
*Alan M. Davis, Editor*

# FIFTEEN PRINCIPLES OF SOFTWARE ENGINEERING

**How to get people and technology to work together.**

*This column is an excerpt from my book,* 201 Principles of Software Engineering, *to be published later this year by McGraw-Hill. Here I can only highlight the 15 most important principles that underlie software engineering, then list the next 15 in the box on p. 96.*

OTHER ENGINEERING DISCIPLINES have principles based on the laws of physics, biology, chemistry, or mathematics. Principles are rules to live by; they represent the collected wisdom of many dozens of people who have learned through experience. A discipline's principles evolve as the discipline grows: Existing principles are modified; new ones added; old ones discarded. Practice, and experience gained through practice, are the basis for the evolution of principles.

**PHYSICAL LAWS DON'T WORK FOR SOFTWARE: OUR PRACTICES ARE BASED ON EXPERIENCE.**

Because the product of software engineering is not physical, physical laws do not form a suitable foundation. Instead, software engineering has had to evolve its principles based solely on observations of thousands of projects. If we were to examine software engineering's principles from 1964 ("always use short variable names;" "do whatever it takes to make your program smaller"), they would look downright silly. Today's principles will look equally silly in 30 years. Nevertheless, I think it's important to describe what our principles are today. The following are probably the 15 most important.

**1. MAKE QUALITY NUMBER 1.** A customer will not tolerate a poor-quality product, regardless of how you define "quality." Quality must be quantified and mechanisms put into place to motivate and reward its achievement. It may seem politically correct to deliver a product on time, even though its quality is poor, but this is correct only in the short term; it is suicide in the middle and long term. There is no trade-off to be made here. The first requirement *must* be quality.

However, there is no one definition of software quality. To developers, it might be elegant design or elegant code. To users, it might be good response time or high capacity. For cost-conscious managers, it might be low development cost. For some customers, it might be satisfying all their perceived *and* not-yet-perceived needs. The dilemma is that these definitions may not be compatible.

**2. HIGH-QUALITY SOFTWARE IS POSSIBLE.** Although our industry is saturated with examples of software systems that perform poorly, are full of bugs, or otherwise fail to satisfy user needs, there are counterexamples. Large software systems *can* be built with very high quality, but they carry a steep price tag — on the order of $1,000 per line of code. One example is IBM's on-board flight software for the space shuttle: three million lines of code with less than one error per 10,000 lines.

Techniques that have been demonstrated to increase quality considerably include involving the customer, prototyping (to verify requirements before full-scale development), simplifying design, conducting inspections, and hiring the best people.

**3. GIVE PRODUCTS TO CUSTOMERS EARLY.** No matter how hard you try to learn users' needs during the requirements phase, the most effective way to ascertain *real* needs is to give users a product and let them play with it. The conventional waterfall model delivers the first product after 99 percent of the development resources have been expended. Thus, the majority of customer feedback on need occurs after resources are expended. Contrast this with an approach that has you deliver a quick-and-dirty prototype early in development, gather feedback, write a requirements specification, and then proceed with full-scale development. In this scenario, only five to 20 percent of development resources have been expended when customers first see the product.

Editors: Alan Davis
University of Colorado
1867 Austin Bluffs Pkwy., Suite 200
Colorado Springs, CO 80933-7150
adavis@vivaldi.uccs.edu

Winston Royce
TRW
1 Federal Systems Park Dr.
STC 7165U
Fairfax, VA 22030
(703) 803-5025/6
fax: (703) 803-5108

**4. DETERMINE THE PROBLEM BEFORE WRITING THE REQUIREMENTS.** When faced with what they believe is a problem, most engineers rush to offer a solution. If the engineer's perception of the problem is accurate, the solution *may* work. However, problems are often elusive. In *Are Your Lights On?* (Dorset House, 1990), Donald Gause and Gerald Weinberg describe a "problem" in a high-rise office building. The occupants complain of long waits for an elevator. Is this really the problem? And whose problem is it? From the occupants' perspective, the problem might be that the wait is a waste of time. From the building owner's perspective, the problem might be that long waits will reduce occupancy (and thus rental income). The obvious solution is to increase the speed of the elevators. But you could also add elevators, stagger working hours, reserve some elevators for express service, increase the rent, or refine the "homing algorithm" so elevators go to high-demand floors when they are idle. The range of costs, risks, and time associated with these solutions is enormous. Yet any one *could* work, depending on the situation. Before you try to solve a problem, be sure to explore all the alternatives and don't be blinded by the "obvious" solution.

