D 7 8 4 [illegible]

SOFTWARE ENGINEERING EDUCATION

Needs and Objectives

PROCEEDINGS OF AN INTERFACE WORKSHOP

EDITED BY
Anthony I. Wasserman
Peter Freeman

Springer-Verlag
New York Heidelberg Berlin
1976

Anthony I. Wasserman
Graduate Program in Medical Information Science
University of California
San Francisco, CA 94143/USA

Peter Freeman
Information and Computer Sciences
University of California
Irvine, CA 92717/USA

Library of Congress Cataloging in Publication Data

Interface Workshop on Software Engineering Education,
University of California, Irvine, 1976.
Software engineering education.

Bibliography: p.
Includes index.
1. Electronic digital computers—programming—Study and teaching (Higher)—Congresses. I. Wasserman, Anthony I. II. Freeman, Peter, 1941– III. Title.
QA76.6.I527 1976 001.6′42 76-45179

Printed in the United States of America.

ISBN 0-387-90216-3 Springer-Verlag New York
ISBN 3-540-90216-3 Springer-Verlag Berlin Heidelberg

To all of our children's superheroes--

Superman, Wonder Woman, Batman, the Bionic Woman,

and the others--

maybe <u>they</u> know how to build complex software systems!

PREFACE

"Software engineering" is a term which was coined in the late 1960's as the theme for a workshop on the problems involved in producing software that could be developed economically and would run reliably on real machines. Even now, software engineering is more of a wish than a reality, but the last few years have seen an increased awareness of the need to apply an engineering-type discipline to the design and construction of software systems. Many new proposals have been made for the management of software development and maintenance and many methodologies have been suggested for improving the programming process.

As these problems and solutions become better understood, there is a growing need to teach these concepts to students and to practicing professionals. As a prelude to the educational process, it is necessary to gain an understanding of the software design and development process in industry and government, to define the appropriate job categories, and to identify the fundamental content areas of software engineering.

The need for quality education in software engineering is now recognized by practitioners and educators alike, and various educational endeavors in this area are now being formulated. Yet, discussions we had had over the past year or so led us to believe that there was insufficient contact between practitioners and educators, with the resultant danger that each group would go off in separate ways rather than working together.

As a result, we organized a one-day workshop to bring together practitioners of software engineering and educators working to build quality curricula in software engineering. Our primary purpose was to help establish communication on a topic of common concern: software engineering education. In that manner, it would be possible for industry representatives to learn about current educational efforts and to influence the direction of new efforts. Similarly, it would be possible for educators to gain a better understanding of industry needs and the roles of software engineers in industry and government, so that existing educational programs could be altered as necessary to fit these requirements and new, more appropriate programs could be developed.

With this motivation, an Interface Workshop on Software Engineering Education: Needs and Objectives was organized and held at the University of California, Irvine, on February 9, 1976. Approximately 40 people, respresenting a balance among industry, government, and universities attended the meeting.

The purpose of this book is to share with a broader segment of the software engineering community the questions, ideas, opinions, and conclusions of the Workshop. The book is patterned after the successful reports on the seminal workshops held on software engineering in the late 1960's [Buxton and Randell; Naur and Randell]. It includes position papers prepared especially for the Workshop, other position statements prepared verbally at the meeting and later written up, edited comments taken from a taped transcription of the discussions, and a few papers presented elsewhere, but of particular relevance to the subject of the Workshop. As is fitting for a meeting dealing with an unexplored area, many of the papers are relatively informal and were written to present some tentative ideas or work in progress.

The discussions are not intended to be a literal transcription of the proceedings of the Workshop. Many of the discussions which took place were not well suited to verbatim reproduction. Also, there was a tendency at the meeting to drift away from and then return to some topics. An effort has been made here to convey a sense of the themes and issues by adding some material and by reordering some of the papers and comments, while remaining faithful to the comments of the individual participants. There is, of necessity, a certain amount of repetition. Many of the industry representatives identified similar needs; many of the educators are attempting similar programs. Many of these remarks, though, come from different perspectives and we have chosen to retain them for emphasis and reinforcement.

These proceedings consist of five major parts. In the first, we expand on the objectives of the meeting and present a thought-provoking paper by Robert McClure, based upon his keynote address to the IEEE Spring 1976 COMPCON. The second section presents a number of position statements concerning industrial needs for software engineering education. The third presents a sampling of approaches being tried or thought about in a number of universities. The fourth section contains a number of discussions from the meeting and our perceptions of the situation as it applies to the next steps in software engineering education. These four parts are followed by a highly selective annotated bibliography on software engineering, along with more details on the meeting and its participants.

