Considerations in the transfer of software engineering technology

by MICHAEL J. McGIN

Syracuse University
Syracuse, New York

INTRODUCTION

The development of new and increasingly efficient techniques of software engineering seems to be impressive to everyone except the professional software developers. In a recent summary of a panel discussing software engineering problems to be faced during the 1980's, Wasserman notes:

Not surprisingly, the panel concluded that the problems of the '80's look very much like the problems of the '70's and depressingly similar to the problems of the '60's. The basic questions... were presented by the chairman as follows:

1) How can well engineered products and systematic procedures for their creation be developed?
2) How can the body of software engineering techniques be applied to existing systems?
3) How can technology be transferred more effectively from the research community to software developers?

This paper will examine the prospects for the use of software engineering techniques in the near future by the "average programmer." The characteristics of technology transfer will be examined and put into perspective with software engineering.

THE PROGRAMMING ENVIRONMENT

Programming is an intellectually challenging exercise with at least two factors, size and complexity, identified as significant by Gries. Size is important because any one person can remember only a small portion of a programming system. However, if size was the only consideration, then the task of programming, like that of knitting, might be challenging but not intellectually stimulating. Complexity factors combined with size factors insure that programming will remain an intellectual challenge. Complexity may be either associated directly with the computational process or with the determination of the correctness of the program.

Structured programming and design are approaches to the size and the complexity problems and have a significant impact on the computer industry. The concept of GOTO—less code has led to case studies of systems with tremendous improvements in productivity, reliability, and reduced maintenance costs. The IBM system developed for the New York Times was the example used by many proponents of structured techniques. Unfortunately, one discovers that numerous maintenance problems persisted even in this exemplary system.

Modular design and top-down design continue to be catch phrases for programmers. Many techniques have been defined for the definition, description, implementation, and testing of programs. The techniques allow for hierarchical descriptions, structured charts, decision tables, control graphs, etc. Each is designed to enhance the programmer's effectiveness and efficiency. However, Yourdan and Constantine note that, "to say that the average programmer's design process is organized, or structured, would be charitable."

The approaches to the design, development, and evaluation of a software product are continually increasing in number and sophistication. For instance, Gries' points out three approaches to programs:

Enumerative reasoning

Used to understand the sequence of statements, conditional statements and some uses of GOTO. It is an approach to understanding and showing the correctness of execution paths.

Mathematical induction

Used to understand iteration and recursion. Induction may be used to show that procedures are correct irrespective of the number of times the procedure is invoked.

Abstraction

Used to isolate the relevant properties or qualities of an object so that one can focus on what the object does. In this manner one examines data types, variables, etc., to insure correct operation.

The ideal is to have a program that can be proven correct. Recognizing these approaches is, however, quite different
from applying them. Most programmers are aware of software engineering devices and in fact recognize the need for increasingly effective methods. Procedures for proving the correctness of a program have been considered for more than a decade.7 The procedures are the results of efforts by numerous computer scientists. Scientists, even computer scientists, tend not to be professional programmers. Professional programmers are more properly categorized as very high level technologists. Therefore, the transfer of software engineering technology is really an issue which concerns the transfer of information about available tools, techniques, and procedures from the computer scientist to the professional programmer technologist. The ultimate concern is with the alteration of programming practices. Many changes will occur at the individual level where the single person is the adapter or rejector of an innovation. Change also occurs at the organization level where it may be called development, specialization, integration, or adaptation. Whatever it is called and on whichever level it occurs, change must ultimately be understood by a focus on the communication process and its participants.6

PROGRAMMER CHARACTERISTICS

The programmer is a consumer of massive quantities of information. This individual must understand the problem environment by acquiring information through both formal and informal channels. The programmer must gather this information from internal and external sources as well as from a memory in order to develop a range of solutions to a problem.7

The programmer's purpose is to produce an acceptable piece of software. This goal is in sharp contrast to that of the computer scientist who is seeking a fundamental understanding of a concept and a resulting publication. The implications for the person supplying information to support both the programmer's function and the computer scientist's function are enormous. The scientist requires a systematic collection and organization of documents as well as a mechanism for making these documents available. Thus, the end product of one scientist (a publication) may serve as immediate input to a second scientist (Figure 1).

The programmer's output is a software product. In order for this to serve as input to a second programmer's task, the program must be decoded and analyzed. The programmer clearly has access to the report of the scientist. However, the programming problem confronting this person is specific, immediate, and probably important to his professional and economic future (Figure 2). The scientist's report is abstract and as such will require interpretation; it is general and will therefore require translation to this specific problem; and it is unfamiliar to the programmer and thus the programmer using the results of this paper is likely to make a few mistakes at first.

It is much more efficient for the programmer to have another program and its programmer explain the software product. These products are used either as a basis for new programs or as prototypes of new programs. Documentation alone does not suffice. It often requires an explicit knowledge of the software product. Typically this requires human intervention or supplementation.

The programmer is faced with the dual problem of (1) following at least the trends in the ever increasing volume of information from scientists on software engineering, and (2) using mechanisms and/or people to aid in the decoding and analysis of already existing software products.

