From ANS MUMPS to ISO M

by Wolfgang Kirsten

Introduction

At the beginning of 1994, I published a textbook for the German-speaking audience titled *Von ANS MUMPS zu ISO M—Fortgeschrittenes Programmieren in M*, which might best be translated as *From ANS MUMPS to ISO M—Advanced Programming in M*.[1] It was accepted very favorably and since has become well known in Europe. A Russian translation will appear soon, along with a Portuguese translation for Brazil. But I think that the book is less known to the American M community, so I am happy to have the opportunity to present its basic ideas to the readers of *M Computing*. An English translation is under negotiation at this time.

The book is aimed primarily at advanced M programmers. There are several introductory texts, including the one by Hesse and Kirsten in German.[2] Although *From ANS MUMPS to ISO M* is likely to become a classroom textbook and therefore contains all language elements of the new standard, it is more; it is very persuasive because it emphasizes again and again the advantages of M compared with other technologies. In many places, the book describes fundamental concepts of M. The explanation of the new language elements is imbedded in the frame of the general concepts of computer science, and that is why I believe that this book belongs in the hands of managers who want to implement data-processing projects with M.

The chapter titles in the book are:

Chapter 1 The ANS MUMPS Standard
Chapter 2 Syntactical and Semantical Basics
Chapter 3 Software Engineering in M
Chapter 4 New and Advanced String Processing
Chapter 5 A System Model of M
Chapter 6 Networking in M
Chapter 7 Database Management
Chapter 8 Global Design
Chapter 9 Programming Portable Applications
Chapter 10 M Windowing Application Programming Interface (MWAPI)

The third chapter conveys the textbook emphasis of this work very clearly. For example, it includes all (new) language elements and concepts that relate in any way to software engineering, such as: program structure and structured programming; subroutines or extrinsic functions; and error processing according to the new standard.

This book contains these basic software engineering concepts in a section on the productivity of M. Because I believe these topics are of uniform importance for all who deal with M Technology, I've chosen to reproduce in this article this portion of the computer book so as to present my views of M and software engineering.

Foundations of Software Engineering

When you work closely with a conventional programming language, sooner or later you will ask yourself how completely that language supports the recognized elements and regulations of software engineering.

In brief, software engineering is the management of the entire life cycle of large-scale software projects. In the past M itself was criticized because it did not allow for structured programming and allowed the programmer to use its individual language constructs—most especially indirection—to produce an erroneous, or at the very least a barely readable and thereby poorly maintainable code. On the other hand, managers of M projects have emphasized repeatedly its high productivity, which is at least five times better than that of comparable programming languages.

Finally, M is uniquely suited—say M supporters—to produce program generators, incorporating the required language constructs achievable for that task with M—here again indirection is mentioned. Meta-level generators further increase M's productivity.

Extensive literature on software engineering and M includes articles showing how to achieve clear programming structures within M. A prime example of these is the documents that Dr. Frederick Hiltz distributes at his advanced seminars during the annual meetings of the M Technology Association - North America. (It is a pity that these witty scripts have not yet appeared as a textbook.)

An article by G.E. Cole raised questions of efficiency and program quality in M.[3] An article by Winfried Gerum compared the control structures of different programming languages.[4] In 1993, Susan H. Johnston's article discussed the influence of programming languages on software costs,

comparing M to COBOL. She received M Technology Association's Distaso Award for this work.[5]

Second, the original German chapter emphasizes publications that discuss questions of software engineering from a meta-language viewpoint. Above all I must mention the two books by Blum describing the special productivity with and the advantages of M.[6,7] Bruce Blum is considered one of the foremost experts of software engineering in the world, and he is also an expert in M. Under his guidance, a clinical information system written in M was developed. It was introduced between 1975 and 1983 at The Johns Hopkins University School of Medicine. This development led to a high-grade CASE (computer-assisted software engineering) tool named TEDIUM.

The third emphasis deals with reports that compare M with other programming languages. Several works are cited below.

The Life Cycle of Software

The term software engineering originated during two NATO Science Committee conferences in the 1960s. At that time, large software projects were experiencing delays in initial delivery, faulty delivered systems (accompanied by expensive improvements), or even termination of large-scale software development projects (the worst imaginable case).

