4-2-1990

Undergraduate Curriculum in Software Engineering

Harlan D. Mills

J. R. Newman

C. B. Engle, Jr.

Lecture Notes in Computer Science

Edited by G. Goos and J. Hartmanis

423

Lionel E. Deimel (Ed.)

Software Engineering Education

SEI Conference 1990
Pittsburgh, Pennsylvania, USA, April 2–3, 1990
Proceedings

Springer-Verlag
Berlin Heidelberg New York London Paris Tokyo Hong Kong

CONTENTS

An Undergraduate Curriculum in Software Engineering

H. D. Mills, J. R. Newman, C. B. Engle, Jr.
Florida Institute of Technology

Abstract

Software development and maintenance is only a human generation old, but is already practiced widely in government, business, and university operations on a trial and error, heuristic basis that is typical in such a new human activity. The term software engineering is also widely used as a commercial buzzword for marketing short courses and tools for specific heuristic approaches to software development and maintenance. But legitimate engineering processes, such as found in civil, mechanical, or electrical engineering, have foundations in mathematics and science that require four year university curricula, not three day short courses. Foundations in mathematics and computer science are just reaching the point where legitimate undergraduate engineering curricula are possible for software engineering. Florida Institute of Technology (FIT) plans to develop an undergraduate software engineering curriculum to provide students with new capabilities and standards for software development, evolution, and maintenance.

Software Goes Critical

This 'first generation' of consumers or users have encountered great frustration in dealing with the products of this human activity in software development. With all the people involved, with all the critical uses of software in both commercial and military operations, it is hard to remember that software development is only a human generation old. When civil engineering was a human generation old, the right triangle was yet to be invented. When accounting was a human generation old, double entry was yet to be invented. There are many more people in software in its first generation than there were in civil engineering or accounting in their first generations. But fundamental ideas still take time to discover and develop, and the very number of people in software today creates a massive intellectual inertia to make good use of fundamental ideas as they appear.

Typically, plans and schedules are easy to make for writing the software. The problem is in getting the software to work at all, and to do the right thing when it does work. Software has turned out to be more complex than it first appears. Twenty line programs, even hundred line programs in school problems don't seem hard. But twenty thousand lines of software, let alone a hundred thousand or million lines of software is quite a different matter. First many people will be writing small parts, a few hundred or thousand lines, which may work by themselves quite well. And some such parts may be written years later than others by complete strangers to earlier authors. But these parts must all work together, with no common sense run time help from their authors. That's where the complexity comes in.

So realistic development schedules involves engineering the software to execute in a completely reliable way under all circumstances. There is not enough time to build such software by trial and error. It needs to be engineered, with engineering checks and balances, dictated by an engineering discipline, complete with engineering inspections of work in progress. In fact, such engineering has been demonstrated in large systems in meeting schedules and budgets. For example in both the NASA space shuttle system (over 100 million bytes) and the Navy LAMPS helicopter and ship system (over 10 million words), every delivery over a four year period was on time and under budget [Mills 80]. But human society and institutions have had no long term, orderly experience or expectations in this engineering discipline because of the short time it has been needed.

The Role of Universities in Software Engineering

The current role of universities in software engineering is also in its infancy. During the present human generation, universities have begun to do research in and teach computer science. As a result, many universities now have computer science departments, which may be located in liberal arts, science, or engineering divisions. Such computer science departments teach computer programming and software system development as part of the computer science curriculum, but seldom teach computer programming as an engineering discipline. They seldom teach software maintenance or evolution in a serious way, even though that is what most of their graduates will be asked to do. There are many interesting approaches to teaching computer programming, using graphics, logic, text formats, but it is such a new human activity that there is still much to learn and be sorted out.

The next need is to move from computer science as a base into software engineering, just as more mature engineering disciplines have used sciences and mathematics as their foundations. The 1989 SEI Workshop on an Undergraduate Software Engineering Curriculum [Gibbs 89] pulled together much of the current thinking on the subject. This Workshop sponsored a set of position statements about the needs and pitfalls of putting a software engineering curriculum into place. [Deimel 89] makes the point that computer programming is, indeed, an important part of software engineering, and yet is not treated as seriously as it should be "under the assumption that entering students already know what they need to know about programming."

