Software Engineering: An Unconsummated Marriage

David Lorge Parnas, P.Eng

Software Engineering Research Group
DEPARTMENT OF COMPUTING AND SOFTWARE
Faculty of Engineering
McMaster University, Hamilton, Ontario, Canada L8S 4K1

Abstract

Although the first of many conferences on "Software Engineering" was held in Munich nearly three decades ago, communication between those who study the problem of building software and those who teach Engineering or work as Engineers has not been effective. Today, the majority of Engineers understand very little of the "science of programming". On the other side, the scientists who study programming understand very little about what "Engineer" means, why we have a self-regulating profession, how the profession is organized, and how engineers are educated. In spite of this mutual ignorance, today's engineers spend much of their time writing and using software, and an increasing number of people trained in computer science or mathematics are doing what work that meets the legal definition of Engineering.

This talk attempts to explain each field to the other and to suggest why, and how, the two groups should work together. The benefits of developing a new branch of Engineering, based on Software Science, will be described.

A Little History

The Software Crisis began 4 decades ago and continues today.

NATO sponsored two conferences in the late 60s.

The organisers called the topic "Software Engineering" in the hope of provoking interest from the Engineers.

It took thirty years, but they have finally shown interest.

A Bit Of Personal History

Educated as an engineer and taught engineering.

Wrote software to help in teaching logic design.

Interested in programming, almost self-taught.

Asked to teach programming course.

Shocked by the fact that we did not and could not teach how to program the way we could and did teach how to design hardware.

Set out to correct that problem.

Understanding the Problem

1. Why we have an Engineering Profession.
2. The growing role of Software.
3. Why we developed a "Science of Software".
4. Early History, Anecdotes and Lessons.
5. The Essence of an Engineering Education.
6. The Essence of Software Science.
7. Requirements: An example of where Engineering can help Computer Science.
8. Program Functions: An example of where Computer Science can help Engineering.
9. They recall cars, why not programs?
10. Software Engineering as part of Engineering.
11. What should we do?

Software Engineering

What should it be?

Just as Chemical Engineering is a marriage of Chemistry with a lot of Engineering areas such as thermodynamics and fluid dynamics, Software Engineering field should be a marriage of two fields:

- the science of Software, and
- the profession of Engineering.

Unfortunately,

The two fields hardly know each other.

A marriage without communication is sterile.

Currently, NSERC and NSF and other such agencies treat Software Engineering as part of Computer Science and not as a branch of Engineering.

We need a speciality within Engineering whose scientific basis is Computer Science.

Everyone who understands the structure of English should see that "Software Engineering" *should* denote a speciality within Engineering".

Why is Engineering a Regulated Profession?

- Bridges collapse; engines explode!

How do we know who is qualified to build them?

How do we know what a qualified designer should know?

How do we know if an institution teaches the required material?

Today, where bridges, engines, aircraft, power plants and medical devices are designed and/or controlled by software, these are exactly the needs in the field of software design.

Engineers in North America have:

- A registration system for individuals.
- An accreditation system for institutions.
- A way of keeping Engineers aware of their responsibilities.
- A mechanism for changing (albeit very slowly).

Software developers have none of the above.

If software developers and engineering educators consummate the marriage, the software development field will benefit!

The Pervasiveness of Software

Some figures from Philips in Eindhoven:

- A "High End" TV has 600 Kbytes of program.
- A VCR has 265 Kbytes.
- A hand-held telephone has 512 K.
- A car radio has from 64 - 256 kbytes.

If you flew here, your life depended on software.

If you have an ABS system, your brakes depend on software.

In modern cars, engines are controlled by software.

Nuclear plants are controlled by software.

Medical devices are controlled by software.

The dynamics of bridges are predicted by software.

These products are all products that traditionally would have been developed by qualified engineers.

Who is qualified to write that software?

- Nobody. There are no software engineers!

Today's Situation in Software Development?

Anyone can build and sell software!

Some institutions feel that they can offer "Software Engineering" programs without accreditation.

There is no general agreement on what a person involved in software development should know.

There are no software engineers!

There is a huge gap in understanding between academic computer researchers and those who actually develop software.

There are few researchers trying to fill that gap.

Much "Software Engineering" research is either camouflaged theory or project management research. There is also some psychological analysis of programmers and some (poorly controlled) experimentation. It is not Engineering research. It does not focus on design and analysis of the product.

We can talk about product and process standards, but teachers, students, and practitioners ignore them.

Software development is not a profession. There is little professional awareness.