Related to those was the famous 1979 U.S. Government Accounting Office report. It contained the results of a study about the quality of delivered software. According to the report, it is possible to classify projects as follows:

- 50 percent suffered from cost overruns;
- 60 percent suffered from delay;
- 45 percent of the ordered software could not be used;
- 29 percent of the ordered software was never delivered;
- 19 percent of the ordered software had to be revised;
- 2 percent of the ordered software could be used without change.[8]

Since then, the reasons for these problems have been identified and recognized more clearly. They include:

- Poor project work;
- Missing or false expenditure estimates;
- Poor productivity;
- Individual, nonmaintainable code;
- Missing or insufficient documentation; and
- Insufficient tests.

Research in the field based on these principles began to make real progress. Structured programming was introduced to improve code generation. In the area of software engineering, various phases in the life cycle of a software product were recognized, and several different models were put into practice.

Fundamentally, the whole process of producing software was modularized into a life cycle, and the individual phases of that cycle were examined very carefully. The phases of software creation are generally acknowledged today to be:

- Definition of the requirements;
- Design of the application, design specification;
- Coding and single test;
- Integration and system test; and
- Routine installation and maintenance.

Splitting big software projects into separate phases represented an important advance. It is particularly important, however, not to view the individual phases as dogma; in particular, one phase need not be completed before the next one can begin.

For some time, therefore, iterations have been built in the cascading-design model ("waterfall model"). It was recognized that the process of software production was not a sequence of clearly modularized steps, but an iterative process, which leads ultimately from an initially defined problem to a product that comes as close to the original specification as possible. This process, of course, was known to the practitioner all along.

The Seven Rules of Software Engineering

On the basis of many years of experience in software production in different projects and different settings, the following seven rules are verified repeatedly (cited according to Blum).[9]

1. The rate of pure programming of a project is estimated at approximately 20 percent. But 40 percent goes to analyzing and designing, and another 40 percent is used for system integration and testing. This rule is also known as the 20-40-40 rule.

2. Most of the errors found stem from the analysis and the design phases; comparatively few are programming errors.

3. Verifying and validating components of the overall system should start as early as possible. The later an error is detected, the more expensive is its elimination. Once the product is delivered, repairing a mistake can cost one hundred times as much as it would have cost during the design phase.

4. The individual productivity of a programmer is independent of the programming language used if one measures productivity by lines of code generated per unit of time. If the programmer counts the implemented functions per time, however, higher programming languages clearly result in higher productivity.

5. Because of the requirement for increased communication between programmers, the individual productivity of a programmer decreases as the number of participants in a project increases.

6. The most important factors affecting cost estimation of a project are the quality and experience of the persons involved.

7. Maintenance costs of a software system during its lifespan are usually twice as much as the original production costs. More than half the costs arise from the improvements to the original product, a quarter of the maintenance costs are caused by adapting to changing needs, and only one-fifth of these costs result from correcting the original code.

A Case for M

In a famous study by Alonso published in various places under the title *A Case for MUMPS* at the beginning of the 1980s, the author compared different characteristics of MUMPS and COBOL.[10] His study produced important results, including performance of database queries, that were very favorable to M[UMPS]. More importantly, however, Alonso discovered that the relationship of the program lines (often called LOC for lines of code) was 1:5 in favor of MUMPS.

These two statements mean that, on the average, a program system in M uses only 20 percent of the program lines that the same program in COBOL would use, and can be written in one-tenth of the time. Still differently expressed: The productivity of a COBOL programmer amounts to only 10 percent of an M programmer, or a team of ten COBOL programmers achieves the same as one M programmer.

The results of this study are in line with findings of Munnecke et al. who concluded that the LOCs had a ratio favoring M by 1:4.[11] In a later article, Munnecke even measured 1:12 for the LOCs and 1:10 for the productivity.[12] Great care in research was taken for both articles, which makes them extraordinarily informative. Anyone interested in the problems of comparing M's productivity to other computer languages should read them.

Also, Blum pointed out the advantages of the productivity of M[UMPS] in great detail.[13] On the basis of a detailed analysis of carefully kept records covering many years, he came to the conclusion that the ratio of the LOCs is about 1:5, and that of productivity is higher than 1:8.

Finally, I want to cite another—yet unpublished, however, partially known—German market review of a big enterprise, which specifies that both ratios are 1:6 (LOCs and productivity).