[Engle 89] discusses the difference between computer science and software engineering, noting that "software for large systems must be developed in a fundamentally different manner than software for small systems." [Ford 89] points out that a software engineering curriculum distinct from computer science is inevitable, but that change is slow and difficult in universities. That change has been difficult already in moving computer science into universities at the expense of established departments.

[Van Scoy 89] describes a specific plan for an undergraduate software engineering curriculum within an existing computer science program. The plan is described in five steps, namely: 1. Change the programming language taught to entering students (to a language which supports software engineering such as Ada); 2. Revise the sequence of courses taken by all freshman and sophomore computer science majors; 3. Add software engineering electives to the computer science major at the junior and senior levels; 4. Split the current computer science major into two tracks; 5. Develop distinct BS CS and BS SE programs. Van Scoy discusses a specific proposal for CS1 [Denning 88] using Ada "to facilitate the teaching of some software engineering ideas subtly and early." This proposal revolves around Ada packages at the outset, first in using packages, second in designing and implementing packages, then finishing with an introduction to Ada tasks. Most of Ada is introduced in the package framework, rather than beginning with procedures and functions before advancing to packages.

[Gibbs 89a] emphasizes the need for computer science fundamentals in undergraduate software engineering, outlining a model curriculum with two years each in Core Computer Science, Mathematics, Software Engineering, and Computer Science. At FIT, our plan is similar in some ways to that discussed in Van Scoy, but even closer to the model curriculum given by Gibbs. As Van Scoy suggests, we plan to evolve a software engineering curriculum within a computer science program. And as Gibbs recommends, we will begin with two years each in Core Computer Science and Mathematics, then finish with two years each in Software Engineering and Computer Science much in the spirit of the Gibbs model curriculum.

As a practical matter, we view Software Engineering as the necessary preparation for the practicing, software development and maintenance professional. The Computer Scientist is preparing for further theoretical studies or specializing in one of the many sub-disciplines such as graphics, artificial intelligence, etc.

Our approach takes its roots in the Denning report [Denning 88]. In addition we have been influenced by the curriculum report on software engineering made by the British Computer Society and Institution of Electrical Engineers [BCS/IEE 89]. Our approach to defining the curriculum for Software engineering is still developing, with our major emphasis at this time being devoted to the first two years.

Starting Right at the Freshman/Sophomore Level

In such a young subject as computer science or software engineering, many topics in graduate school are there because they are recent in origin, not more complex. Certainly our current students should not have to complete an undergraduate degree, spend some time working in the real world encountering the wrong way to create software, then return for a graduate degree before we teach them the right way to do software engineering. For example, we believe that the SEI Report on Graduate Software Engineering Education [Ardis 89] provides a good background for future undergraduate curriculum planning. The material and objectives presented in this report, seems natural to migrate down to the upper undergraduate levels as they become better articulated and agreed upon by the software engineering community. However, we expect to migrate the more formal topics of today's graduate software engineering programs right down to the freshman/sophomore levels.

In fact, the challenge in many science and engineering areas has always been to find simpler rigorous ideas more powerful than early heuristic ideas born out of immediate practice. For example, in mathematics, ordinary arithmetic preceded group theory, then set theory, by hundreds of years. Sets are simple and powerful, but took much human time to discover for effective use. In computer science today, the simple mathematical ideas have also arrived later than the initial practical heuristic ideas about program design. But we believe we can start university students learning the necessary engineering fundamentals to create correct and efficient software. This is possible through the use of mathematical foundations just because those foundations are simple, easily understood mathematical ideas in sets, functions, relations, not even requiring numbers in any necessary way.

We plan to create freshman/sophomore course work to introduce computer programs as mathematical objects from the start, with their expressions and analyses in programming languages as integral parts of the concept. A computer program is a rule for a mathematical function that maps a set of initial states to their final states. This course work is not about writing programs by trial and error. It is about the mathematical derivation of programs as rules for functions from formal specifications, which are mathematical relations or functions themselves.

