have your teammates carefully inspect them. Then, of course, implement the program and thoroughly review, inspect, and test it to make sure that the code properly reflects the design. If you

and your teammates do this for every module you and your team develop, you will improve the quality of your programs by at least 10 times and probably much more. You will also save a lot of test time.

Stage 3 – High-Level Design Quality

Through Stage 2, the defects are yours. You inject them, and you are the best person to find and fix them. Above this stage, you must work with others. The defects at Stage 3 concern the interfaces, interdependencies, and interactions of your program with the other parts of the system. Defects in any module could affect system properties such as performance, security, and safety. These properties result from the correct operation of all or most of the parts of the system. This is another case where sound design practices are important. For security, for example, a properly-produced high-level design would specify the authentication practices and the data-security and protection conventions that the modules should follow.

At Stage 3, the key quality practices are first, to get and review the high-level design specifications for your module. If these specifications don't exist or are inadequate, work with your teammates and system designers to clarify the design specifications. Second, after you obtain these specifications, follow them in producing the module design. Then, third, review the design to ensure that it meets these specifications and that it will work with all of the other modules. If you don't do this, there is a good chance that the modules you develop will not work properly with the rest of the system. Often, such problems cannot be fixed without a major module redesign or a complete module replacement. Finally, get your team's help in thoroughly inspecting and correcting the design. This is particularly important because, at this stage, your products begin to address system-level issues that you may not even be aware of.

Stage 4 – Requirements-Focused Quality

At Stages 1, 2, and 3, you work with familiar issues. At the requirements stage, however, you are no longer an expert. The two principal difficulties with requirements problems are, first, that requirements are not your field of specialty. Therefore, it is easy to think that you understand something when you do not. Second, misunderstandings and errors in requirements interpretation can easily cause you to build the wrong product.

If not caught early, requirements misunderstandings can be fatal. This is why you should start with a thorough examination of the requirements and resolve any confusion or uncertainty with requirements experts before starting to design the product. While many requirements details need not be settled at this point, it is hard to know what is a detail and what is fundamental. Once you are familiar with the requirements, settle the critical points before starting the design work and resolve the remaining issues in parallel with the design and implementation work. The two key practices at Stage 4 are to understand the requirements and to resolve any confusion or uncertainty before starting on the design. In implementing these practices, get your team's help as well as the help of the organization's systems designers or requirements experts. Some of these people have been thinking about this requirement for a long time and will likely understand the user's needs better than you possibly could with just a few brief weeks or months of exposure.

Stage 5 – User-Driven Quality

The user stage is a totally different ballgame. Here, as shown in Figure 1, the problems are not with what you know or even what you don't know, they are with what you don't know that you don't know. This problem is particularly tricky because the users often don't understand the issues either. A truly great product must perform a desirable user function in a convenient and elegant way. While the users may think that they know what they need and have strong opinions on how the product should work, they will often be wrong. They won't completely understand or be able to specify the functions that they want, and their view of how to build a product to perform these functions will approximate what they do today, rather than some elegant new approach.