**5. EVALUATE DESIGN ALTERNATIVES.** After the requirements are agreed upon, you *must* examine a variety of architectures and algorithms. You certainly do not want to use an "architecture" simply because it was used in the requirements specification. After all, that "architecture" was selected to optimize the understandability of the system's external behavior. The architecture you want is the one that optimizes conformance with the requirements.

For example, architectures are generally selected to optimize constructability, throughput, response time, modifiability, portability, interoperability, safety, and availability, while also satisfying the functional requirements. The best way to do this is to enumerate a variety of software architectures, analyze (or simulate) each with respect to the goals, and select the best alternative. Some design methods result in specific architectures, so one way to generate a variety of architectures is to use a variety of methods.

**6. USE AN APPROPRIATE PROCESS MODEL.** There are dozens of process models: waterfall, throwaway prototyping, incremental, spiral, operational prototyping, and so on. There is no such thing as a process model that works for every project. Each project must select a process that makes the most sense for that project, on the basis of corporate culture, willingness to take risks, application area, volatility of requirements, and the extent to which requirements are well-understood.

Study your project's characteristics and select a process model that makes the most sense. When building a prototype, for example, choose a process that minimizes protocol, facilitates rapid development and does not worry about checks and balances. Choose the opposite when building a life-critical product.

**7. USE DIFFERENT LANGUAGES FOR DIFFERENT PHASES.** Our industry's eternal

## FIFTEEN MORE SOFTWARE PRINCIPLES

**16. UNDERSTAND THE CUSTOMERS' PRIORITIES.** It is possible the customer would tolerate 90 percent of functionality delivered late if they could just have 10 percent of it on time. Find out!

**17. THE MORE THEY SEE, THE MORE THEY NEED.** The more functionality (or performance) you provide a user, the more functionality (or performance) the user wants.

**18. PLAN TO THROW ONE AWAY.** One of the most important critical success factors is whether or not a product is entirely new. Such brand-new applications, architectures, interfaces, or algorithms rarely work the first time.

**19. DESIGN FOR CHANGE.** The architectures, components, and specification techniques you use must accommodate major and incessant change.

**20. DESIGN WITHOUT DOCUMENTATION IS *NOT* DESIGN.** I have often heard software engineers say "I have finished the design. All that's left is its documentation." Can you imagine a building architect saying "I have completed the design of your new home. All that's left is to draw a picture of it"?

**21. USE TOOLS, BUT BE REALISTIC.** Software tools make their users more efficient. By all means, use them. Just as a word processor is essential to a writer, a CASE tool is essential to a software engineer.

**22. AVOID TRICKS.** Many programmers love to create programs with tricks — constructs that perform a function correctly, but in an obscure way. Show the world how smart you are by avoiding tricky code.

**23. ENCAPSULATE.** Information-hiding is a simple, proven concept that results in software that is easier to test and much easier to maintain.

**24. USE COUPLING AND COHESION.** Coupling and cohesion are the best ways to measure software's inherent maintainability and adaptability. Learn them. Use them to guide your design decisions. (Larry Constantine and Edward Yourdon, *Structured Design*, Prentice-Hall, 1979).

**25. USE MCCABE COMPLEXITY MEASURE.** Although there are many metrics available to report the inherent complexity of software, none is as intuitive and easy-to-use as Tom McCabe's ("A Complexity Measure," *IEEE Trans. Software Eng.*, Dec. 1976, pp. 308-320).

**26. DON'T TEST YOUR OWN SOFTWARE.** Software developers should never be the primary testers of their own software.

**27. ANALYZE CAUSES FOR ERRORS.** It is far more cost-effective to reduce the effect of an error by preventing it than it is to find and fix it. One way to do this is to analyze the causes of errors as they are detected.

**28. REALIZE THAT SOFTWARE'S ENTROPY INCREASES.** Any software system that undergoes continuous change will grow in complexity and will become more and more disorganized.

**29. PEOPLE AND TIME ARE NOT INTERCHANGEABLE.** Measuring a project solely by person-months makes little sense. If a project can be completed in one year by six people, does that mean 72 people could complete it in one month? Of course not!