Incompetent Software Developers

We have known how to write programs so that data representations are easy to change for more than 25 years.

Methods are included in many textbooks and taught in many courses.

The year 2000 was predicted.

Millions of lines of code have been written in which the representation of dates:

- will not work in the year 2000
- Is very hard to change.

Nobody made sure that basic methods were taught.

Anyone could declare themselves a programmer.

There was no licensing or certification.

There was no accreditation of educational programmes.

There is no standard that allows us to say that these programmers were negligent.

Licensing: Why we have it in Engineering and not in Science

Scientists' work is usually judged by peers:

- Research papers and proposals are peer reviewed.
- Work in laboratories is reviewed.
- Scientific work is often replicated in separate experiments.
- Scientific results are usually used by other scientists and by engineers who are able to detect problems.

Engineers' work is not always subject to such review:

- Results are built, not published.
- Results are often confidential
- Engineers often work alone.
- Engineers often deal directly with the public or their non-engineer representatives. Public safety and well being is affected.

That this is also true of software development; it was noted 20 years ago in another context. See: Richard A. DeMillo, Richard J. Lipton, Alan J. Perlis: *Social Processes and Proofs of Theorems and Programs*. POPL 1977: 206-214. Also published in CACM 22, 5, May 1979, 271-280

Accreditation: Why we have it in Engineering and not in Science

The accreditation process that we have in engineering would be considered an unacceptable violation of freedom by teachers in science.

- Scientists need not be licensed.
- Scientists can be specialists.
- There is no agreed "core body of knowledge".
- Major differences in what physicists or computer scientists know is not considered remarkable.
- "Model curricula remain suggestions.
- Evaluations are shallow and half-hearted.

Many Scientists and Computer Scientists seem unaware of the Engineering accreditation process.

- large visiting teams
- detailed evaluation procedure - about a week
- reading of class notes and student work
- ***critical to survival***

The CS accreditation has "no rigid standards".

The Engineering Style Of Education?

1. It is quite rigid - students have little choice.
2. Course contents are often prescribed - Professors have little choice.
3. It deals with rapidly changing fields by teaching fundamentals.
4. When teaching math and science, the emphasis is on how to apply, *not extend*, knowledge.
5. Stress on how to design.
6. Stress on professional responsibility and ethics.
7. Things are never taught just because they are elegant. Students expect to be shown how to use what to learn.
8. Theory (math) and practice are integrated.

There is a set of things that you can expect every <*>engineer to know.

Safety *demands* this style of education.

The Engineering Style Of Education?

An Engineer's education is *not* specialised!

- Chemical Engineers must know much more than Chemistry - fluid dynamics, thermodynamics,...
- There is often a shared first year for all branches.

There is solid agreement on the core material. Everyone has something that they want to add, but they all agree on the core that is now required.

Engineers learn that they must do "dog work", that they must be disciplined and thorough.

Systematic Analysis is stressed, case analysis

Classical Subjects such as:
differential equations, probabalistic analysis, optimization, numerical methods, testing, experimentation, information theory,...

Something is missing!

- Where is software?
- This was not important 30 years ago. Today it is pervasive.
- Teaching a language is not enough.

Why do we have "Software Science"?

In the 60s, we began to speak of a "software crisis".

We began to realise that we could not really teach software design. We taught syntax and showed programs, but did not teach design.

Some could design programs well, some could not. It was an art, not a teachable skill.

Everything depended on intuition. Math was only used for numerical issues, and by a few specialists.

A thirty year long "*crisis*" was beginning.

A world-wide research effort also began.

Today. the situation is quite different.

We have a Science of Programming.

We know a great deal about how to design and document software, but...

The "Software Crisis" continues unabated!
Incorrect software is the norm.
We must ask why!

Who Writes Software?

In a word, *everyone*!

- all kinds of engineers (mechanical, electrical, chemical,...)
- scientists (social and physical)
- psychologists
- mathematicians
- commerce/business majors
- computer science graduates
- . kindergarten teachers, lawn cutters,...

Who knows the Science of Software?

Bluntly, *few of the above*. Even CS graduates might not know it! (no core body of knowledge)

The software crisis continues because the communication between Computer Scientists and those who write software, including the Engineers, has been very poor.

Current software standards, are weak, superficial, and are not based on software science.

Process oriented "standards" are empty because there are no product/document standards.

Computer Science Education?

(1) It is not rigid - students have lots of choice.

(2) Course contents are not prescribed, vary widely.

