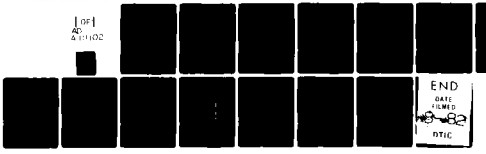
AD-A111 102 DEFENSE MAPPING AGENCY WASHINGTON DC F/G 9/2
LESSONS LEARNED ON THE ROAD TO A MODERN PROGRAMMING ENVIRONMENT--ETC(U)
JAN 82 A J KRYGIEL

UNCLASSIFIED NL



UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

LEVEL II

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER	2. GOVT ACCESSION NO. AD-A111 102 / 3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Lessons Learned on the Road to a Modern Programming Environment	5. TYPE OF REPORT & PERIOD COVERED Final
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Dr. Annette J. Krygiel	8. CONTRACT OR GRANT NUMBER(s) N/A
9. PERFORMING ORGANIZATION NAME AND ADDRESS Defense Mapping Agency Directorate for Systems and Techniques Advanced Technology Div. - Bldg 56, USNAVOBSY, Wash DC 20305	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Mapping Agency Directorate for Systems and Techniques Advanced Technology Div. - Bldg 56, USNAVOBSY, Wash DC 20305	12. REPORT DATE 6 January 1982 13. NUMBER OF PAGES 17
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Defense Mapping Agency Directorate for Systems and Techniques Advanced Technology Division Bldg 56, USNAVOBSY, Wash DC 20305	15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

This document has been approved for public release and sale; its distribution is unlimited.

DTIC ELECTE FEB 1 0 1982 E

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Modern Programming Environment
Software tools
Tools facility
Software engineering

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The problems of coping with increasing levels of automation in a large organization are formidible ones. Not the least of these is the organization's investment in the computer software needed to deliver its products. The environment in which the software is developed must be supportive, allowing rapid generation of error-free and maintainable computer code which meets user requirements. Defining the components which constitute this environment -- called a Modern Programming Environment (MPE) -- is merely a first step. Constructing the -

(OVER)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

AD A111102

DTIC FILE COPY

20. Continued:

environment and then ensuring the proper interaction of personnel with it must also be accomplished.

This paper discusses some of the steps that have already been taken by the Defense Mapping Agency to proceed to an MPE and the lessons learned along the way.

Lessons Learned on the Road to a Modern Programming Environment

Annette J. Krygiel

Defense Mapping Agency
Directorate for Systems and Techniques
Advanced Technology Division
Bldg 56, U.S. Naval Observatory
Washington, D.C. 20305

Biographical Sketch

Dr. Annette Krygiel has been an employee of the Defense Mapping Agency since 1963. She is presently a member of the Advanced Technology Staff at the Headquarters where she directs research and development activities to apply advanced computer technology to the Agency's computer resources. This includes applying automated tools and techniques to the actual process of developing the Agency's software.

Dr. Krygiel has a B.S. in Mathematics from St. Louis University, and a M.S. and D. Sc. in Computer Science from Washington University, St. Louis. She is a member of the Association for Computing Machinery and the Institute for Electrical and Electronic Engineers.

Abstract

The problems of coping with increasing levels of automation in a large organization are formidible ones. Not the least of these is the organization's investment in the computer software needed to deliver its products. The environment in which the software is developed must be supportive, allowing rapid generation of error-free and maintainable computer code which meets user requirements. Defining the components which constitute this environment -- called a Modern Programming Environment (MPE) -- is merely a first step. Constructing the environment and then ensuring the proper interaction of personnel with it must also be accomplished.

This paper discusses some of the steps that have already been taken by the Defense Mapping Agency to proceed to an MPE and

the lessons learned along the way.

Introduction

In 1979, when Mr. Owen Williams addressed this forum, he provided insight into the Defense Mapping Agency's (DMA's) future Mapping, Charting and Geodesy (MC&G) systems [1]. He described the growing requirement to deal with MC&G sources of a digital nature, such as GEOS-3 and LANDSAT, stressing the growing need to produce mapping products in digital form, in support of advanced weapons systems, such as terminally guided re-entry systems and cruise missile. Indeed, DMA has passed the point where the bulk of our products are conventional paper products; instead today the majority of these products is in a digital form.

