Watts New

Some Programming Principles - Requirements

Watts S. Humphrey

In this and the next several columns, I discuss some principles for programming work. These are principles that, when followed, will consistently produce superior software that meets the needs of customers and businesses. This column concentrates on the principles that are inherent in software work because of the nature of software products and their requirements. These principles also concern the characteristics of the people who use these products. These programming principles are as follows:

• When we program, we transform a poorly understood problem into a precise set of instructions that can be executed by a computer.

• When we think we understand a program’s requirements, we are invariably wrong.

• When we do not completely understand a problem, we must research it until we know that we understand it.

• Only when we truly understand a problem can we develop a superior product to address that problem.

• What the users think they want will change as soon as they see what we develop.

The Programming Job

As any experienced programmer knows, it is hard for people to be absolutely and completely precise about anything. However, to produce a usable program, we must specify exactly what the program is to do under every possible circumstance. The difficulty of being precise is brought home to me whenever someone gives me directions.

“Go three blocks, turn left at the gas station, then take the third street on the right.”

“But,” I interrupt, “precisely where do I start and in what direction should I face?”

For some reason, the questions that I must ask to get precise directions always seem to annoy people. Why should this annoy them? Don’t they know that, without precise and complete information, I might get lost and waste a great deal of time?
The answer is, no they do not. In fact, even when they know exactly what they want done, most people are unable to tell you precisely what to do. What is worse, they will even get annoyed when you press them for the required details. This attitude complicates the programmer’s job. To work properly, computing systems must be given absolutely precise instructions. Therefore, to write any program, the programmer must reduce that problem to a precise form before it can be executed by a computer. This means that we must somehow persuade one or more knowledgeable users to explain all of the problem’s details. In summary, there are four reasons why this is hard.

- First, the users will not know how to be precise about their needs.
- Second, they will get annoyed when we press them to be precise.
- Third, we will often think that we understand the problem before we really do.
- Fourth, the users will often misunderstand or not even know what they need.

Programming is and always will be a precise intellectual activity that must be performed by people. While machines can help us in this work, they can never figure out what we need, so they can never replace us. The problem is that people are error prone and programs are extraordinarily sensitive to errors. The programming challenge we face is to devise processes and methods that will help us to produce precise intellectual products that are essentially defect-free.

**Understanding the Problem**

The difference between thinking that you understand a problem and truly understanding it is like night and day. It is amazing how often I think that I know something but then find that I really do not. My mother used to explain that being positive was “being wrong at the top of your voice.” When it matters, like when writing a program, preparing a talk, or drafting a paper for publication, I try to prove that what I think is true really is true. When I look at data, consult a reference, or talk to an expert, I often find that my initial opinion was either wrong or too simplistic. The real story is invariably much more interesting and often more complex than I had initially realized.

It is easy to settle for half-baked answers. Some years ago, a programming manager told me about a conversation that she overheard between two of her programmers. They were developing a new version of IBM’s COBOL compiler and were debating a particular feature. One of them felt that the users would prefer one approach and the other felt that a different format would be more convenient. This manager told me that, even though
neither of them really knew which approach would be best, they ended up agreeing on some middle ground that they both thought would be OK.

On any large product, there are hundreds to thousands of these minor decisions and, when programmers don’t really know the proper or most convenient or most efficient answer, they usually guess. Even if the odds of being right are 50-50, they will make hundreds to thousands of incorrect decisions. Then, the odds of their producing a convenient and highly-usable product will be essentially zero.

Researching Problems

There are lots of ways to research problems. For example, I often write about how people develop programs and the costs and benefits of various programming practices. Ever since I started my research work on personal and team programming methods over 13 years ago, I have gathered data about my own and other people’s work. I now have a database of over 11,000 programs that I regularly use to verify some opinion or to answer some question. Depending on the question or problem, data can be very helpful.

The second approach is to ask someone who knows the answer. The problem here is that, like the COBOL programmers, you may not have an expert on hand and will often settle for just getting a second opinion. While it might make you feel better to have someone agree with your guess, it is still a guess. You can be pretty sure that it will either be wrong or a simplistic approximation of the true situation.

Often, we cannot find a convenient expert and the time required to get expert help seems likely to delay the project. However, this is only an excuse. Once you build all of your guesses into a program’s design, the costs of fixing it later will be vastly greater than any conceivable delay. So bite the bullet—take the time to get the facts needed to do a truly superior job. It will invariably pay off.