We should note that these proceedings presume a prior familiarity on the reader's part with some of the topics and issues of software engineering. Many of the papers and comments refer to software engineering concepts with the assumption that the audience is familiar with them. It is our hope that the brief bibliography will be helpful to those who do not presently have this background.

We believe that these proceedings will be of interest to all persons involved in developing computer science and software engineering curricula, not only in universities, but also in industry. Furthermore, we hope that these proceedings can serve as the starting point for additional work in the development of coherent software engineering curricula.

We owe a deep dept of gratitude to our colleagues who came to the meeting and prepared papers. Of equal importance was the foresightedness of our publisher in understanding the importance of timely and wide distribution of these Proceedings. The office staffs in our respective departments assisted us greatly in the logistics of arranging the meeting. Brian Kernighan suggested the inclusion of the bibliography. The large majority of the typing was done by Edie Purser. We, of course, take sole credit for any errors. Above all, we give special thanks to Cheryl Burke for her work on arrangements before and during the meeting, and to Tina Walters for typing up all of the loose ends in the process of manuscript preparation.

Anthony I. Wasserman
University of California, San Francisco

Peter Freeman
University of California, Irvine

July 23, 1976

TABLE OF CONTENTS

SECTION I.

INTRODUCTION

SOFTWARE ENGINEERING EDUCATION: WORKSHOP OVERVIEW

Peter Freeman
University of California, Irvine
and
Anthony I. Wasserman
University of California, San Francisco

The primary objective of the Interface Workshop on Software Engineering Education was to facilitate communication between leading educations and practitioners on the needs and objectives of software engineering education. This purpose was captured well in the opening remarks delivered at the Workshop by Prof. William Parker, Assistant Vice-Chancellor for Plans and Programs, University of California, Irvine:

> The objective that you have set for yourselves in examining the relationship of an educational program to industrial needs in the area of software engineering is a manifestation of a general problem which any university must face these days. The university is faced with the problem of adjusting its educational programs to respond to the ever changing needs of society. Industry and the university have different objectives. If we do not have discussions, such as these today, we will end up going our own ways. So I would hope that during these discussions there would be a frank exchange of views so that the university can learn what the industrial needs are and the people in industry can realize the constraints within which the educational programs must function.

The objectives of the workshop included communication between both groups of people on the specific needs for software engineering education, the present and future roles of software engineers, and the value of current attempts to provide software engineering education.

Within the broad objective of establishing communication on any topics of relevance to software engineering education, some more specific questions were suggested for discussion:

- What is the role of a software engineer? Now? Five years from now?
- What are the duties of a software engineer in various organizations?
- What is the proper balance between formal education and employer training for software engineers?
- What are successful educational techniques for software engineering? Work/study programs? Software laboratory courses?
- At what level of education should software engineering be taught? Should there be a professional degree in software engineering?
- Who should become a software engineer? Should they be certified?

- What is the relevance of various topics--programming management, economics, program verification, etc.--for software engineering jobs?
- How should practicing software engineers receive continuing education on software engineering and related developments?
- How well prepared are current graduates for software engineering jobs?

While this is not an exhaustive list, it includes many of the questions that the participants wanted to address.

The organization of this report reflects the three main discussion themes of the Workshop: industrial needs for software engineering education, educational programs, and future developments. Within these three broad themes, we can identify several recurrent ideas or opinions that were heard repeatedly.

The software engineering practitioners generally agree that better educational programs are needed in order to provide them with people who can predictably carry out a range of activities. There seems to be agreement that the software engineer is a generalist and is something more than just a good designer or programmer. The need for solid communications skills and general problem-solving ability was stressed. Having a solid understanding of computer science fundamentals was perceived as a necessary basis for a software engineer. Familiarity with and ability to use management and economic techniques and knowledge was also considered to be one of the important attributes needed by software engineers. Finally, it seems clear that industry needs people who not only have all of these capabilities at the outset, but who will be able to adapt and grow both within software engineering and within the specific industry.

There are several trends that seem evident to us in looking at the university presentations. First, most universities are approaching the problem of providing software engineering education with caution. At present, most efforts are simply modifications or additions to existing curricula in computer science, industrial engineering, or electrical engineering, often no more than a single course. This cautiousness was reflected in most of the comments heard at the Workshop. Second, those schools which are attempting to meet the perceived needs of industry are beginning to make some attempts at software engineering education. While universities may eventually provide programs and turn out students which improve the quality of software development in industry, the current situation seems to be more one of trying to find ways to provide through educational programs what industry must now provide for itself through experience and in-house training. Finally, all of the university programs disucssed at the Workshop included a large amount of practical work in addition to more traditional academic work.