These results are summarized in the following table:

Study	LOCs M:COBOL	Productivity M:COBOL
Munnecke et al.	1:4	no statement
Munnecke 1980	1:12	1:10
Alonso 1984	1:5	1:10
Blum 1990	1:5	1:>8
German Study 1990	1:6	1:6

Table 1. The ratio of lines of code (LOCs) and the productivity between M and COBOL by various studies.

What are the reasons for these results if you also take the above seven rules into consideration? One could argue that LOCs are a poor measure for a comparison between M and COBOL because in M several commands stand in one line, which is not the case in COBOL. That might be true on the average, but Munnecke pointed out in his case study of 1980 that the number of characters in two equivalent programs, which describes a better measure, will result in a ratio of about 1:10 also.

The reasons for the much higher productivity lie deeper: M is a programming system and not just a programming language. M incorporates an integrated multiuser database system (which, of course, is no database management system in today's sense) and contains language elements for input/output and those that in other languages are relegated to job control language (JCL).

Munnecke referred to this fact when he said that a COBOL project is never realized solely in one language, but required the knowledge of at least a dozen other system functions with nearly two thousand pages of system documentation to learn, understand, and apply.

M is complete and comprehensive. It does not require the database specialist, the JCL guru, the transaction-processing monitor expert. Only the M specialist is required. With M, and only with M, is it possible to program complete, operative application programs for customers' implementation. Another observation is that M represents a programming language one step higher than COBOL and comparable languages. Software houses that are successful in the M market used to develop all their applications inhouse, but nowadays there are M generators on the market to further simplify application development in M.[14]

Normally these tools comprise not only the actual coding—which according to Rule 1 takes only 20 percent of the total time—but all phases of the software-development process.

Blum pointed this out in his book on TEDIUM, in which he estimated the increase in productivity through the use of TEDIUM against M would be four times higher (the same ratio of improvement noted for M over COBOL). The other named tools are probably similarly productive.[15]

On the basis of a record extending for many years, the average per day program lines produced in TEDIUM and a comparative analysis of the same codes with M and COBOL, Blum concluded that fifteen lines in TEDIUM represent about sixty lines in M and these compare to three hundred lines in COBOL. Here Rule 4 applies, which states that programmers write approximately the same number of lines per unit of time in any language; however, one line in M is five times as productive as one in COBOL. TEDIUM on the other hand is four times as productive as pure M code would be.

Program generators permit far larger-scale prototyping (they are an essential prerequisite for these tasks) than could be done using pure programming languages. Prototyping helps to avoid errors in analysis and design and helps us recognize them earlier. In this context, Rule 2 and Rule 3 are applicable.

Communication in Project Teams

Both of Blum's books devote significant space to communication within a project team, which increases exponentially with each additional member. He discussed the known "Rule of Five," referring to the idea that a maximum of five people can cooperate effectively at a functional level. It clearly is proven that individual productivity decreases with the size of the team, which was expressed as Rule 4. Larger teams use too much time in tuning themselves and thereby become inefficient. This rule also pertains to higher hierarchies of management. This means that five groups will be supervised from one manager and in turn five managers will be supervised from one upper-level manager. Three levels of hierarchies are needed for projects involving more than 130 employees.

As an example, Blum showed us an application he created called OCIS (a large tumor-information system), which is composed of six thousand lines in TEDIUM. Under the mentioned conditions, these would be equivalent to more than one million lines in COBOL and would need a large development team within which a great factor for internal communication must be calculated. Blum stated that such a project would be classified as nonpracticable before it has begun, at least in a hospital setting.

Indeed, most known large M applications such as FileMan and TEDIUM have been developed and written by only a small number of persons.

Now, a final remark about program maintenance mentioned in Rule 7, which is often viewed with skepticism for M applications. In my judgment, this skepticism is groundless. The already-mentioned German study from 1990 concluded that the maintenance effort for M programs is six times less than for comparable COBOL programs. This ratio is about equivalent to the productivity advantage and it is plausible to assume that an experienced M programmer can, in a given time, modify as many lines in M as an experienced COBOL programmer can modify in COBOL, except that a line in M is five times denser than its COBOL equivalent. This means that the maintenance productivity is five times greater.