Getting Informed Input

Some years ago, one of the projects in my organization was developing a very large programming product. I was convinced that usability was a key need and kept pushing the project managers to have a panel of experts review their designs. The managers kept telling me that it was too early to hold such a review. They argued that they had not yet completed enough of the design work. Then, one day, they told me that it was too late. It would now be too expensive to make any changes. I insisted, however, and we held a two-day review.
I sat in on the meetings with a panel of a dozen user experts. The programming managers could not answer several of the more detailed operational questions and had to call their programmers for help. At the end of two days, these managers had agreed to make half a dozen changes before they shipped the first release. They now agreed that, without these changes, the product would have been unusable. After I moved on to a different position, this management team never held another expert review. Since these were very smart people, I concluded that they must not have appreciated the enormous importance of getting informed user input.

There are many ways to research requirements questions. Sometimes you will need experts but often a simple prototype and one trial use will tell you the answer. You can also get associates or even non-programmer support people to try to use a prototype. By observing their reactions, you can resolve many usability issues. In other cases, you can accumulate several questions and build a prototype to try out the most promising alternatives. Then you can often get some experts to test the prototypes and to give you their views. In any event, keep track of your assumptions and guesses, and verify them as soon as you can.

**Building Superior Products**

During my time as IBM’s Director of Programming, I received a constant stream of requests to incorporate one or another field-developed program into the IBM product line. These programs were developed by IBM’s customer support personnel to help their accounts solve critical problems. We did accept a few of these programs and, in a few years, these few field-developed programs were consistently at the top of our customer satisfaction ratings. Their ratings were well above those for the standard products that had been developed in our laboratories.

In contrast to the company’s “standard” products, these field-developed programs had been designed by people who were intimately familiar with the users’ environment. These developers intuitively understood precisely how the product should work and didn’t have to guess what would be more convenient. They knew. If you want to develop a superior product, either become intimately familiar with the user’s environment or have someone with that familiarity readily available to answer your questions. You will need to consult them almost every day.

**Changing Problems and Products**

If expert advice were all that was needed, requirements would not be such a serious problem. However, there are three issues that complicate the story. First, the users will
not be able to tell you what they want. Generally, these are not development people and they cannot visualize solutions to their problems. They could tell you how they work right now but they will have no idea how they would work with a new product, even if its designed were based on exactly how they work today. The second problem is that few users understand the essence of the jobs they are doing. By that, I mean that they do not understand the fundamentals of their jobs, so they cannot help you to devise a product solution that addresses the real operational needs. All they can usually do is help you define how to mechanize the often manual system that they are currently using. The third problem is a kind of uncertainty principle. By introducing a new product, you actually change that user’s environment. Often, this will completely change the way the job should be done and it could completely obsolete the original requirements.

What these three problems tell us is that your initial definition of the requirements will almost certainly be wrong. While you must get as close as possible to a usable solution, you must recognize that the requirements are a moving target that you must approach incrementally. To produce a truly superior and highly-usable product for a realistic cost and on a reasonable schedule, you must follow a development strategy and process that assumes that the requirements will change. This requires that you encourage early changes and that you develop and test the program in the smallest practical increments. Also, if possible, get the users to participate in the early testing. Then, when you find what you did wrong or what your customers could not explain or what the users did not know, the amount of rework will be small enough to do in a reasonable time. Then you can quickly take another user checkpoint to make sure you are still on the right track.

These few principles are fundamental to just about every programming job. What is most interesting is that experienced programmers will generally recognize these principles as correct, even while they work in an environment that does not apply them. The challenge that we programmers face is to convince ourselves, our managers, our executives, and even our customers to devise processes, to plan projects, and to manage our work in a way that is consistent with these principles. While it might seem simple and easy to convince people to do something so obvious, it is not. There are many reasons for this, but the most basic reason is that welcoming early requirements changes exposes our projects to unlimited job growth, which has historically led to serious schedule overruns. These are problems that I will address in the next few columns where I discuss the principles for designing products, guiding projects, leading teams, and training and coaching the people who do programming work.

**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

**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. 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 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℠* (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 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.

© Capability Maturity Model, Capability Maturity Modeling, Carnegie Mellon, CERT, CERT Coordination Center, CMM, and CMMI are registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

SM Architecture Tradeoff Analysis Method; ATAM; CMM Integration; COTS Usage Risk Evaluation; CURE; EPIC; Evolutionary Process for Integrating COTS Based Systems; Framework for Software Product Line Practice; IDEAL; Interim Profile; OAR; OCTAVE; Operationally Critical Threat, Asset, and Vulnerability Evaluation; Options Analysis for Reengineering; Personal Software Process; PLTP; Product Line Technical Probe; PSP; SCAMPI; SCAMPI Lead Assessor; SCAMPI Lead Appraiser; SCE; SEI; SEI-Europe; SEPG; Team Software Process; and TSP are service marks of Carnegie Mellon University.

TM Simplex is a trademark of Carnegie Mellon University.