Determining a Complete Set of Software Development Standards: Is the Cube the Answer?

It's old news that every software development team needs a set of development standards to speed communications among team members and to minimize communication errors. However, very few of us can define what a complete set of standards is. For that matter, very few of us can define a complete set of tools or techniques for developing software. Some new ideas are now being experimented with in our industry, and in this issue we will look at one model for a programming environment that provides a conceptual framework for identifying the complete set of standards required for a software development project.

Every development organization needs to increase product reliability, decrease development costs, and reduce legal liability risk. Typically, corporations use a software quality assurance program to address these needs. Most quality assurance programs take the same three-pronged approach: prevent bugs from occurring, find bugs early, and make the developer more efficient. The first two actions reduce rework. The third speeds the creation process. All three actions require the use of software development standards.

Prevent bugs. Software developers can prevent bugs from occurring. Two approaches have been shown effective in other industries, as well as ours. One approach is the use of modeling and prototyping, and the other is the use of standards. Modeling and prototyping enable the developer to gain a better understanding of what the final product will do. That comprehension will minimize errors in the definition of the product. Standard techniques, terminology, documents, tools, and measurements decrease the probability of misinterpretations among developers. All errors cannot be prevented by using these approaches, but a significant percentage can be. If you are going to use these approaches, you've got to have the standards established before development starts.

Find bugs early. Software teams can find bugs before testing. A number of techniques for finding bugs early in the development cycle are now well-known in most segments of our industry. Barry Boehm put his finger on most of these techniques in his article, "Verifying and Validating Software Requirements and Design Specifications," in our premiere issue of IEEE Software. All of the techniques he identified provide ways to check the product for potential problems. All of the techniques would be easier to implement and more effective when applied if the product were created in accordance with standards.

Make developers more efficient. Other industries have made workers more efficient by purchasing tools for them. Still, there is a trade-off between the cost of worker time and the cost of capital equipment. Capital equipment in our industry can be hardware or software tools or both. The enormous amount of work going on in the development of programmer workstations indicates considerable interest in using capital equipment to make the developer more efficient. And there are a lot of tool builders out there betting on buyer acceptance of this approach. But that leaves difficult questions for the buyer or the development organization to wrestle with. Which tools? How do we define requirements for tools? Only one guideline seems applicable: the tools must support the techniques, standards, and measurements used by the development organization. Getting tools established prior to development start-up is very important.

The long-term goal. We need to increase product reliability, decrease development costs, and reduce legal liability risk. These needs and widely accepted approaches to reducing rework and enabling workers to be more efficient provide us with the historical perspective to articulate now a long-term goal for our development organizations. In the long term, we want to improve both the software development process and products from project to project. We want every new product to be of higher quality than the last one. We want each new project to be more effective than the previous one. It is not practical to make too many improvements at once, though, so we must work on them one step at a time.

The first step is to define the requirements for the total programming environment. If we are to reach our long-term goal, both the software product being created and the process of creating it must be measurable, repeatable, and changeable. These most important requirements lead us to basic design constraints on the environment.

We have only three perspectives from which to work. These perspectives constrain the design of a programming environment. We can look at the programming environment from the point of view of (1) the product being created, (2) the staff performing the work, and (3) the policies and facilities that govern the work.

The product being created. Little, if any, control is exercised by the development team over the definition of a product being created. Product functionality, size, complexity, and reliability are determined mainly by what the user or market wants. What the user or market usually wants is a problem to be solved. The solution to the problem usually bounds the definition of the product. The only control developers have over the product is to partition the evolution of the product into manageable increments.

Developers divide the life cycle of software into phases. Each phase terminates with intermediate products or documents, which are reviewed or inspected and stored in a library for control and use. In
our industry today, many life-cycle descriptions are available. Which one the development team chooses is not important, but it is important that the team choose one and stick to it. The life-cycle phases constitute an axis of the programming environment as illustrated in Figure 1.

### Staff performing the work

The most important factor affecting the quality of the product and the efficiency of development is the staff performing the work. Yet the organization, including management, probably has little control over the experience, knowledge, and skill levels of the staff. Usually a project starts with a known staff mix, and it is not possible, practical, or desirable to hire or fire many people for the benefit of one project. Therefore, the organization has to deal with the staff it has.

The organization must look at the available staff and partition it into job functions. Some people must plan and control the effort, so a project management function is established. Some must establish techniques, standards, measurements, tools, and training so that development work can be carried out effectively. This is called the project assurance function. It is vital to identify, capture, and control code, documents, test data, etc., as they are produced. The people who keep track of the product pieces are performing the configuration management function. Most of the staff will be involved in defining, designing, and building the product. They perform the development function. Those who check and evaluate the product for correctness, completeness, and consistency throughout its life cycle are performing the verification, validation, and testing function. If it were a one-person project, some percentage of that person's time would be spent on each of these functions.