The readability of programs (as an important prerequisite for the maintenance) is strongly dependent on the experience of the programmer. Susan Johnston showed us an example in her 1993 article. She compared the Evaluate-Statement in COBOL with the $SELECT function in M. Both of the following program segments are roughly equivalent:

```
COBOL
      Evaluate TYPE
            When 1
                  Move "a" to NAME-CODE
            When 2
                  Move "b" to NAME-CODE
            When 3
                  Move "c" to NAME-CODE
            When other
                  Move " " to NAME-CODE
      End-Evaluate

M
      SET NameCode=$SELECT(T=1:"a",T=2:"b",T=3:"c",1" ")
```

The program segment in COBOL reads easily. An M programmer has no difficulties with the $SELECT. What is immediately astonishing is how short M's formulation is. This leads overall to shorter programs in M, which in my opinion are much more readable than the page-by-page printouts in COBOL.

Wolfgang Kirsten, Ph.D., M.Sc. in mathematics, is a scientific worker in the Center of Medical Informatics at the J.W. Goethe University Medical Center in Frankfurt, Germany. He is a member of the M Development Coordinating Committee - Europe, of the Board of Directors of the M Technology Association - Europe, and editor in chief of *M Professional*. Kirsten is also coauthor of an introductory textbook on M, and is an assistant professor for advanced programming in M for computer science students studying medicine as a minor subject.

Endnotes

1. W. Kirsten, *Von ANS MUMPS zu ISO M - Fortgeschrittenes Programmieren in M* (Epsilon Verlag Darmstadt Hochheim, 1994).
2. S. Hesse and W. Kirsten, *Einfuhrung in die Programmiersprache MUMPS, Second Edition* (Berlin, New York: de Gruyter, 1989).
3. G.E. Cole, "Perspectives on Program Efficiency and Quality," *MUG Quarterly* 17:4 (1989): 47–49.
4. W. Gerum, "The Marvels of the FOR-Command," *MUG-Europe Newsletter* 8:2/3 (1991): 6–7.
5. S.H. Johnston, "The Effect of Language on Software Costs," *M Computing* 1:3 (1993): 39–54.
6. B.I. Blum, *TEDIUM and the Software Process* (Cambridge, MA and London, England: MIT Press, 1990).
7. B.I. Blum, *Software Engineering: A Holistic View* (Oxford University Press, 1992).
8. U.S. Government Accounting Report cited from Blum, 1992.
9. Blum, 1990.
10. C. Alonso, "A Case for MUMPS," *Computerworld* (January 1984).
11. T. Munnecke et al, "MUMPS: Characteristics and Comparisons with Other Programming Systems," *Medical Informatics* 2:3 (1977): 173–196.
12. T. Munnecke, "A Linguistic Comparison of MUMPS and COBOL," *AFIPS Conference Proceedings* (49) (1980): 723–729.
13. Blum, 1990.
14. Munnecke, 1980.
15. Blum, 1990.

CALENDAR

June 1–4, 1995

MUMPS Development Committee meeting, Hyatt Regency O'Hare, Chicago, Illinois. Call 301-431-4070 for details.

June 5–9, 1995

M Technology Association, 24th Annual Meeting, Hyatt Regency O'Hare, Chicago, Illinois. Registration packets shipping early March. Call 301-431-4070 for details.

July 23–27, 1995

International Medical Informatics Association Medinfo '95. Vancouver Trade and Convention Centre, Vancouver, British Columbia, Canada. For information, write Medinfo '95 Administration Office, Suite 216, 10458 Mayfield Road, Edmonton, Alberta, Canada. Phone: 403-489-8100; fax: 403-489-1122.

October 15–19, 1995

ACM's 10th OOPSLA (Object-Oriented Programming, Systems, Languages, and Applications), Austin, Texas. For information on the meeting contact the OOPSLA '95 Office, 7585 SW Mohawk Street, Tualatin, Oregon 97062. Phone: 503-691-0890; fax: 503-691-1821; e-mail: oopsla95@acm.org.

November 6–10, 1995

MTA-Europe Annual Meeting, Barcelona, Spain. For information, contact the MTA-Europe Office, Avenue Mounier 83, B-1200, Brussels, Belgium. Phone:32-2-772-9247; fax: 32-2-772-7237.