**30. EXPECT EXCELLENCE.** Your employees will do much better if you have high expectations for them.

thirst for simple solutions to complex problems has driven many to declare that the best development method is one that uses the same notation throughout the life cycle. This is not the practice in any other engineering discipline. Electrical engineers use different notations for different design activities: Block diagrams, circuit diagrams, logic diagrams, timing diagrams, state-transition tables, check plots, stick diagrams, and so on. Why should software engineers use, say, Ada for requirements, design, and code unless Ada was optimal for all these phases? Why should they use object orientation for all phases unless it was optimal for all phases? Select a set of techniques and languages that is best for the phase you are working in. The transitions between phases *are* difficult, but using the same language doesn't help. On the other hand, if a language is optimal for certain aspects of two phases, by all means use it.

**8. MINIMIZE INTELLECTUAL DISTANCE.** Edsger Dijkstra defined *intellectual distance* as the distance between the real-world problem and the computerized solution to the problem. Richard Fairley has argued that the smaller the intellectual distance, the easier it is to maintain the software. To minimize intellectual distance, the software's structure should be as close as possible to the real-world structure. This is the primary motivation for approaches such as object-oriented design and Jackson System Development. But you can minimize intellectual distance using *any* design approach. Of course, the real-world structure can vary, as Jawed Siddiqi points out ("Challenging Universal Truths of Requirements Engineering," Mar. 1994, pp. 18-19). Different humans perceive different structures when they examine the same real world and thus construct quite different "realities."

**9. PUT TECHNIQUE BEFORE TOOLS.** An undisciplined carpenter with a power tool becomes a dangerous undisciplined carpenter. An undisciplined software engineer with a tool becomes a dangerous undisciplined software engineer. Before you use a tool, you should understand and be able to follow an appropriate software technique. Of course, you also need to know how to use the tool, but that is secondary to having discipline.

**10. GET IT RIGHT BEFORE YOU MAKE IT FASTER.** It is far easier to make a working program run faster than to make a fast program work. Don't worry about optimization during initial coding (but don't use a ridiculously inefficient algorithm or data structure, either). Every software project has tough schedule pressures. Given this situation, any time a component is produced on (or ahead of) time and it works reliably, there is cause for celebration. Try to be the reason for celebration rather than desperation. If your program works, everyone on your team will appreciate it.

**11. INSPECT CODE.** Inspecting the detailed design and code, first proposed by Michael Fagan (*IBM Systems Journal*, July 1976), is a *much* better way to find errors than testing. Inspection can find as many as 82 percent of all errors, consumes about 15 percent of development resources, and reduces net development costs by 25 to 30 percent. Your schedule should account for time to inspect (and correct) every component. You might think *your* project cannot tolerate such a "luxury." However, data shows that inspections can reduce time-to-test by 50 to 90 percent (Tom Gilb and Dorothy Graham, *Software Inspections*, Addison-Wesley, 1993). If that's not incentive, I don't know what is.

**12. GOOD MANAGEMENT IS MORE IMPORTANT THAN GOOD TECHNOLOGY.** The best technology will not compensate for poor management, and a good manager can produce great results even with meager resources. Successful software start-ups are not successful because they have great process or great tools (or great products for that matter!). Most are successful because of great management and great marketing.

Good management motivates people to do their best, but there are no universal "right" styles of management. Management style must be adapted to the situation. It is not uncommon for a successful leader to be an autocrat in one situation and a consensus-based leader in another. Some styles are innate; others *can* be learned.

**13. PEOPLE ARE THE KEY TO SUCCESS.** Highly skilled people with appropriate experience, talent, and training are key. The right people with insufficient tools, languages, and process *will* succeed. The wrong people with appropriate tools, languages and process will probably fail (as will the right people with insufficient training or experience). When interview-

*Continued on page 101*

## LOGGING OFF

After more than two years of sharing the editing responsibilities for this department, Win Royce and I are stepping down. Win will continue to serve as an industry adviser, and I will take on the challenge of being editor-in-chief. In our place will be Roger Pressman, who owns R.S. Pressman and Associates, a consulting firm specializing in software engineering methods and training.

Roger's *Software Engineering: A Practitioner's Guide* is the world's most widely used software-engineering textbook. He has written many technical papers and five other books. *A Manager's Guide to Software Engineering* (McGraw Hill, 1993), his most recent, presents management guidelines for instituting new technology.

Roger will encourage columns that take on the conventional wisdom and stress pragmatic discussions of real-world software-management, process, and technology problems. He will challenge you to think about trends, question hype, and appreciate what it really means when we say there is no silver bullet. I'm sure you'll find these pages filled with lively discussion.