We plan to make Ada the primary undergraduate curriculum programming language, even though other programming languages will be taught. However, the point of Ada as a programming language is its use to describe rules for mathematical functions that can be analyzed for correctness and performance with mathematical rigor. Other languages, such as assembly languages, Fortran, etc. can also be used to describe rules for these mathematical functions, as well. This shift from viewing programs as step by step instructions to computers (which they certainly are) to a new form of function rules brings mathematical rigor and engineering discipline into direct focus.

Ada as the Base Undergraduate Programming Language

Ada offers many advantages as a base language for the curriculum. It is rich enough to be useful for most programming concepts; it is a practical language, finding widespread acceptance in government, and thus, industry.

Ada is the first programming language which was "engineered" to support software engineering. Ada was not an evolutionary language
where new features were added to an existing shell. Rather, the Ada language was designed as any other software product. This effort was initiated by soliciting and obtaining a series of requirements for the new language. These requirements were refined by widespread public review into a set of specifications upon which a design could be developed. This preliminary design was also given widespread review and a final design for the language was approved, before any implementations of the language existed. This was a novel concept in the design of programming languages; obtain consensus on the requirements before implementing it! This shows that Ada was designed and engineered to perform specific functions, prominent among them was the support of software engineering concepts.

The concepts which Ada was expressly designed to support include abstraction, information hiding, localization, completeness, modularity, reliability, maintainability, reusability, and extendibility, among others. This list gives credence to the claim that the design of Ada was intended to support modern software engineering concepts and practices as we understood them. Arguably, the implementation of the language manifest in numerous compilers on numerous machine configurations, provides the much needed support for software engineering that has been missing in older languages.

The support of modern software engineering practices and concepts is very important. If a language is very rich in expressiveness, then it becomes less difficult and less error prone to translate the problem to be solved from the design space to the solution space. If the language is somewhat limited or constrained in its expressive power, then the mapping from the design space to solution space is more difficult. For example, if the design of a solution to the problem at hand conceptually requires the concept of parallelism, then if the language in which the design is being implemented supports parallelism, this portion of the solution can be directly mapped from the design to the implementation. If, on the other hand, your language does NOT support parallelism, then you must serialize your conceptual parallelism, which means that you must introduce additional complexity into the implementation to achieve the effect of your design. This necessarily perturbs the design and makes maintenance more difficult. In summary, the more powerful the language in terms of expressiveness, the more easily you can map the design to the implementation without introducing additional complexity.

In view of the foregoing, the rich set of constructs and programming expressiveness available in Ada make it the logical choice for our curriculum. While some may argue that the language is "too big" or too complex" for freshmen, we take the opposite view. We need only acquaint the student with that portion of the language which is necessary for them to solve the problems that we provide. In time, this will be the full language. What we obtain from this is the ability to go from simple sequential concepts to more complex ideas such as parallelism or genericity without having to transition the student from a smaller, less powerful language to Ada. They will have been using the same language since the first programming assignment!

Freshman/Sophomore Strategy

At a more general level, we see freshman/sophomore course work dealing with two main areas of core computer science, namely:

- Base Knowledge
  - Computer Operations
  - Computer Programming Languages
  - Data/Text Processing and Storage

- Base Skills
  - Program Analysis/Design
  - Algorithm Analysis/Design
  - Formal Syntax/Semantics Methods
  - Data Structures/Access Methods
  - Systems Analysis/Design

We regard the Base Skills as mathematics skills in the computer domain, as illustrated above with programs viewed as rules for mathematical functions.

FIT is on a quarter basis. We plan to organize the freshman/sophomore course work in the following sequence:

- Quarter 1
  - Program Analysis/Design in Characters/Sequences
  - Formal Syntax/Semantics of Sequential Programs

Quarter 2
- Program Extensions to Integers/Arrays/Records
- Algorithm Performance Analysis/Design

Quarter 3
- Program Extensions to Reals/Data Abstractions
- Formal Syntax/Semantics of Program Modules

Quarter 4
- Programming Languages/Assembler and High Level
- Generic Extensions to Programming Languages

Quarter 5
- Program Extensions to Concurrent Execution
- Concurrent Algorithm Analysis/Design

Quarter 6
- Program Extensions to Real Time Systems
- Real Time Algorithm Analysis/Design

Since the subject contents of these courses do not follow a traditional grouping, we need to prepare much of the material ourselves. The contents of the freshman/sophomore stream of quarter courses will be organized for convenient use in a stream of semester courses in other universities.