The discussions on the future centered around three trends. First, everyone agreed that there was a need for continued dialogue between all parties. This book, in fact, is a result of the general feeling that a broad range of people should be addressing these problems. Second, complementing the call for more dialogue was the feeling that increased cooperation between industry and the academic community is needed, both in educational and research programs. Finally, it seems clear that the development of curriculum guidelines for software engineering education is a necessary next step.

Within the context of a one-day meeting, it was possible to present the problems and constraints from both the industry and the academic side, to review briefly some of the plans and programs in software engineering education within the universities, to begin to evaluate these needs and programs, and to open up some of the lines of communication between industry and universities. However, it was not possible to deal with many of the issues at any depth and it was apparent that there is a great need to establish a continuing dialogue in order to work toward solutions to the various problems that were raised at the Workshop.

The Difference between a Programmer and a Software Engineer

Implicit in all of the discussions is the understanding that a software engineer is fundamentally different from a programmer or a computer scientist. A programmer is an individual whose primary responsibility is rapidly becoming the production of code. A programmer is typically familiar with one or more notations for programming and with the relevant features of the operating system on which the program will run. In general, though, the programmer plays little, if any, role in the development of system requirements or the specification and overall design phases of the software life cycle.

In contrast, a software engineer has responsibility throughout the entire software development process, with emphasis on tasks of specification and design. The software engineer must have a solid background emphasizing the design and construction of software systems. This background must be supplemented by a knowledge of managerial and economic issues, so that he/she can serve as a problem-solver, a designer, an implementor, a manager, and a communicator.

A computer scientist is still a different type of individual. The computer scientist possesses rigorous academic training which includes emphasis on a number of foundation areas, including automata theory, discrete structures, computer organization/architecture, and formal languages. The job of the computer scientist is to provide basic understanding of the underlying theory and concepts.

The contrast between a programmer and a software engineer can be shown still further by examining some of the typical tasks which each might perform in a software development effort. First, consider the several activities involved in programming at the lowest level (coding):

- devising local and concrete data representations for information;
- forming precise algorithms for doing necessary processing;
- taking care of housekeeping details necessitated by the particular programming system used (language plus run-time environment);
- choosing names, forming syntactically correct language statements, and making the program letter perfect.

By contrast, the software engineer is concerned with somewhat different activities:

- abstracting the operations and data of the task situation so that they may be represented in the system;
- determining precisely what is to be done by the software under design;
- establishing an overall structure of the system;
- establishing interfaces and definite control and data linkages between parts of the system and between the system and other systems;
- choosing between major design alternatives;
- making tradeoffs dictated by global constraints and conditions in order to meet varied requirements such as reliability, generality, or user-centeredness.

These distinctions indicate the qualitative difference between the activities of the software engineer and the programmer as perceived by most of the Workshop attendees. It appears that the concept of a software engineer is just beginning to have an impact and that it will be several years before the relationship between software engineers, computer scientists, systems analysts, programmers, and programming assistants is clearly defined.

SOFTWARE -- THE NEXT FIVE YEARS

Robert McClure
Consultant, Saratoga, California

This paper is based upon a keynote address given at IEEE Spring '76 COMPCON concerning the next five years in software practice and development. Since the software field is too large for any one person to know in depth, I drew not only upon my personal experiences, information from friends and associates, and published material, but also from the neighborhood tea leaf reader. Since the intention in this paper is to generalize, the result is clearly not directly applicable to specific companies, Universities, professional associations, or programming groups.

Decade #1 -- 1956-1965: The Great Years

In order to do a credible job at foretelling the future, I must first give a brief (and subjective) review of the past two decades. The first real decade in software comprised the years from 1956 through 1965. This covers the years from the introduction of the IBM 704 to the first deliveries of the IBM System/360.

This first decade saw the invention, development, and acceptance of substantially the entire body of programming lore as it is known today. At the start of the decade, large numbers of programmers were still actively engaged in writing programs in octal. The symbolic assembly program was known to but a few. Macro assemblers would have been considered revolutionary. FORTRAN (I) was a mere gleam in IBM's eye, even without separately compilable subprograms.