(3) Fundamentals are far removed from practice and practical courses ignore them.

(4) It is very important to be "current".

(5) Little stress on how to design. (this varies)

(6) Little stress on professional responsibility.

(7) Things ***are*** taught just for their beauty.

(8) There is nothing that you can expect every Computer Scientist to know.

This is not an Engineering Education.

It shouldn't be an engineering education. We need Computer Scientists.

We also need engineers with a solid understanding of certain parts of Software Science.

Science Education compared to Engineering Education

Scientists must learn:

- what is true (an organised body of knowledge),
- how to *confirm* what is true, and
- how to *extend* knowledge in their field.

In other words, scientists learn science and scientific methods.

Engineers must learn:

- what is true, (an organised body of knowledge)
- how to *apply* that body of knowledge,
- how to *apply a broader area* of knowledge necessary to build complete products that must function in a real environment, and
- the *design and analysis discipline* that must be followed to fulfil the responsibilities incumbent upon those who build products for others.

The latter are often called "engineering principles".

These differences make sense!

If you are going to be doing specialised research, extending science,

- you can afford to be narrow but,
- you cannot afford to be out-of-date.

If you are going to be applying science to build reliable products,

- you rarely need the *very latest* research results, but
- you must have a broad understanding that allows you to take many factors into account when designing your product.
- Engineers must take responsibility for the correctness of their own designs.
- Engineers often have responsibility for the whole product and so must have some knowledge outside their speciality and the ability to work with other engineers.

Some Illustrations of the Gap

Two views of Finite Automata

The (Computer) Engineering view:

- Finite State Machines (FSMs) as a design tool
- How do you find the right state table?
- How do you minimise state tables?
- How do you design machines/programs from a description of a state table?

The CS view

- FSMs as a model of computation.
- What languages can an automata recognise?
- What problems can an automata solve?
- Which types of automata are equivalent in computational power?

Both views are valid but they are different!

Some Anecdotes:

- reviewing a book on algebraic automata theory.
- reducing the size of "minimal machines".
- Dave Parnas' "recent research results", FSMs.
- My 1965 office-mate.

More Illustrations of the Gap

Two views of Logic

CS: Logic embodies the rules of thought.

Eng: Logic is a way to describe discrete systems.

CS: Logics are defined by proof-rules. If you don't have proof rules it isn't formal.

Eng: Logics can be precisely defined by giving evaluation rules.

CS: Logic should be monotonic or strict.

Eng:Logic should be compact, predicates total.

CS: Endless debate about whether f(a) = f(a).

Eng: Who cares? 1/0 = 1/0 isn't ***true***.

Both views are valid, valuable but different!

Engineers (other than Computer Engineers) often do not know either form of logic.

How Do We Chose Research Problems?

Engineers: What problems do Engineers encounter in practice? How can they best be solved? How can we use our knowledge to design? Does it work in the "real world?"

Computer Scientists: What's a nice model (language)?

What properties of this model (language) can we prove? How should the world be? What would it be like if the world were this way? How can we simplify the world so that we can publish results?

Both views are valid but they are different and they lead to grave misunderstandings.

How Do We Chose Research Problems?

Some illustrative anecdotes

1. Concurrency
   - CS: Philosophy about the nature of time
   - Eng: What really happens when two events *appear to be* simultaneous? What should we do?

2. Specification Languages
   - Eng: What information is needed in a specification?
   - CS: How can we make the language more powerful?
   - Eng: How can we make the documents easy to read and check.
   - CS: Are there differences in expressive "power"?

   Is the language the object of study?

Some Oversimplifications:

- Scientists extend our knowledge of the world.
- Engineers apply knowledge by designing products.
- Engineering researchers extend our knowledge of how to design.
- Engineers find their problems in practice, their solutions in the literature.
- Computer scientists find their problems in the literature, and often find their solutions in new (bigger, faster) products on the market.

However, these things aren't central!

The most important issue is education: science education is, for good reasons, different from engineering education.

What kind of education should software developers receive?

Analogy: Two Views of Mathematics

Inspiration: the work of N.G. de Bruijn

Why is the mathematics used in Engineering and Applied Mathematics so different from that used in pure mathematics, logic, and Computer Science?

Math and CS: Mathematics is a structure on its own. Everything must follow from explicit statements.

Eng: Mathematics is a means of describing the world. General properties of the world need not be stated and stated and stated.