*—Al Davis*

painstaking and time-consuming task, I realize the importance of the data."

Another student noted an improvement in his process: "My initial process was informally defined and very undisciplined. My goal was to get the program coded, do half the design during code, and see what happens. The course has introduced a more disciplined and systematic approach to my software-development process."

The curriculum and the PSP course have also changed the faculty's views on teaching software engineering. They are more aware of the importance of process concepts and the rigorous treatment of them in software-engineering education. One instructor, for example, used to teach software design and construction mostly by focusing on a particular development life-cycle and a particular method. Now the course includes process concepts — the students must define and measure the design process throughout the course. And the semester-long term project now provides an opportunity for process measurement and improvement.

There is no question that process engineering is an important element of modern software-engineering education. The open questions are what materials should be included and where should they be placed in the curriculum. Nonetheless, it is imperative that software-engineering education have a formal approach to covering process concepts. ◆

## EDUCATION EVOLUTION

Software-engineering education, like software engineering itself, is evolving. Educators once felt that software engineering was so broad that only students with industry experience could really benefit from and appreciate study in the field. Therefore, the first curriculum guidelines on software engineering, published in 1976 by Peter Freeman, Tony Wasserman, and Richard Fairley[1] assumed a graduate-level curriculum. These guidelines identified five general areas: computer science, management science, problem-solving, design, and communication skills. In 1978, Freeman and Wasserman proposed a "first approximation" to a curriculum design leading to a master's degree in software engineering.[2]

The SEI curriculum assumes prerequisite knowledge equivalent to an undergraduate degree in computer science and also identifies five subject areas: systems engineering, software design and specification, implementation, verification and validation, and control and management. There are currently more than 20 graduate programs in software engineering, and most of them are based on the SEI curriculum.

## REFERENCES


1. P. Freeman, A. I. Wasserman, and R. E. Fairley, "Essential Elements of Software Engineering Education," *Proc. Int'l Conf. Soft. Eng.*, IEEE CS Press, Los Alamitos, Calif., 1976.
2. P. Freeman and A.I. Wasserman, "A Proposed Curriculum for Software Engineering Education," *Proc. Int'l Conf. Soft. Eng.*, IEEE CS Press, Los Alamitos, Calif., 1978.


## LOGGING OFF

This issue marks the end of my tenure as Quality Time editor. I want to thank Carl Chang for offering me such an enjoyable and informative opportunity for interaction with the software-engineering community. I have learned a great deal from the authors of articles as well as the readers of the department, including both those who took the time to write letters and those who passed on comments verbally.

Next issue, Shari Lawrence Pfleeger will take my place. Pfleeger is a president of System/Software, Inc., a consulting firm specializing in software-engineering and software-quality issues. She will continue to seek articles reporting on a practical view of quality: What it means to customers and users, how researchers and practitioners are working together, and how different organizations define, assess, predict, and improve quality.

— *Dave Card*

MANAGER

## FIFTEEN PRINCIPLES

*Continued from page 96*

ing prospective employees, remember that there is no substitute for quality. Don't compare two people by saying, "Person $x$ is better than person $y$, but person $y$ is good enough and less expensive." You can't have all superstars, but unless you truly have an overabundance, hire them when you find them!

**14. FOLLOW WITH CARE.** Just because everybody is doing something does not make it right for you. It *may* be right, but you must carefully assess its applicability to your environment. Object orientation, measurement, reuse, process improvement, CASE, prototyping — all these might increase quality, decrease cost, and increase user satisfaction.

However, only those organizations that can take advantage of them will reap the rewards. The potential of such techniques is often oversold, and benefits are by no means guaranteed or universal. You can't afford to ignore a "new" technology. But don't believe the inevitable hype associated with it. Read carefully. Be realistic with respect to payoffs and risks. And run experiments before you make a major commitment.

**15. TAKE RESPONSIBILITY.** When a bridge collapses we ask, "what did the engineers do wrong?" When software fails we rarely ask this. When we do, the response is, "I was just following the 15 steps of this method," or "My manager made me do it," or "The schedule left insufficient time to do it right." The fact is that in any engineering discipline the best methods *can* be used to produce awful designs, and the most antiquated methods to produce elegant designs.

There are no excuses. If you develop a system, it is your responsibility to do it right. Take that responsibility. Do it right, or don't do it at all. ◆