At the end of the decade, hundreds of systems programmers were actively writing the first PL/I compiler. COBOL had been out long enough for the government to conclude that the final answer to programming was at hand. ALGOL had gone through two iterations. In fact, most of the languages on Jean Sammet's list had already been implemented for the first time. Compiler building had progressed from being an arcane art to that of graduate student exercise. One software house was advertising that they could turn out FORTRAN compilers on an assembly line. Translator writing systems were available to everyone. Anyone who cared knew that parsing was no longer a problem, but that good code generation was a very difficult job. (Note. It still is!)

At the start of the decade, users of computers worked mostly "hands off". That is, they got their programs to execute by sticking a one card binary bootstrap loader on the front of an absolute binary deck, putting the cards in the hopper, and pushing the load button. Corrections were often made by carefully patching the deck and punching the checksum ignore bit.

At the end of the decade, enormous time sharing systems (Multics, TSS, and others) that would unleash unbounded computer power at the flick of a terminal key were thought to be just around the corner. And IBM had 5000 shock troops churning out code for OS/360.

It was a great time to be in the software game. Almost every day brought new wonders to behold. The future seemed bright indeed.

Decade #2 -- 1966-1975 -- or What Went Wrong?

If the first decade was like a drunken orgy, the second was more like a hangover. Early in the decade, even IBM conceded that the five thousand programmers involved in BIG OS were not making much progress. Amidst much embarrassment, mighty M.I.T. and Bell Laboratories admitted to being humbled by the Multics project. And Digitek bent its pick trying to turn its FORTRAN compiler technology loose on PL/I. In fact, the principal thrust was a monumental effort to secure the gains of the first decade, and to restore some order to an industry clearly out of control.

In any event, the second decade saw almost no inventions comparable to the first roaring decade. No languages of significance were developed during this period. (PASCAL and ALGOL 68 enthusiasts please don't write.) A similar claim can be made in the case of operating systems. As in the case of languages, several developments started in the first decade were not completed until much later in the second. It is a fact that substantially all of the operating systems in use at the end of the second decade had been started before 1966.

Although there were few inventions, several significant things did happen in this second decade. For example, there was finally a genuine acceptance among the applications programming fraternity of higher level languages, most notably FORTRAN and COBOL. In retrospect, this acceptance is due to two major factors in addition to the basic fact that it is a better way to write code. The first of these consisted of a very strong push from the US government in the direction of COBOL. By requiring compilers for all machines used for data processing, and requiring COBOL use for all appropriate applications, the US government guaranteed COBOL's success. The second of these factors is what is called the "Bandwagon" effect. In other words, it became fashionable for an engineer or scientist to know how to program in FORTRAN.

Interestingly enough, the major software segment to ignore higher level programming were systems programmers themselves. Neither of the two factors just mentioned were present.

The major surge in the second decade has been the great enthusiasm for "software engineering", "structured programming", "GO TO less programming", "top-down design", "chief programmers", "structured walk-throughs", etc. This seems to have been mainly triggered by the first NATO Software Engineering Conference held in 1968. At this now famous meeting a number of software elders gathered to complain to each other that they really did not know how to control the production of software, and to try to decide what if anything was possible to do about it. A careful reading of the proceedings of that meeting reveals a consensus that building software is not basically different from any other engineering work, and that the corrective actions required are the same. However, it was widely assumed that a new discipline had been created. This produced the spate of buzzwords referred to previously.

Unfortunately, my personal experience indicates that the great masses of programmers have not changed their habits. Their code is as unstructured as it ever was. Simply put, the great idea has not been reduced to practice.

Nor does there appear on the horizon the slighest sign of winds of change. So far there has not been the equivalent of a "COBOL edict". Without this, the "Bandwagon" effect cannot get started.

This second decade also saw a strong push toward language standardization. Almost every language of any significance saw the formation of at least one standards committee, some under government auspices, and at least one (TRAC) by private fiat. The careful observer may have noticed, however, that publication of a standard did not automatically guarantee that compilers would be made to conform to the standard. Only through major effort by Grace Hopper and staff in the case of COBOL, has even glacial motion toward standards conformity been recorded. The desire of almost all compiler writers (and their sponsors) to construct semi-permeable membranes to prevent the migration of code away from their machines has not really improved the portability of programs (and data) in the last ten years.

Although the second decade was not inspirational in the same way that the first decade was, it was nevertheless interesting. During this decade, software stopped being fun and games, and was recognized as a serious business. It was noted that operating systems, compilers, and other esoteric tools were necessary, but applications were really where the money was. This did not escape the major computer manufacturers, either. By the end of the decade the major software effort among the mainframers was overwhelming on the side of application packages rather than systems software.