As a result of these requirements and trends, we have escalated automation of the mapping and charting process, and are constructing systems to accept source information in digital form, exploit that information digitally, and output it as a digital product. There are many processes which occur from the point the source information arrives at our doors to the point that the product exits. At points in between, we are building large repositories of digital data for increased responsiveness and flexibility. Many (if not all) of our processes are directly supported by computer systems of varying levels of sophistication. Therefore it is accurate to say that our computer systems are needed to accomplish our mission, and our dependence on them can only grow with increasing quantities and varieties of digital data, both input to us, and produced by us. Consequently it is also accurate to state that our software is mission critical, and it is growing in cost, complexity and size.

This is a problem faced by many organizations, and acknowledged within the Department of Defense to the point where a major Software Technology Initiative is being constructed to improve software productivity; Figure 1 illustrates the motivation [2]. If the trends in software costs continue, they will dominate the cost of automated systems. However another troubling aspect is the growing complexity of the software. As additional functions are performed by computers, the reliability of the software becomes more important, but also more difficult to attain. An example of growth is seen in Figure 2 which derives from National Aeronautical and Space Administration data on the U.S. manned space flight programs [3,4,5,6]. It required about a million computer instructions to launch Mercury; however it required about 40 million to support the Space Shuttle operation. This reduced the number of personnel who were required to participate in the launch operations.

In DMA, the growing software size and complexity is reflected in a comparison of the DMA Aerospace Center (DMAAC) active software inventory resident on the Center's mainframe computers (Figure 3). Comparing active software programs in 1970 versus 1981 reveals an increase in number of programs (231 versus 916), and an increase of 40%, 49%, or 73% in average lines of code per program (dependent on alternative views of the total inventory of 916 programs).

In 1979 I delivered a paper here which described the tools in development which are intended to increase the productivity of all our DMA scientists who engage in software development, as well as improve the software product itself [7]. These tools are only one part of an entire environment which is necessary to support a productive and disciplined approach to software development. Today I want to share this broader perspective, and discuss what we have learned in the process of introducing software tools. I will also comment on where we are headed.

Definition of Modern Programming Environment

There are many ingredients or components constituting an MPE. These include:

a methodology for software development based on the life cycle

organizational standards and guidelines for software development

computer system(s) hosting automated tools to facilitate the development process

people trained in modern programming practices

management commitment to develop and sustain the environment

The life cycle approach acknowledges distinct phases in the software development process, and incorporates the processes of validation and verification of the products of each phase. These phases are: requirements definition, which is the generation of complete, consistent, and unambiguous specifications describing what the software will do; design, which is a specification of the software architecture, its control and data structures, its components, and the interaction and communication between components; coding, which states what the software must do in terms of a formal language intelligible to a computer; testing, which exercises the code to ensure correctness; and maintenance which includes updates to the software, including corrections,

modifications and enhancements.

Organizational standards and guidelines define the policies establishing what rules will be followed when software is developed. Examples include guidelines to accomplish requirements definition, i.e., formally or informally, whether a team approach will be employed, and under what circumstances, when structured walkthroughs will take place, etc.

An integrated tools facility is the computer system(s) which offers the automated support to develop software. There are varying levels of this support, and most facilities are viewed as evolutionary, proceeding from some minimum configuration to some maximum configuration of hardware and software. Software development tools are computer programs that aid the specification, design, construction, analysis, documentation maintenance, and management of other computer programs. These tools include those traditionally acknowledged, such as compilers and editors, those recently developed, such as requirements analyzers, and design aids, and those in research, such as formal verification tools and programming environments [8].

People trained in modern programming practices are one of the key ingredients to this environment. Among the factors which influence software productivity, Boehm notes that people factors dominate [5,6]. Modern programming practices include structured programming, top down design, program library functions, use of project teams, and structured walkthroughs.

A management commitment which sustains the people and computer resources, as well as enforcement of standards.

Lesson No. 1: Take Time to Assemble All MPE Components!

One important observation about transitioning to a MPE is this: it is a long process and all components must be sustained. We note this as one of our first lessons learned.