It may be of interest to compare this sequence of contents with that of [Van Scoy 89]. Van Scoy plans to introduce most of Ada in the first semester, certainly packages and tasks, in order to bring more realism into the course. "A relatively small unit on tasking is included in the first course in the belief that students bring to CS1 a view of the world that is essentially concurrent" [Van Scoy 89]. In the sequence planned above, Ada packages are not introduced until Quarter 3, Ada tasking until Quarter 5. Although Ada is the underlying programming language in both cases, the sequence of development is quite different. There are certainly many merits to Van Scoy's approach.

Our approach teaches mathematical foundations first, with programming languages used to express engineering designs based on mathematical reasoning. Those mathematical foundations take time to develop and understand. For example, in the FIT plan, the only data introduced in Quarter 1 is characters and sequences of characters, but formal syntax, semantics, and proofs of program correctness are fully developed for this simple data. Characters and sequences correspond to ruler and compass constructions in geometry, where mathematical proofs can be introduced in relatively simple contexts. But characters and sequences are fully capable of defining any operations possible in programmed computers, such as sorting, searching, or adding hundred digit numbers! Once such fundamentals are understood, integers, arrays, and records are introduced in Quarter 2, reals and data abstractions (Ada packages) in Quarter 3, in each case with expanded proof rules to deal with mathematical correctness.

Such an approach for the programming language Pascal appears in [Mills 87], a two semester text that develops formal syntax, semantics, and program correctness in characters and files in the first semester before introducing numbers and other data aggregates.

Freshman/Sophomore Coursework Contents

The new freshman/sophomore coursework at FIT is planned for introduction in 1990/91/92, the freshman coursework in 1990/91, the sophomore coursework in 1991/92.

A new text in two volumes is needed for the mainline material on Base Skills in the freshman/sophomore coursework beginning in 1990/91. Volume I (needed 1990/91) will introduce sequential programs and modules as mathematical objects for engineering analysis and design. Volume II (needed 1991/92) will continue with programming generics, concurrent, and real time programs and modules as mathematical objects for more complex engineering analysis and design.

Volume I will introduce sequential programs and modules as rules for mathematical functions, using the sequential parts of the Ada programming language. Volume II will continue the mathematical treatment of generic, concurrent and real time programs and modules, using the remainder of the Ada programming language. Program analysis and design thereby become mathematics based software engineering with a well defined language of application in Ada.

In more detail, the freshman/sophomore mainline coursework is planned in the following sequence:

Quarter 1

Program Analysis/Design in Characters/Sequences
Sets, relations, functions, predicates. Programs with boolean and character data and sequential files. Programs as rules for mathematical functions. Structured programs as expressions in an algebra of functions. Program specifications as mathematical relations or functions. Program correctness between a specification relation and a program function. Program design as creating rules for functions to meet specification relations.

Formal Syntax/Semantics of Sequential Programs
Formal syntax and semantics for structured programs with boolean and character data and sequential files. BNF for context free syntax. Uses of BNF in program specification and design.

Quarter 2

Program Extensions to Integers/Arrays/Records

Extension of scalar data to integers and their use in program analysis and design. Introduction of aggregate data in arrays, records and their use in program analysis and design. All language extensions in both formal syntax and formal semantics.

Algorithm Performance Analysis/Design

Analysis and design of algorithm performance in both time and space requirements as well as correctness. Understanding and creating high performance at machine levels as well as programming language levels.

Quarter 3

Program Extensions to Reals/Data Abstractions

Extension of scalar data to reals and their use in program analysis and design. Roundoff errors and algorithm design to contain and minimize problems of numerical approximation. Introduction of data abstractions for program modules of procedures and retained data. Data abstractions as state machines defined by transition functions with rules defined by the procedures.

Formal Syntax/Semantics of Program Modules

Formal definitions of modules in both syntax and semantics as extensions of programs. Extension of program correctness to module correctness. Relation to object oriented design/development.