Again, both views are valid but, if we want our work to influence practice, shouldn't researchers be thinking about the Engineering view, be looking for compact notation? Shouldn't we be able to argue in context of accepted assumptions? Shouldn't we be checking mathematical results against reality?

Anecdotes: Lipton: "It doesn't matter that the hypothesis is false, the theorem is true." "It has to be obtuse to be formal."

This attitude is alive in many fields.

Something that Engineering and Computer Science Share:

An excessively narrow view of Engineering

Engineering is the use of science and technology to construct products that will be used by other people.

Engineering is no longer limited to building bridges, tools, or other physical products. Software is a product that is used by other people.

I am working with both the engineers and computer scientists on these issues. Engineers cannot imagine a branch of engineering that is not based on physics. Computer Scientists question whether software developers need to know physics or chemistry.

Those who work on information systems should be considered engineers as well, and educated accordingly.

Our health, wealth, and happiness depends on information systems as much as it does on more traditional engineering products.

Things Lots of Engineers Don't Know

1. Discrete mathematics and how to use it.
2. The Science of the Artificial (Simon).
3. Use of logic, modern algebra in programming.
4. Advantages of processes, process design.
5. Scheduling theory.
6. Synchronisation.
7. Meaning of "structured programming".
8. Modularisation via abstraction.
9. Recursion.
10. How to use graph theory.
11. Data structures/ data structure design.
12. Software hierarchies.
13. Loop invariants, monotonically decreasing quantities.
14. How to specify a program without writing one.
15. Information processing with perfect information but lots of it.

Engineers who write programs need this knowledge.

Things Lots of Computer Scientists Don't Know?

1. Control theoretic approaches, using time functions.
2. The role of a physical model of the environment.
3. Information theory - bit vs. binary digit.
4. Minimisation and optimization.
5. Dealing with noisy data.
6. Experimental validation of mathematical models.
7. The importance of reviews.
8. The responsibility to document.
9. When existence proofs are not of interest.
10. How to write for an engineering audience?
11. How to find the right mathematical model for some investigation? What does "right" mean?
12. When and what to test?

Engineering can help Computer Science

How can we document system requirements?

Identify monitored variables (m_1, m_2, •••, m_n)
Identify controlled variables (c_1, c_2, •••, c_p)
For each scalar variable, x, denote the time-function describing its value by "x^t"
The value of x at time t is denoted "x^t (t)"
The vector of time-functions
(v^t_1, v^t_2, ..., v^t_n)
will be denoted by "$\underline{v}^t$"

continued on next slide

NOTE:

- Conventional Mathematics does deal with real time.
- Continuous Models are often simpler than discrete (approximate) models.
- Studies of "air conditioner control", cruise control, etc. in CS, neglect fundamental properties of the physical "plant".

How can we document system requirements (cont'd)?

Describe the following relations:

RELATION NAT

- domain(NAT) is a set of vectors of time-functions containing only the instances of $\underline{m}^t$ allowed by the environmental constraints,
- range(NAT) is a set of vectors of time-functions containing only the instances of $\underline{c}^t$ allowed by the environmental constraints,
- $(\underline{m}^t, \underline{c}^t) \in$ NAT if and only if the environmental constraints allow the controlled quantities to take on the values described by $\underline{c}^t$ when the values of the monitored quantities are described by $\underline{m}^t$.

RELATION REQ

- domain(REQ) is a set of vectors of time-functions containing the instances of $\underline{m}^t$ allowed by environmental constraints,
- range(REQ) is a set of vectors of time-functions containing only those instances of $\underline{c}^t$ considered permissible,
- $(\underline{m}^t, \underline{c}^t) \in$ REQ if and only if the computer system may permit the controlled quantities to take on the values described by $\underline{c}^t$ when the monitored variables are described by $\underline{m}^t$.

We just took ideas familiar to Engineers and described them in terms familiar to CS people.

Result: Few really understood!

One way that Computer Science can help Engineers

Program-function tables.

Specification of a search program

($\exists$ *i*, B[*i*] =x)	($\forall$ *i*, ((1 ≤ i ≤ N) ⇒ B[*i*] ≠ x))

j' |	B[j'] = x	***true***	
present'=	true	false	∧ NC(x, B)

- The program sets two program variables called "j" and "present".
- There is one row for each variable set.
- There is one column for each case that must be distinguished.
- A "|" in the vertical header indicates that the variable's value must satisfy a predicate.
- When "=" appears, the expression gives a value, not a predicate.
- NC indicates variables that may not be changed.
  - Predicate values are distinguished from the values of boolean variables by font.

