Software Productivity

This is a collection of 19 published papers and internal IBM memoranda written between 1967 and 1981. Those formerly available are again welcome (they do not seem at all dated); those available for the first time complement the others well. It's good to have these old favorites back again, and it's also good to have a collection to recommend to colleagues and students.

These articles present a good foundation of software engineering to the reader—a foundation in both senses of the word in that the articles provide the historical and fundamental underpinnings of the field. The earliest articles are from the era when "software engineering" was only a clever phrase denoting a problem area. These discussions and their suggestions for development of an engineering discipline were on target and are highly recommended for those in the field today.

Software engineering has suffered from overenthusiastic hype, and current wisdom tells us that the only significant variable affecting programmer productivity is the individual programmer. Even if this is the case, the super programmer can be leveraged as described in the article "Chief Programmer Teams: Techniques and Procedures." Top-down programming, a necessary component of that management technique, is described in "Programming Techniques: From Private Art to Public Practice."

Although a lot of Mills's ideas are often sold by others as good old common sense to an audience who can easily say, "that's the way we've always done it," Mills bases his ideas on mathematical underpinnings (discrete mathematics), which is not the way we've always done it. Articles such as, "Mathematical Foundations for Structured Programming" and "The New Math of Computer Programming" show, from a theoretical basis, why top-down structured programming and the chief-programmer team are correct. Top-down is not the obvious, goal-directed approach, and the chief programmer is not what to do with a promoted techie. Rigorous discrete mathematics, not cracker-barrel management, is what we need as programmers and what we must provide as educators and managers.

By learning Mills's science, we can become successful practitioners of the programming art. As he explains more than once, the student who merely gets the right answer just barely passes; the student who also writes down the work deserves the "A." And the teacher knows that the second student is ready to attack the next piece of harder material.

Mills never tries to sell us a crank we could turn to tighten up our discipline. But he does give us principles to use to understand what we do, so we can do it better, and teach it, and manage with it.

Peter G. Anderson
Rochester Institute of Technology