Several years ago in our efforts to improve the production environment, we developed for our Production Centers a few support tools for the programming activities. One of these was a tool to restructure conventional FORTRAN programs automatically for easier maintenance. Later we developed some verification tools to facilitate and improve the software testing process by identifying more exhaustive test cases. Our experience with these tools revealed that conversion onto our existing computer facilities was difficult. Some tools were developed for particular computer configurations, and they were not readily portable. Also a tool would be appropriate for a particular TYPE

of environment, and ill-suited for another, such as a tool appropriate for an interactive environment, rather than batch. While the functions the tools offered were useful, the tools were difficult to exercise. And they were error-prone (partially due to re-hosting), unfriendly, and the tools sometimes required an unacceptable level of computer resources to exercise (partially due to rehosting). Despite the expenditures of people, computer, and fiscal resources, it became uncertain that a high payoff would be derived, though there had been good anticipation at the outset of their acquisition and installation.

Contributing to the difficulties was the lack of familiarity with applying the tools (despite initial training) by our personnel. Due to the time stretchout of introducing the tools, those personnel that had been trained became mobile, necessitating retraining. Due to the conversion time, resource requirements and inexperienced personnel, management had to wait patiently for tool evaluation results that were significantly delayed. Then we suffered from not only inexperienced people, and computer resources mismatched to the tools, but also the erosion of our management commitment.

After finally developing and converting the tools, they became generally available to the users, but we found the tools were not frequently exercised because there were no programming standards and guidelines which required their use. The lesson is that it is not possible to create a MPE by quickly and simply introducing a few select tools. All the ingredients of an MPE are essential and must be sustained. Transitioning to an MPE will require time and resources.

Lesson No. 2: Plan for Evolution!

An important second lesson is that there is a requirement for a plan to proceed to an MPE which acknowledges distinct phases. As stated, initial experience convinced us that achieving an MPE was going to be a long process. (Incidentally this is consistent with other large organizations that have embarked on the same path.) Establishing standards, identifying and acquiring the appropriate computer resources needed, introducing a few mature tools and progressively building on these capabilities is a good approach. Prioritizing the steps and identifying and acquiring highpayoff tools and techniques need to be clearly mapped out. This should be done at the outset and reviewed along the way.

We accomplished a characterization of the existing DMA programming environment and the development of a plan to transition to an MPE [9]. The principal recommendations were:

Establish programming standards and guidelines

Train DMA people in modern programming practices

Optimize existing computer resources

Establish quality assurance practices

Establish configuration control

Establish a software tools group

It is noteworthy that these recommendations include all components of an MPE. Since this assessment, the Centers have developed programming standards and guidelines. We're in the process of acquiring more computer terminals to change the programming environment from batch to interactive. We're preparing the optimization of some software programs which are consistently large computer resource users. We've been training our personnel in modern programming practices and software engineering principles.

Lesson No. 3: Measure Something!

Once a plan is recognized, it is important to understand the payoffs which can be realized from its implementation because people, computer, and economic expenditures will be required. The management commitment is jeopardized without this understanding. Frequently tool acquisitions can provide a function which is needed by an organization and cannot be satisfied in any other manner. But tools and practices which supply seemingly non-quantifiable improvement in software performance, such as tools which generate better test cases, may require justification. Evaluation of the impact of these tools is expensive and difficult, and payoffs are not easy to establish beyond dispute. This problem is plaguing the software industry. To date we, and others, have relied on limited assessments of performance improvements. This brings us to Lesson No. 3. It is important to quantify software productivity improvements in the environment today so that you can measure software productivity progress tomorrow. Unfortunately much research remains to be accomplished on software metrics. However there are at least three approaches to this task today --extrapolate from the experiences of others, use controlled experiments, and build data bases for analysis.

While quantitative analyses of the benefits of certain

programming standards and use of software tools are limited, there are some comprehensive experiments on large software data bases which justify the use of MPEs. These results can be extrapolated to a particular environment, and, assuming analogous software development requirements, similar benefits can be anticipated. This may constitute a reasonable first approach to some kind of "measurement" of benefits.

Figure 4 is reproduced from a report by Barry Boehm on 63 software development projects [5,6]. It illustrates the high leverage cost factors which influence software productivity. For instance, applying modern programming practices can improve performance by 51% and using software tools can improve it by 49%. These factors are cited by Boehm as "multiplicative", indicating that if several factors are employed, benefits are synergistic.

Figure 5 is extracted from an extensive analysis performed by Walston and Felix of IBM on 51 software development projects [10]. Their assessment isolated the range in performance in software development attributed to certain modern programming practices. The three productivity columns indicate the performance experienced (in terms of delivered source statements per person month), when the technique was applied minimally, to an average extent, or extensively. That is, if none-to-little structured programming were used, 169 source statements were achieved per person month; if extensive structured programming were applied, 301 source statements per person month were achieved. There are significant improvements in productivity attributed to use of all practices.