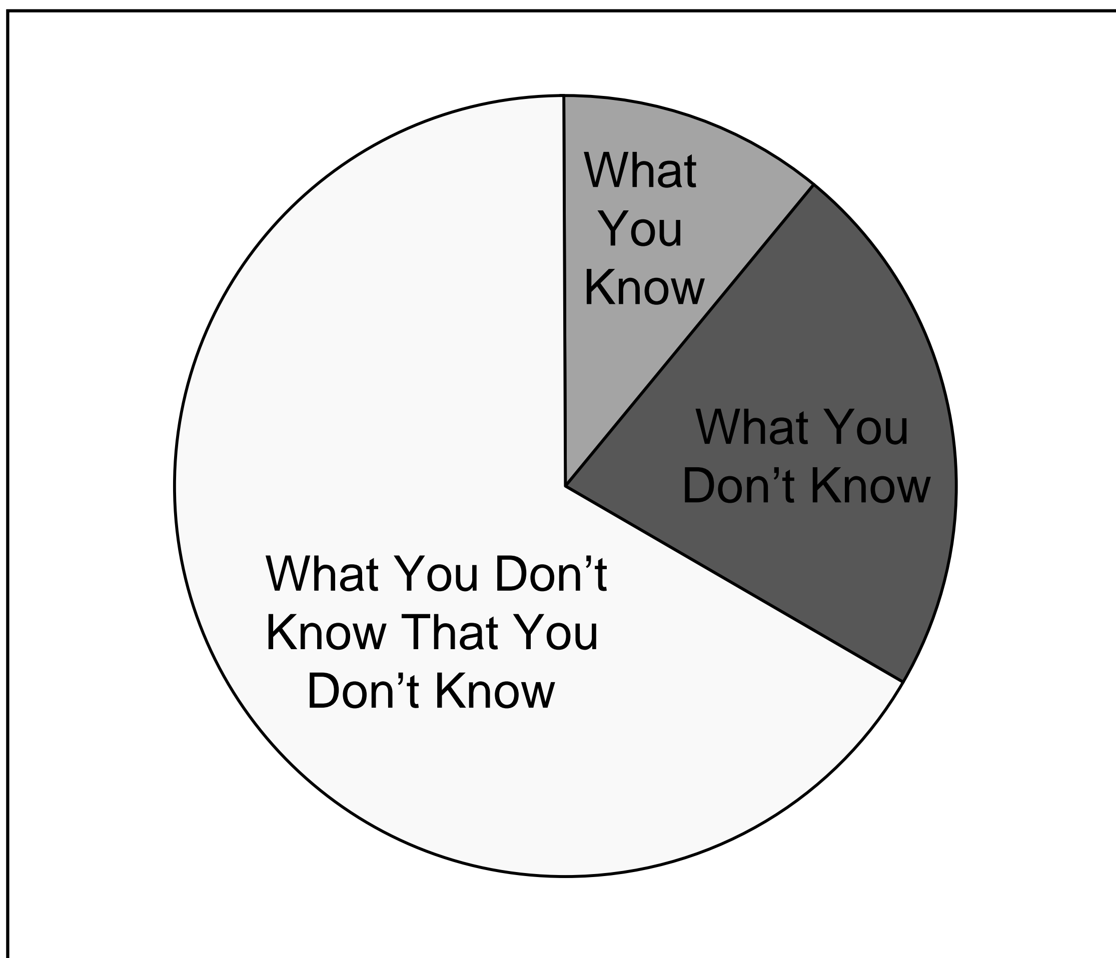

Figure 1. What We Know and Don't Know

The challenge at the user stage is to get a clear understanding of what the users think they want and then to develop an intuitive sense for what they really need. Once you do this, keep thinking about the problem and you could come up with a truly creative solution. This is how breakthroughs like the spreadsheet or mouse were conceived. A developer who understood the problem had a “crazy” idea that worked. While these creative leaps don’t happen often, preparing for them will help you to build a fine product. You will have solved a problem in a better way than you otherwise would have, and you will have given yourself the chance to do something great.

What makes the user-driven stage so different is that elegant solutions often change the problem, and sometimes they change it in fundamental ways. Your objective at this stage should be to develop a deep enough intuitive understanding of the user’s needs so you can see the opportunities for dramatic departures that will transform the problem while enabling a breakthrough solution.

Putting It All Together

Quality is an individual issue. If you really want to do great work, there are lots of ways to do it. You may have to settle for incomplete requirements and specifications, and you may not even be able to talk to any users, but you can always do a first-class job at Stages 1, 2, and 3. Keep thinking about the higher quality stages and what you can do to extend yourself and your team to do great work. You may not succeed often, but it is worth the try. Major advances take time, a lot of insight, and some perspiration. They also take an occasional inspiration. Even if it takes many years, a great product is something that you will always be proud of. So keep trying to do quality work. It will make your life much more rewarding.

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 comments and suggestions of Dan Burton, Jim McHale, Bill Peterson, Marsha Pomeroy-Huff, and Dan Wall.

In closing, an invitation to readers

In these columns, I discuss software issues and the impact of quality and process on developers and their organizations. However, I am most interested in addressing the issues that 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 planning future columns.

Thanks for your attention and please stay tuned in.

Watts S. Humphrey
watts@sei.cmu.edu

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 several books. His most recent books are *Introduction to the Team Software Process* (2000) and *Winning With Software: An Executive Strategy* (2002). 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 Baldrige 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.