Many CS people tell me this is trivial, but most Engineers and Software Developers can't read it.

How can Engineers be Professional Today?

A Professional Engineer must vouch for the safety and correctness of a product, but...

They design that product using software that they did not write.

That software was not produced by a Professional Engineer.

They do not know the qualifications of the designer.

A software product usually carries a disclaimer where other products have a warranty.

What should they do?

- Should they do their calculations by hand?
- Should they write their own software?
- **Should we have real Software Engineers?**

Solving this problem would be reason enough to have software engineering programmes.

Professional Practice in the *Next* Millennium

Sample future professional practice exam questions

1. Antenna System Design

The President of a company sues the developer of field-strength prediction software because he missed a critical phone call with his new microcellular system. The antennae placement program predicted that there would be adequate field-strength throughout the building, but it was wrong. The program contained no check for convergence. Who was liable?

2. Conveyer Control Design

The program controlling a conveyer for molten metal goes into an infinite loop and the molten steel does a lot of damage. The Engineer is asked how she checked the loop for termination conditions. Was she negligent?

3. Remote Communications Controllers

A network of remote data collection stations gets into a deadlock situation requiring expensive visits to each of the stations. This repeats about every 5 months but did not happen in testing. Analysis shows that each controller has 10 "modes" and that a state-table analysis would have revealed the deadlock possibility, which could have been easily corrected. Who pays the costs?

4. Impellor Design

20,000 Impellor blades, distributed around the world. All must be replaced because the mesh size used in the design software was too large. The designer had trusted the software, did not know the method that it used, and did not check the mesh size. Is this required standard practice?

They recall cars, why not programs?

In Canada and the U.S., there are many car "recalls". If something wasn't right - it is fixed.

If I discover a defect in my operating system, I am told to buy the new version or subscribe to "maintenance".

Why the difference?

Most Software Designers would not know how to guarantee a program, either through proof or statistical testing.

A customer who demanded a guarantee would get a laugh. Programmers don't understand professional responsibility. Today's Engineers don't understand programs and how to certify them fit for use.

Being able to stand behind your project is not magic. It has a scientific basis. Software Engineers could do it too, but there are none.

Software Engineering as part of Engineering

In the U.S. there is a movement to try to establish Engineering as a profession on its own.

Canada also has disputes between programming groups and engineering societies over who can determine the qualifications of Software people.

This is the wrong approach:

We have to convince the Engineering Societies that Software Engineering is a (new) branch of engineering - Computer Scientists must work with them not against them.

The first step is to build mutual understanding.

Computer Scientists think that because Software Engineering is based on Computer Science it is Computer Science. That makes EE Physics.

Mathematics and Software Engineering

Some computer science pundits say, "Software Engineers don't use/like/understand mathematics".

If we believe them, it follows that:

Software Engineers are *not* Engineers

because

Engineers *do* use mathematics.

Moreover, Every programmer uses "formal methods" because programs are formal and programming is formalisation.

However, in software we use different mathematics!

We need discrete mathematics and notations suited for piecewise continuous functions (tabular expressions).

Why We Need Both, a Computer Science Programme and, a Computer-Science Based Engineering Programme

Many CS professors do not accept the need for a software engineering programme. They think that they are teaching software engineering now.

Many engineering professors say that you can program without knowing much computer science.

Computer science has become *mature* enough that it is routinely applied to new tasks of importance to the public. We need an engineering programme for the students who will do that.

Computer science is still a *young* field with many open issues. We need computer science graduates to extend and refine our understanding, to build new tools, and in general to explore the characteristics of computer systems and explain how to build them.

There are distinct career paths, two distinct sets of students, and we need two distinct programmes.

Why The Distinction Is Hard To See?

Nature abhors a vacuum! (Spinoza)

There have been no Engineering Programmes specialising in software, but the need for such programmes was great.

There have been several responses:

- Engineers and others have learned about software in *ad hoc* ways after graduation.
- Various educational programmes have included some software courses in their programmes and software has also been taught as part of other courses
- Computer Science departments have tried to fill the gap by including so-called "systems" or "applied computer science" courses in their course offerings.

Many see a degree in Computer Science as the best path towards such careers. There is no other choice.

Today's computer science programmes are rarely pure science programmes, but neither are they programmes that could win accreditation as Engineering programmes. They are compromises that do neither job well.

Accreditation is not important to CS departments. It will be important for Software Engineering departments.