Another analysis was performed by Brooks using 48 of the 51 projects examined by Walston and Felix [11]. Brooks was interested in isolating the effects of structured programming, and classified the 48 software projects into those which were unstructured (0-10% of structured code); those that were partially structured (11-89% of structured code); and those that were fully structured (90-100% of structured code). Brooks also distinguished the results on large versus small projects. The conclusion reached was that for both large and small size projects, structured code apparently affects productivity favorably. Under the conditions most adverse for software development, such as extremely large size, extremely complex projects etc, the ratio of the apparent gain in productivity ranges from 200% to over 600%.

The use of experimental team(s), one employing a practice or tool proposed for adoption and one(s) employing current practices or no tool has also been used to measure productivity

improvements. This can be costly especially in a production environment since twice the resources are expended to achieve a single assignment. Also the validity of the results can frequently be challenged since all factors are not necessarily equal. For instance, an argument can be made that the results are spurious since one team does not equal the other in skills (note Boehm's assessment of the influence of personnel capability on software productivity, earlier cited). If different projects are assigned to avoid duplication of resources, the argument of unequal tasks can be levied against results. However this technique has been used, especially in academic circles.

Research conducted at the University of Maryland in the nature of controlled experiments was performed to verify the effectiveness of a particular programming methodology [12]. Specific software development tasks were replicated under varying environments, and three distinct classes of developers were used, i.e., individuals, three-person teams, and three person teams using modern programming practices. Statistical analysis resulted in the interpretation that a disciplined methodology provides for a product at minimum cost, while the product itself (the software developed) at its worst approximates the product developed by an ad hoc team and in some respects resembles that of the individual.

DMA employed the team approach at each Center to evaluate the effects of modern programming practices and two tools, i.e., a Programming Support Library (PSL) and the FORTRAN Automatic Verification System (FAVS). The report generated did not produce conclusions using a statistical analysis, since sufficient data collection was not achieved. Also extracurricular events transpired during the course of the projects which affected the progress. Each Center's project was ambitious, and employed four person teams, including a Chief Programmer and Programming Librarian. An expert advisor was also utilized as "overseer". The conclusions reached were in the nature of a qualitative assessment and were cited as [13]:

More maintainable software was developed

Better project control resulted

Less throw-away code resulted

FAVS faciliated logic reviews of code generated and automatically detected unreachable code

PSL rehosting to DMA computers was detrimental

The creation of a data base to record expenditures of resources on software development/maintenance seems an ultimate step to measure improvements in productivity attributable to new practices and tools. A number of organizations have proceeded to this collection to support software cost estimating. (In order to estimate accurately the resources required for new software development, it is necessary to refer to the organization's historical data base). Frequently such a collection tabulates for each project the people and computer resources used in each phase of the life cycle -- the requirements, design, etc. It is possible to record what tools and practices were applied, project constraints, etc. However the product resulting from the project -- the software itself -- must be characterized in some manner. The software industry has yet to develop a universally accepted metric for its product, and there is much controversy on this subject, and much research required to resolve this issue. The use of delivered source instructions, or object instructions per person month seems an interim compromise [5,6,10,11].

Lesson No. 4: Introduce Tools Carefully!

The benefits that can be anticipated from software tools have already been discussed. Some insight into the difficulty of introducing tools has already been gleaned. There are several recommendations published as to methods for introducing tools into an organization [5,6,14]. The DMA experience is discussed here.

If possible, tools should be acquired only after some hands-on experience by limited numbers of users from the organization (again, the experimental use by a select team or individual). This should be accomplished by lease or remote access to sites hosting the tool. This avoids conversion costs and time.

It is advantageous to schedule training, not just initially, but over the entire experimental period. While the rudiments of tools use can be readily assimilated, applying a tool for maximum benefit is not so obvious, particularly when personnel have no prior experience, or the environment for which the tool was developed is different from that of the organization. We have discovered that interactions with expert personnel who constantly review the application of the tool is an extremely beneficial approach.

The maturity of the tool for use in a production organization is important. Unfriendly user interfaces, conversion errors, etc. must be removed before granting access to large numbers of users. Developmental tools can be exercised for

purposes of evaluating merit, or identifying modifications or enhancements for later development. Introducing unfriendly tools to large numbers of people is more detrimental than not providing the tool. Users are "turned off". Reinstating the tool after improvements is extremely difficult.