Combining the job-function perspective with the life-cycle-phase perspective gives the matrix in Figure 2. The job functions and their relationship to the life cycle are illustrated and defined more formally in the boxes in Figure 2.

### Policies and facilities that govern the work

We've already noted that the staff is the most important factor in determining product quality and process efficiency. A staff member can operate effectively only when he knows the answers to basic questions regarding the job:

- What is expected of me?
- Why is it expected?
- How do I do what is expected?
- What must I produce?
- How will my product be evaluated?
- What tools are available to me?
- What training is available to me?

Support elements provide the answers to all these questions and form the third perspective of the programming environment. Support elements are the management policies, techniques, standards, measurements, tools, and training that enable a person to perform a job func-
tion. Let's examine the elements one at a time.

Management policies define the life-cycle phases and the job functions. They are written descriptions of what should be performed by each job function in every life-cycle phase. These descriptions may be called a methodology, a corporate policy, an instruction, or a procedure. Whatever they are called, they must be in place.

Sometimes we find management policies written by technical people and supported by management. This is a manifestation of a quality circle, with the technical team defining what should be done.

Techniques are written descriptions of how to perform a particular job. They may be called a methodology, a method, or a procedure. However, they are referred to, they are extremely important. Examples of techniques include the Jackson System Development Methodology, the Constantine Structured Design Methodology, and the Barry Boehm Verification Techniques mentioned earlier.

We usually think of a product as being the end result of the entire development process. For our purposes here, we will consider the result of the work performed by those in one job function, in one life-cycle phase, to be an intermediate product or configuration item. An intermediate product may be a document, a section of a document, a collection of code, or a set of test data. Any person performing any job function in any life-cycle phase needs a description of the intermediate product he is to produce.

Given a description of the intermediate product, the next question becomes how to measure the product. We need objective metrics on both the process and the generated products. Measurements provide the immediate benefit of refining the development plan and the long-term benefit of characterizing the effectiveness of the current programming environment. Well-defined measurements are necessary. The measurement may relate to size, complexity, functionality, or the number of errors discovered during reviews. Whatever the measurement, it must be defined before development begins. As with all the support elements, measurements are needed for each intermediate product produced by people in each job function in each life-cycle phase.

Tools are a big deal. The right tool can be a blessing. The wrong one is a curse. The National Bureau of Standards tools database lists far more tools than the average person can possibly screen. The NBS taxonomy does provide a way to search through the list to identify a required tool, but the taxonomy is cumbersome to use. There is no question tools are needed. But, which tools?

Figure 3. Programming environment model.

Several guidelines are useful in selecting or defining tools. Most importantly, a tool should make doing the right thing the easiest thing to do. Next, the tool should be an integral part of the support elements. It should support the management policies, techniques, intermediate product descriptions, and measurements. It should be supported by training. A tool should be defined in terms of the job functions it supports and the life-cycle phases where it is used.

The inputs to a tool should come from the person performing the job function in that life-cycle phase and from the output of any previously used tool. The output of a tool should be available to the person performing the job function in that life-cycle phase and to all people or tools in later life-cycle phases. It is a mistake to buy or define a tool that is not an integral part of the support elements.

Training is mentioned as the last of the support elements here, but it is vital to the success of the programming environment. Having a defined technique is useless unless every member of the team knows how to use the technique. Training should be provided on all of the support elements.

Combining the support element perspective with the other two perspectives gives the complete Programming Environment Model illustrated in Figure 3.

We can return now with an affirmative response to the issue raised at the beginning of this article. Yes Virginia, it is possible to identify the complete set of standards required by a software development project. Looking at the cube in Figure 3, we can see that we need support elements for each job function in each life-cycle phase. The standards for the project are unique, nonredundant support elements that occupy all the cubicles in the figure. If any standard support element is missing, the cube has a hole. (By the way, since a tool is a support element, we have also determined a method for identifying a complete set of tools.)

This type of model provides an additional benefit: support elements can be changed or improved from project to project. They are separated and identified, and their impact on the development process and product quality can be measured. The cube shows us a way to reach our long-term goal.

The cube represents one approach to identifying a complete set of software development standards. Leonard Tripp, chair of the IEEE Software Standards Working Group on Standards Taxonomy, is presently exploring approaches for classifying IEEE software engineering standards. As Tripp's group nears completion of its work, we look forward to discussing its findings. In the meantime, we at the "Software Standards" department encourage your questions and comments on the cube.