Misuse of the Name

In 1967 and 1969 a group of researchers tried to stimulate the interest of Engineers and Engineering Societies in Software.

They failed! The Engineers ignored them.

The Science needed as a basis for Software Engineering was developed anyway.

Courses in that Science were sometimes called Software Engineering.

Specialists in that science sometimes called themselves “Software Engineers” as did a broad variety of people (including some without any technical degrees) who were developing software.

We need to protect the public from incompetent software developers.

We need to distinguish between the science and the profession.

It is important to recognise the legal status of the title Engineer and not to have programmes called Engineering that will not qualify their graduates for licensing as an engineer.

Some Thoughts on Curriculum

If all other engineers know something, software engineers should know it too.

The intersection of the engineering fields is small and contains very basic material.

We must focus on fundamentals, *viz*.things that were valuable a decade ago and will be valuable decades from now. The field is moving too fast to chase it. It is good to use recent tools in laboratory work, but they cannot be the subject of a course.

There must be emphasis on design principles and their mathematical basis. Students must be taught how to design and how to evaluate designs.

Theory and Practice must be integrated in each course.

Basic Mathematics

Calculus
Mathematical Foundations and Discrete Mathematics
Logic
Matrix Algebra
Differential Equations
Probability and Applied Statistics.

Basic Science

Chemistry
Physics

Engineering Topics

Electric Circuits, Electronics and Instrumentation
Physical Systems and their Analysis
Communications and Signal Processing
Engineering Economics
Communication Skills - Explaining Software
Complementary Studies
Safety Training
Engineering Design and Communication

Computer Science Fundamentals

Fundamentals of Computation

Basic Software Topics

Software and Social Responsibility

Systematic Programming

Selection and Design of Computer Algorithms and Data Structures

Software Design I - Programming and Specifications

Software Design II - Design of Sequential Software Systems

Software Design III - Design of real-time, distributed, and concurrent software

Digital System Principles/Logic Design.

“Architecture” of Computers and Multi-Processors

Numerical Computing

“Machine Level” Computer Programming

Advanced and Specialised Topics

Performance Analysis of Computer Systems
Simulation and Modelling Methods
Optimization Methods, Graph Theoretic Models, Search and Pruning Techniques
Design and Evaluation of Operating Systems
Design and Selection of Programming Languages
Design of Real-time Systems, Computerised Control Systems
Computer Communications
Design of Parallel/Distributed Computer Systems and Computations
Design of Human Computer Interfaces
Data Management Techniques
Computers in Communication Systems
Evaluation, Selection, and Design of Computer Assisted Design (CAD) Tools
Robotics and Intelligent Manufacturing
Computerised Image Processing (Computer Vision), Pattern Recognition
Geographic Information Systems
Computer Security
Digital Signal Processing
Business Applications of Computers
Information Retrieval
Computational Geometry
Senior Thesis: One final design experience.

Most of this is "software science" but it must be taught for engineers, i.e. they must be taught how to design using these ideas.

Topics That Are (Deliberately) Missing

Current languages and Systems.

Compiling Techniques.

Programming tricks that are language dependant.

Specification languages such as Z, VDM, B, SDL, Estelle, and their many variants.

Computational Models of Semantics.

Specific Tools as the main subject of a course.

Buzzwords like O-O, client server, agents,....

Software Development Process, CMM, ISO 9000.

Configuration Management.

Software-Specific Project Management.

Artificial Intelligence.

Anthropomorphic terms and analogies.

The "17 operas that Verdi wrote" (Leo Aldo Finzi).

A Possible Transition Plan

The length of time a student spends in University or Institute of Technology is limited.

Politicians (everywhere) want to reduce that time and to limit it strictly.

Engineering is a full education - so is Computer Science. Neither has room for much more.

Until we have learned to understand each other and develop joint curricula we have to develop "extra year" curricula. We need two:

- An extra year for engineers to teach them software science and how to use it.
- An extra year for students in Computer Science to teach them more about Engineering.

The lack of Computer People and Engineers who can communicate is a frequent complaint by industry. We must produce such people.

Conclusions

1. It is time for engineering to take up the challenge of 1967.
2. We need licensed Software Engineers
3. We need accredited Software Engineering Programmes.
4. It will require the marriage of software scientists and Engineers. Neither can do it alone.
5. There is meaning to the word "Engineer". Lets not destroy it by misusing it.
6. Both Engineers and Computer Scientists have ignored the problem for too long. Neither has defined the core body of knowledge, the knowledge that we expect of all software engineers.