There is a need to assess or quantify the benefits of the tools, and this has already been discussed under measurement. Since resources will be required to train people in use of the tool, to enable exercise of the tool, and to maintain the tool, there must be some understanding of what is gained from its implementation. This ensures the management commitment.

Lastly, a formal role must be assigned for a "toolsmith", an individual or team responsible for the tool, and recognized as such by the organization. This provides for the formal and documented assessment of the tool, training for its application, and responsibility for its maintenance.

The Future!

We have a series of activities on-going or imminent which will take us to an MPE which has all the necessary components. The development of standards and guidelines, the acquisition of terminals, and optimization of computer resources have already been discussed.

With respect to introduction of tools into the organization, we are adopting the evolutionary approach. The FAVS tool, after extensive pilot team exercise, is targeted for production use. There is a counterpart to this tool which facilitates the testing of COBOL programs -- the COBOL Automatic Verification System. This is currently in development and is being exercised remotely. The knowledge derived from the FAVS application is being used so that we have every reason to believe that we will be able to transition the tool quickly to production use.

Meantime we are re-validating the view of the DMA programming environment in order to identify a minimum set of tools which would support each phase of the software life cycle, that is, a tool for requirements analysis, design, etc. Also included will be a tool for project management. An initial candidate set of tools has been exercised by individuals from each Center, and that evaluation generated so as to converge to a final recommended set. Then we shall proceed in accordance with Lesson No. 4.

The Centers are preparing plans for the introduction of tools into DMA, reviewing appropriate responsibilities for the

"toolsmith", and translating these into present organizational functions.

In the longer term we shall initiate an analysis of appropriate metrics to employ in our environment for quantifying the software product. We also will identify the contents of a database to facilitate our cost estimating and productivity assessments. Where possible, we hope to accomplish data collection automatically.

Summary

The lessons provided herein are in the nature of conclusions, and provide an informal methodology for proceeding to a programming environment. We feel our experiences have been similar to those of other large organizations which have embarked on this path. The goals to be reached are significant -- more maintainable and error-free software, more productive programmers, and improved software management.

Bibliography

1. Williams, Owen W., "Outlook on Future Mapping, Charting and Geodesy Systems", Proceedings of the Technology Exchange Week, May 1979, pp. 751-763.

2. Redwine, Jr., Samuel T., et al, "Candiate R&D Thrusts for the Software Technology Initiative", May 1981.

3. Reifer, D. J., "Software Acquisition Planning for the DoD Space Transportation System (Space Shuttle)," Proceedings of AIAA/DPMA Third Software Management Conference, Washington, D.C., December 1977, pp. 81-90.

4. Stokes, J. C., "Managing the Developing of Large Software Systems; Apollo Real-Time Control Center," Proceedings of WESCON 70, August 1970.

5. Boehm, Barry W., "Improving Software Productivity", COMPCON Proceedings, Fall 1981, pp 2 -17.

6. Boehm, Barry W., "Software Engineering Economics", Prentice-Hall, 1981.

7. Krygiel, Annette J. "The Application of Computer Software and System Engineering Research to the Defense Mapping Agency", Proceedings of the Technology Exchange Week, May 1979, pp. 110-127.

8. Houghton, Jr., Raymond C.,"Features of Software Development Tools", National Bureau of Standards Special Publication 500-74, February 1981.

9. Stucki, Leon; Brown, John; Hammond, Linda, "DMA Modern Programming Environment Study", RADC-TR-79-343 Final Technical Report, January 1980.

10. Walston, C.E. and Felix, C.P., "A Method of Programming Measurement and Estimation", IBM Systems Journal, No. 1, 1977, pp. 55-73.

11. Brooks, W. D., "Software Technology Payoff: Some Statistical Evidence", IBM Software Engineering Exchange, Volume 2, No. 3, April 1980, pp. 2-7.