The Next Five Years -- A Gloomy Outlook

At the start of the third decade, the major inventions are at hand, and it is unlikely that there will be any new major breakthroughs. The managerial techniques and controls needed to prevent the calamities of the mid '60's are in place. Nevertheless, at this moment software production in most shops is still running at half speed or worse. There is no reason to believe it will soon improve.

There are several reasons for a gloomy outlook for any major upgrading of the quality of software production in the next five years. Let me note here some of the most important ones:

1. There exists a vast overburden of programming history in the form of enormous amounts of code that must be modified and maintained. Much of this code is still in assembler language. Most of the rest is in COBOL. Almost none of it is "structured". The bill payers seem quite reluctant to scrap this code and redo it in civilized style.

2. There are rapidly growing numbers of small computers, both mini and micro. These seem to be programmed mostly in a style in vogue in 1955, with heavy emphasis on programming trickery, memory economy, and execution efficiency.

3. Languages which are suited for writing well structured programs (such as PL/I, ALGOL, or PASCAL) have failed to displace to any significant degree FORTRAN and COBOL, which are not.

4. A major manufacturer has recently introduced a portable computer that supports only BASIC and APL. Neither of these languages are satisfactory from a structured programming point of view.
Worse, APL seems to encourage a cryptic programming style that only other "one-liner" devotees could love.

5. There have appeared a few signs of a trend toward "unionism" in the ranks of practicing programmers, with concerns about seniority systems, reluctance to change, overtones of featherbedding, and a recent ruling that programmers were not "exempt" from overtime rules. These reactionary trends will clearly inhibit further innovation in actual software production.

The Next Five Years -- A Note of Optimism

So much for the gloomier part of the next five year outlook. There are also a few positive things to be said. Let me comment of two of these that I view as most significant:

The first of these is the new acceptance of simpler systems that are not intended to solve all problems for all users for all time at once. The concept that, for instance, a single operating system could simultaneously provide complete generality, utmost efficiency, absolute reliability, total security, and consummate ease of use for batch, time-sharing, and real-time users has been discredited. The early goals of Multics have been trimmed, and the latest developments in 370 operating systems foretell of fewer access methods rather than more. The grassroots enthusiasm for UNIX, an unadvertised operating system for the PDP-11 developed by Bell Laboratories, is a case in point. It's popularity is due to the fact that what it lacks in generality, it makes up for in simplicity and consistency. This abandonment of unconstrained generality is a very healthy trend, and will be speeded along by one of the major influences in the computing business, IBM, which now seems to understand the basic problem of over-complexity.

The second of the positive trends is the increasing willingness of business management to purchase (or lease) software. There seems to be a new awareness of the real cost of writing software in-house. To some extent, this has also been brought about by IBM. Their decision to unbundle, and then their introduction of the System/32, a machine meant to run canned application programs (and sold as such) has been very influential. I wonder how far the phonograph would have gone if all users had to record their own programs. With increasing markets, the software industry will become healthier and more competitive, and be forced to find ways to making their products more reliable, more effective, and cheaper. If there is a major move toward improved programming practices it must come from the independent software producers because they can benefit most quickly from improved productivity.

A Few Specific Predictions

A paper like this should always include a few specific predictions to throw rocks at, so here goes:

1. FORTRAN and COBOL will continue to reign supreme as application languages. The principal reason is that only these two have support from all the major computer manufacturers, and further only these are known by the majority of working programmers.

2. "Structuring" extensions to FORTRAN and COBOL will be mostly ignored. To adopt such extensions at this time invites spectres of non-portability. Until there are new standards, new compilers, and a new generation of programmers, structured programming in FORTRAN and COBOL will remain a curiosity.

3. Operating systems will mostly continue to be written in assembly language. This follows from the premise that most of the new computer designs will be at the micro end of the spectrum, and that systems for them will need to be as small as possible, etc.

4. In fact, almost all software for micros will be written in assembly language. Same reason as above.

5. Multiple processor systems will finally come to be widely used. This follows from the very low cost of processors of the micro class and the ease of constructing systems from a multiplicity of them.

6. No new language will gain wide acceptance. There has been a growing

feeling that the basic problems will not be solved no matter how good the programming language. Ergo, use the old language at hand. Even the heavy hand of IBM didn't gain major acceptance for PL/I.

To summarize, the next five years is not likely to see any major improvements in the way in which software is specified, designed, written, tested, or used. Outside of a few very sophisticated large users (this does not include all of the computer manufacturers) and a few progressive software houses, there is not likely to be any major practical adoption of the concepts of software engineering.