Quarter 4

Programming Languages/Assembler and High Level

General properties and possibilities in assembler and high level programming languages. Translation between programming languages. Performance analyses on constraints of programming languages.

Generic Extensions to Programming Languages

Bases for more general, reusable programs and modules through use of programming generics and subsequent automatic development of specific designs by defining parameters.

Quarter 5

Program Extensions to Concurrent Execution
Addition of concurrency to program and module design with potential nondeterminism in execution that converts functional behavior to relational behavior. Extension of program and module correctness with relational behavior.

Concurrent Algorithm Analysis/Design
Analysis and design of concurrent algorithm performance in both time and space requirements as well as correctness. Understanding and optimizing concurrent performance at machine levels as well as programming language levels.

Quarter 6

Program Extensions to Real Time Systems
Addition of real time behavior to program and module design with potential nondeterminism in real time execution that converts functional behavior to relational behavior in time. Extension of concurrent program and module correctness to real time behavior.

Real Time Algorithm Analysis/Design
Analysis and design of real time algorithm performance in both time and space requirements as well as correctness. Understanding and optimizing performance in real time at machine levels as well as programming language levels.

Filling out the Undergraduate Curriculum

In our vision, the upper level coursework for undergraduate software engineering can be divided into three categories:

Junior/Senior Coursework

Programming Language Translators
- Assemblers/Linkers/Loaders
- Compilers/Interpreters

Base Systems
- Operating Systems
- Data Base Systems
- Real Time Systems
- Network Systems
- Graphics Systems

Advanced Skills
- Large Scale Systems Maintenance
- Large Scale Systems Development
- Statistical Quality Control of Software Production
- Information Systems
- Artificial Intelligence
- Law and Ethics

Our plan is to evolve the Junior/Senior coursework more deliberately, beginning 1992-93, from current offerings in computer science, with a new emphasis on both maintenance and development in large scale systems [Linger 88], statistical quality control [Mills 87a].

We expect to provide software engineering undergraduates with entirely new capabilities and standards in large software systems. These capabilities will stem from disciplined software engineers operating in concert in well disciplined teams with common methods. The mathematical basis in programming from the freshman/sophomore work will lead to new expectations in high performance, zero defect software to system specification. Zero defect software, in contrast with defect prone software expected and condoned in today's widespread heuristic methods, is very possible. For example, the 1980 Census system of reading, assembling, and communicating data from marked Census questionnaires to a central point from twenty geographic locations ran its entire ten months of operation with zero defects. This Census system involving over 25 KLOC of software earned its principal software engineer, Paul Friday, a gold medal, the highest award of the Department of Commerce [Mills 86]. The IBM Wheelwriter typewriter product, using three micro processors and 65 KLOC of software has been used by millions since its introduction in 1984 with zero defect performance [Mills 86].

Implementation at Florida Institute of Technology

The material presented in these six freshman/sophomore courses or 30 quarter hours, replaces what has previously been taught in twelve, traditional three hour courses at the same level. This introductory sequence, includes the fundamental body of knowledge any programmer needs to begin correctly and effectively solving real problems. Thus this becomes not only the introductory sequence for software engineers, but for all computer scientist, information systems specialists and to some degree, computer engineers. During the developmental stages of this program, we will have all of our Computer Science students take this sequence. Beginning in the junior year, the student can, through choosing the appropriate elective courses, specialize in what we call our Software Engineering or Information Systems options.

Although it will take three years for our first group of SE majors to complete our introductory sequence, we have already begun to modify our Junior/Senior courses to strengthen our software engineering emphasis. In addition to our existing courses in operating systems, data bases, compilers, graphics, artificial intelligence, analysis of algorithms, data communications, and ethics, we have added specific courses in large scale systems development, and advanced information systems analysis and design. Beginning in the fall of 1990 our present majors will be able to select a software engineering or information systems option, by choosing the appropriate elective courses from those we already offer. We intend to introduce courses in statistical quality control of software production, real time and distributed systems development and others as we can develop the necessary course material.