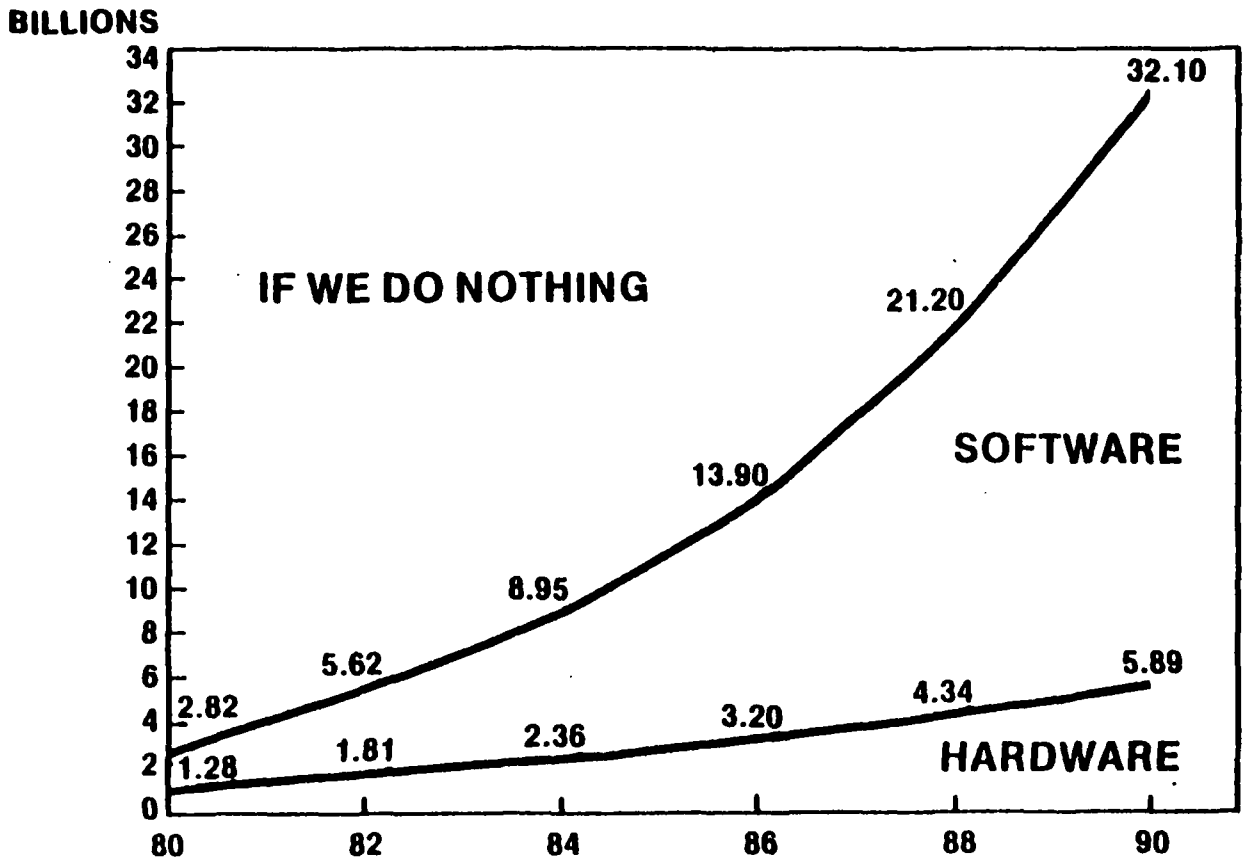
12. Basili, V. R.; Reiter, Jr.., R. W., "A Controlled Experiment Quantitatively Comparing Software Development Approaches", IEEE Transactions on Software Engineering, Volume SE-7, No. 3, May 1981, pp. 299-320.

13. Brittle, Emily, "DMAHTC Modern Programming Environment Pilot Project Evaluation Report", 30 July 1981.

14. Hecht, Herbert, "Preliminary Draft Guidelines for the Introduction of Software Tools Into a Programming Environment", National Bureau of Standards Report, April 1981.

Figure 1

DOD EMBEDDED COMPUTER SOFTWARE/HARDWARE



Source: Electronic Industries Association.