Keeping up with the available literature is an acknowledged problem among scientists. The growth rate of potentially relevant literature is increasing with no indication of achieving or approaching a limit.8 This, of course, increases the difficulty of locating useful items among the useless items. Mooers9 developed a "law" which states that information will not be used when it is more trouble to acquire the information than it is to be without the information. As a result, "information scientists" have developed information systems to help scientists meet their information needs. These attempts are documented and analyzed in the fourteen volumes of the Annual Review of Information Science and Technology. Unfortunately these systems are not adequate to meet the needs of the high level technologist such as the programmer.7

It was pointed out that programmers rely heavily on associations with other programmers. This does not occur without restrictions. Organizational constraints are placed on a programmer's ability to interact with those from other organizations. The programmer is expected to work with the employer to achieve this goal and is expected to refrain from preliminary disclosures. Thus, proprietary information inhibits the free flow of information among programmers.

---

Figure 1—Scientists' development and use of information.

Figure 2—Programmers' use of information sources.
In spite of barriers, some techniques do seem to diffuse quickly among organizations. Allen suggests that since humans are the best carriers of information, employee turnover may be a cause of the spread of technology. As an example, Roberts and Wainer showed that applications for space technology occurred as individuals left the academic and space research organizations to start their own businesses or work for others.

For the truly effective transfer of technical information, one must make use of the human ability to recode and restructure information so that it fits into new contents and situations; each job change brings a record of experiences and a vast amount of "proprietary" information; a continual flow of job changes insures that no single firm is far behind.

INTEGRATION OF SCIENCE AND TECHNOLOGY

A common assumption is that science provides the basis for technology. However, Price investigated patterns of citations among technologists and scientists over a 20-year time period to determine this dependence. He concluded that science and technology progress quite independently of one another. Gibbons and Johnston tried to refute this by examining small technological advances. They did indeed trace five technological advances to a scientific source. The average time between the publication of the scientific finding and the technological advance was 12.2 years. There are examples of faster adaptations. If a need is developed by the technologist, then science will often attempt to fill the need and if successful a rapid use will be made of the scientific finding.

So there is some reason to believe that a connection exists between science and technology. In most cases, this is a slow process with occasional exceptions.

IMPLICATIONS FOR SOFTWARE ENGINEERING

The realities of software engineering technology are (1) the computer scientist is frustrated by the lack of adaptation of what are considered to be valuable tools and techniques, and (2) that the professional programmer is faced with the task of identifying information immediately relevant to a specific programming task; scientific papers are often viewed as abstract academic exercises with little practical application.

It is not surprising that one finds a variety of reasons why innovations have not occurred. For example, at a recent panel consideration on the lack of formal specification of programs, the following reasons were presented:

- We lack adequate tools and support. We need better database management facilities, better tools for viewing a system at different levels of abstraction, and better proof tools.
- Progress has been slow because systems are hard to specify.
- Software development occurs over such a long time span (7-10 years) that both the software requirements and the state of the art in software specifications are likely to change.
- Completely formal specifications are hard to write and people do not need them.
- People fail in writing specifications because they approach it from only one viewpoint.
- Progress is slow partly because the people working on specification techniques are not working on real systems.

A commonly heard prescription is that the programmer must be educated in the modern tools and techniques of software engineering! In fact, the proliferation of organizations, seminars and "experts" willing to reeducate professional programmers do provide evidence that many individuals and organizations recognize this as a need.

On the other hand, suggestions have been made to make the published papers available to the professional programmer more relevant to their immediate needs. For example Gerhart suggests that the repeated publication of software engineering advances which use trivial or easily specified programs such as sorting or greatest common divisor is not a helpful practice for the professional programmer. It is a useful technique for the computer scientist in that it assists in the comparison of methods. However, the programmer ends up with a lack of either depth or breadth of experience with the programming tools. Gerhart suggests a publication outlet for free standing proofs of a variety of programs.

One might also suggest that information professionals have not done their jobs. There is little known about the information needs of the professional programmer. On the contrary one is more likely to read statements about what the professional programmer should be:

The world today has about a million "average programmers," and it is frightening to be forced to conclude that most of them are victims of an earlier underestimation of the intrinsic difficulty of the programmers task and now find themselves lured into a profession beyond their intellectual capabilities. The conclusion that competent programming required a fair amount of mathematical skills has been drawn on purely technical grounds and, as far as I know, has never been refuted.

But software engineering technologies will be accepted and used by even the most mediocre programmer if there is adequate motivation provided. Rogers and Shoemaker note that:

The innovation process begins with an individual, or set of individuals, recognizing that their organization is facing a "performance gap" between their expectations and reality. This problem recognition sets off a search for alternatives, one of which may be innovation. The new idea usually comes from outside the organization, and must be matched with qualities of the organization's problem. Usually the innovation must be modified somewhat as it is implemented to fit the organization's conditions. So the innovation process consists of problem recognition, searching for alternative solutions, matching the innovation with the organization's problem, and implementation of the innovation, leading eventually to its institutionalization when it is no longer recognized as a separate element in the organization.

The information professionals must recognize the needs
of the professional programmer; the programming managers must recognize a need to create programs more effectively; professional programmers must be given the opportunity to communicate in the manner most effective for them to acquire the available tools and techniques; and adequate reward structures must be present to encourage change rather than inhibit it. There are no individuals at fault, and no simple prescription for change. Rather, there needs to be a recognition that the goals of the programmer are different from the goals of the computer scientist. The information useful to one is not useful to the other and the needs of the scientist are not the needs of the programmer.

REFERENCES