Software Engineering Techniques for Non-Computer Science Majors

If, as we project, this is the proper way to train software engineers to correctly solve problems using the available computing resources, it is also necessary to address the needs of other academic disciplines to introduce their scientists, engineers, businessmen etc. to these new techniques. We accept the premise that in developing large scale computer based systems, a team of specialists will be involved. In this environment it is reasonable to expect a greater depth of knowledge of the software engineering techniques by the software specialists. We recognize, however, that as problem solving using computers continues to pervade every academic discipline, we have an obligation to distill the essence of software engineering into a sequence of service courses for other disciplines. It is our plan to develop such a series of service courses based on our experience with our Freshman/Sophomore sequence.

Conclusions

The Computer Science Department at the Florida Institute of Technology will incrementally introduce a software engineering undergraduate degree, beginning in the fall of 1990. The curriculum will emphasize mathematical derivation of programs from formal specifications, which are mathematical objects themselves. From this formal basis the analysis, design and implementation of systems, programs and component modules will be developed. Ada will be used as the common basis because of its valuable properties and future widespread use. Upon this formal foundation, topics in software architecture, computer systems, software analysis and development/maintenance process techniques will be covered. Optional courses will be available at the Junior/Senior level to allow specialization. We will investigate the possibility of condensing the initial two year sequence to serve as service courses to other academic disciplines.

Once we have established the value and creditability of our Software Engineering program and have demonstrated the value of our service courses to other disciplines, we plan to address the issue of accrediting software engineering as a legitimate field in the engineering professions.

In summary, we expect future FIT software engineering graduates to be capable of engineering zero defect software at high productivity, and, of course, to schedules and budgets as well. Such engineering performance requires effective management of a rigorous software engineering process, not simply hoping for the best from heuristics and good intentions.

References

[Ardis 89] M. Ardis and G. Ford, "SEI Report on Graduate Software Engineering Education", in [Gibbs 89], pp 208-250

[BCS/IEE 89] The British Computer Society and The Institution of Electrical Engineers, "A Report on Undergraduate Curricula for Software Engineering", June 1989

[Deimel 89] L. E. Deimel, "Programming and its Relation to Computer Science Education and Software Engineering Education", in [Gibbs 89], pp 253-256

[Denning 88] P. J. Denning, D. E. Comer, D. Gries, M. C. Mulder, A. Tucker, A. J. Turner, and P. R. Young, "Computing as a Discipline: Final Report of the ACM Task Force on the Core of Computer Science", ACM Press 1988

[Engle 89] C. B. Engle, Jr., "Software Engineering is not Computer Science", in [Gibbs 89], pp 257-262

[Ford 89] G. Ford, "Anticipating the Evolution of Undergraduate Software Engineering Curricula", in [Gibbs 89], pp 263-266

[Gibbs 89] N. E. Gibbs (Ed.), Software Engineering Education, Lecture Notes in Computer Science, Springer-Verlag 1989

[Gibbs 89a] N. E. Gibbs, "Is the Time Right for an Undergraduate Software Engineering Degree?", in [Gibbs 89], pp 271-274

[Linger 88] R. C. Linger and H. D. Mills, A Case Study in Cleanroom Software Engineering: The IBM COBOL Structuring Facility, IEEE Compsac 1988

[Mills 80] H. D. Mills, D. O'Neill, R. C. Linger, M. Dyer, R. E. Quinnan, "The Management of Software Engineering", IBM Systems Journal, V 19, 1980

[Mills 86] H. D. Mills, "Structured Programming: Retrospect and Prospect", IEEE Software, November 1986

[Mills 87] H. D. Mills, V. R. Basili, J. D. Gannon, R. G. Hamlet, Principles of Computer Programming: A Mathematical Approach, Wm. C. Brown, 1987

[Mills 87a] H. D. Mills, M. Dyer, and R. C. Linger, "Cleanroom Software Engineering". IEEE Software, September 1987

[Van Scoy 89] F. L. Van Scoy, "Developing an Undergraduate Software Engineering Curriculum within an Existing Computer Science Program", in [Gibbs 89], pp 294-303

[Washington Roundup 1989] Aviation Week and Space Technology, February 6, 1989, p 17