GROWTH IN SOFTWARE

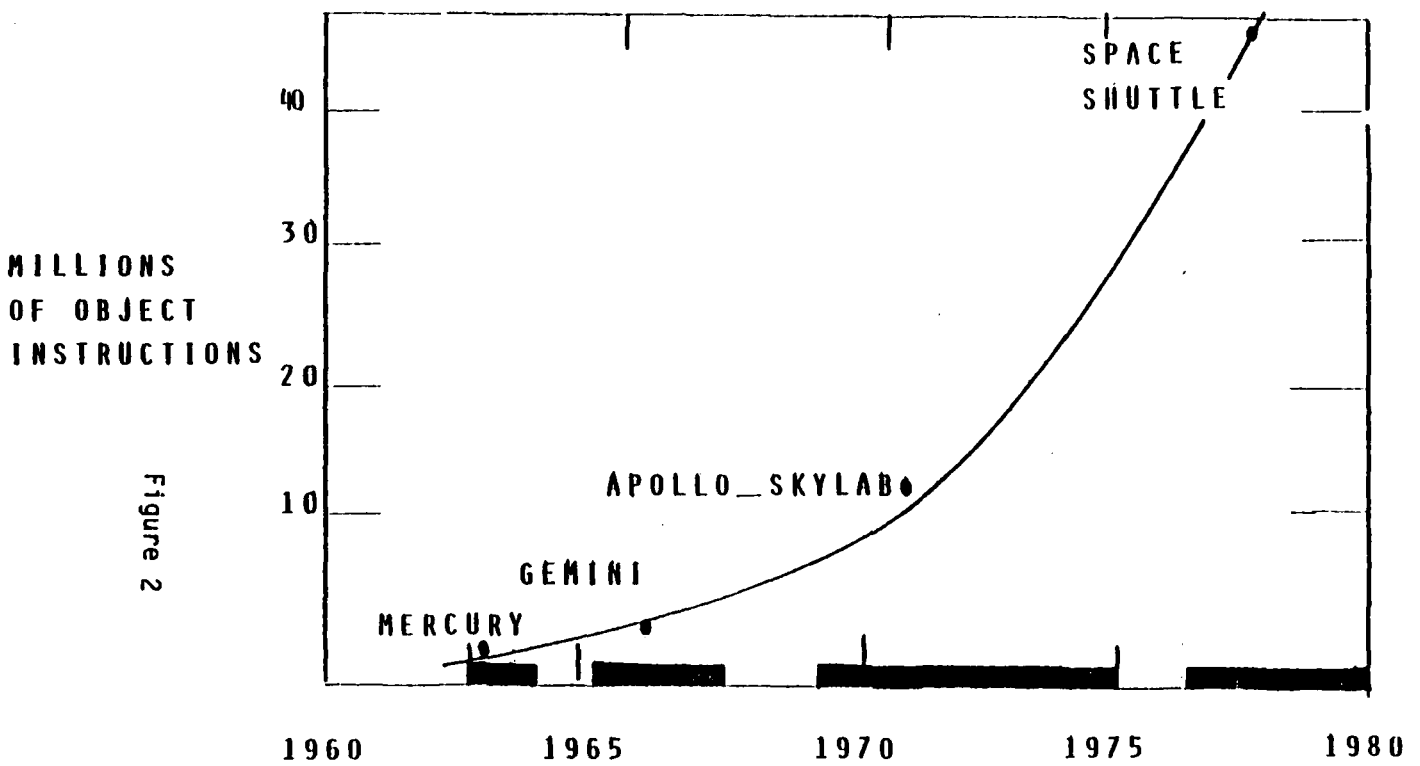


Figure 2

DMAAC SOFTWARE CONVERSION

UNIVAC 1100'S	NO. OF PROGRAMS	AVERAGE LINES OF CODE
*ALTERNATIVE 1	916	1417
ALTERNATIVE 2	290	1509
ALTERNATIVE 3	36	1756
IBM 7094'S	231	1015

* ANALOGOUS TO IBM 7094 INVENTORY

Figure 3

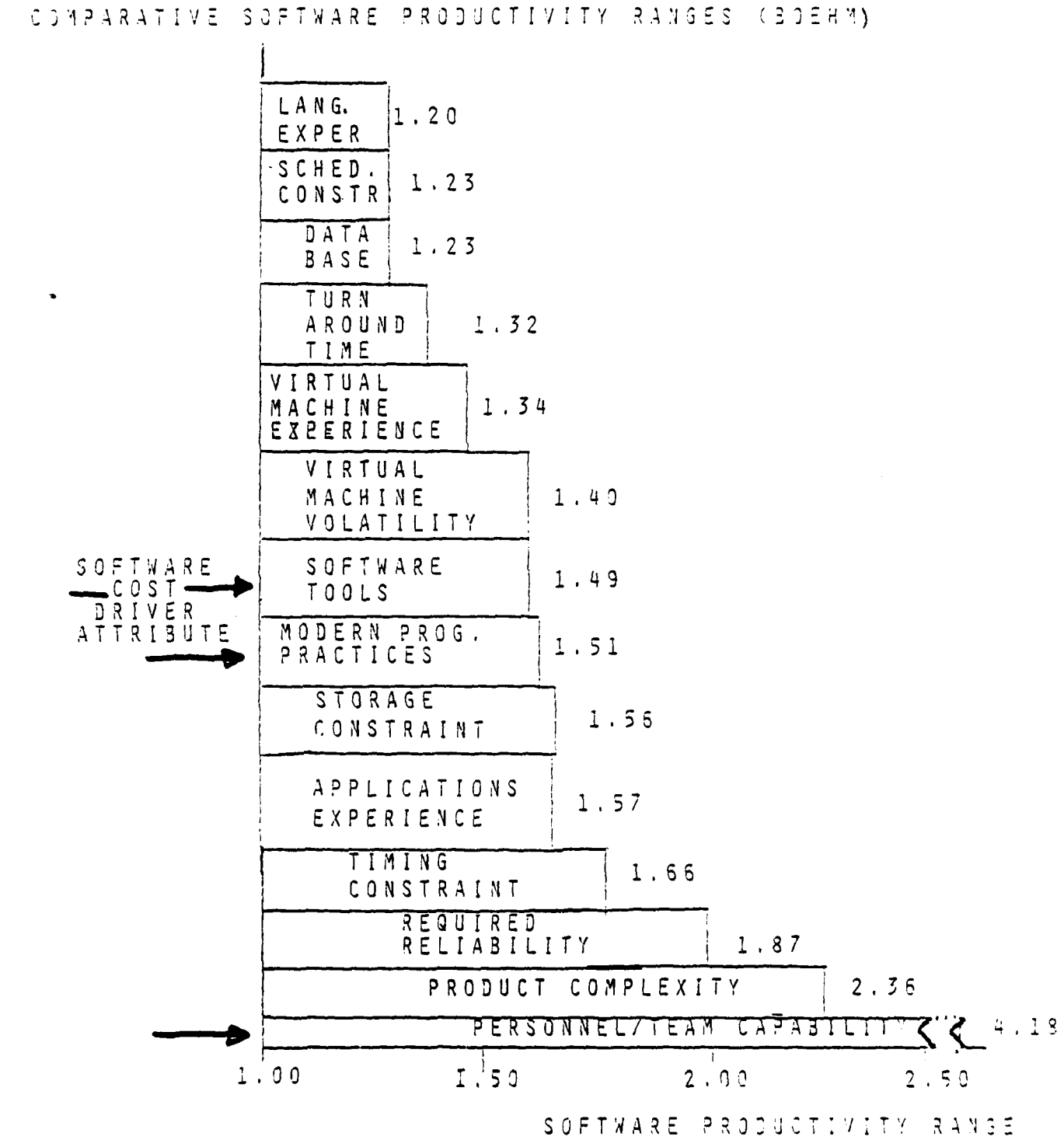

COMPARATIVE SOFTWARE PRODUCTIVITY RANGES (BOEHM)
LANG. EXPER
1.20
SCHED. CONSTR
1.23
DATA BASE
1.23
TURN AROUND TIME
1.32
VIRTUAL MACHINE EXPERIENCE
1.34
VIRTUAL MACHINE VOLATILITY
1.40
SOFTWARE TOOLS
1.49
MODERN PROG. PRACTICES
1.51
STORAGE CONSTRAINT
1.56
APPLICATIONS EXPERIENCE
1.57
TIMING CONSTRAINT
1.66
REQUIRED RELIABILITY
1.87
PRODUCT COMPLEXITY
2.36
PERSONNEL/TEAM CAPABILITY
4.18
SOFTWARE COST DRIVER ATTRIBUTE
1.00
1.50
2.00
2.50
SOFTWARE PRODUCTIVITY RANGE

Figure 4

PROJECT STATISTICS - WALSTON & FELIX

Variable	Productivity (Delivered Source Statements/ Entire Project Effort)			Percentage Difference (1st to 3rd)
Structured Programming	0-33% 169	34-66% –	67-100% 301	+ 78%
Top-Down Development	0-33% 196	34-66% 237	67-100% 321	+ 64%
Chief Programmer Teams	0-33% 219	34-66% –	67-100% 408	+ 86%
Design/Code Inspections	0-33% 220	34-66% 300	67-100% 339	+ 54%

Figure 5


FILMED