Academic Legitimacy of the Software Engineering Discipline

Daniel M. Berry
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213
U.S.A.

Abstract: This article examines the academic substance of software engineering. It identifies the basic research questions and the methods used to solve them. What is learned during this research constitutes the body of knowledge of software engineering. The article then discusses at length what about software makes its production so difficult and makes software engineering so challenging an intellectual discipline.

1 Introduction

In academic life, there are occasions that prompt an examination of the foundations of what is claimed to be an academic discipline. These occasions include the consideration of hiring, promotions in general, promotion to tenure in the specific, acceptance of a student's thesis topic, planning of a degree program, and planning of curriculum. The discipline of software engineering is generally housed in a department of computer science, computer engineering, electrical engineering, applied mathematics, mathematics, or management. All too often, the examination of the foundations by these departments has yielded conclusions indicating a misunderstanding of the nature of software engineering. The results have been denial of hiring, promotion, and tenure, rejection of topics, and poor or no degree programs or curricula. The purpose of this document is to examine some of these conclusions, the questions that prompt them, and the reasoning that leads to them, and finally to refute the claims that software engineering lacks substance. The refutation consists of a description of the content of software engineering, its research problems and methods, an explanation of the rationale behind these methods, and then a lengthier discussion of the complexity of software and of its intellectual challenge.

Among the questions that are dealt with in this article are the following.

1. What is software engineering?
2. What is software engineering research?

3. Why is software engineering research necessary?
4. How should software engineering research be done?
5. Why is software engineering research done in the way it is done?
6. How can software engineering research be evaluated?
7. Why is it necessary to write software artifacts for the research?
8. Is programming itself research?
9. Why is it necessary to consider human behavior and management issues?
10. What is the formal difficulty of programming?
11. What are the foundations of software engineering?
12. What is the academic substance of software engineering?

In one sense, there should be no problem of legitimacy of software engineering. Software engineering is an engineering field. It is well established that engineering is an academic discipline. There are PhD-granting institutions that focus entirely on engineering and have other departments mainly to complete the education of their engineering students. However for better or worse, currently software engineering programs are usually housed in other, more established departments, usually computer science. The problem arises when the housing department applies the standards appropriate for its mainline interests in considering software engineering issues, hiring, and promotions. Perhaps the long-term solution is for software engineering academics to form their own departments; that is a topic for still another paper. However, and perhaps from misplaced sentimentality, I still believe that much is gained in both directions when software engineering is housed in another department such as computer science. Consequently, this article is written on the assumptions that software engineering is housed mainly in computer science departments, and that this placement is good and worth continuing. Certainly, the first assumption reflects what *is* the case at most places.

This article does not consider the situation of a software engineering program that is housed in an engineering department or school, simply because the acceptance problem appears not to be as severe in engineering programs.

This article is derived from a longer technical report of the same title issued at the SEI; in the rest of this article, this technical report is called “the report”. The report contains all the in-depth discussion that has been excised from this article to meet size requirements. A hard copy of the report can be requested from SEI Publications or the Defense Technical Information Center (DTIC). A PostScript version of the report, suitable for printing locally, may be obtained by anonymous ftp from `ftp.sei.cmu.edu` from the file `tr34.92.ps` in the directory `pub/documents/92_reports`.

2 What is Software Engineering?

First, it is necessary to define software engineering, if for no other reason than to allow someone to know if he or she is a software engineering academic. There are many definitions of software engineering, almost as many as there are people claiming to be software engineers. My own operating definition of any field is that the field is the sum total of all the work that anyone claiming to be in that field is doing. I prefer to include topics that are on or beyond the fringe of my perception of the area than to exclude someone just because a definition that I came up with happens not to mention a certain topic. Having said, in effect, that a definition is useless and dangerous, I now give not just one, but two definitions.

2.1 Definitions of Software Engineering

> “The use of the engineering method rather than the use of reason is mankind’s most equitably divided endowment. By the *engineering method* I mean *the strategy for causing the best change in a poorly understood or uncertain situation with the available resources*; by *reason*, I mean the ‘ability to distinguish between the true and the false’ or what Descartes has called ‘good sense.’ Whereas reason had to await early Greek philosophy for its development—and is even now denied in some cultures and in retreat in others—the underlying strategy that defines the engineering method has not changed since the birth of man.”
>
> — Billy Vaughn Koen [Koen85]

Two definitions are offered, amplified, interpreted, and elaborated to yield the desired list of subfields.

The first definition comes from the *1990 SEI Report on Undergraduate Software Engineering Education* [Ford90], which quotes an unpublished definition of software engineering developed by Mary Shaw, Watts Humphrey, and others. This definition in turn is based on a definition of engineering itself from the same source.

- “Engineering is the systematic application of scientific knowledge in creating and building cost-effective solutions to practical problems in the service of mankind.”
- “Software engineering is that form of engineering that applies the principles of computer science and mathematics to achieving cost-effective solutions to software problems.[page 6]”

Ford elaborates and interprets this definition. A paraphrase follows.

- For software, creation and building are not enough; the software must be maintained. Therefore the word “achieving” has been used in place of “creating and building” to cover the entire software life cycle.
- In fact, software engineering is not limited to applying principles only from computer science and mathematics. As any other engineering discipline, it is based primarily on principles from one or more disciplines, but may draw on whatever principles it can take advantage of. Since most software is developed by teams of people, these principles may very well include those from psychology, sociology, and management sciences.
- “Cost-effective” implies accounting not only for the expenditure of money, but also for the expenditure of time and human resources. Being cost-effective also implies getting good value for the resources invested; this value includes quality by whatever measures considered appropriate. For software, these measures include those applying to the software itself, such as correctness, reliability, and performance, and those applying to the production process, such as timeliness.
- A particular piece of software is not considered an acceptable solution unless it is correct, and reliably so, and its performance is acceptable, among other things. By “correct” is meant only that the program behaves consistently with its specification; by “reliable” is meant that the software behaves correctly over a given time period with a given probability; and having “acceptable performance” means that the program runs on available machinery and finishes or has response time consistent with the tasks at hand and human patience to deal with them. Presumably, the specification of the program, the required time period and probability, and the required response time are all according to the customer’s needs.

The second definition comes from the *IEEE Standard Glossary of Software Engineering Terminology* [IEEE91]. This definition, too, is based on a definition of engineering itself.

> “**engineering.** The application of a systematic, disciplined, quantifiable approach to structures, machines, products, systems, or processes.”
>
> “**software engineering.** (1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software. (2) The study of approaches as in (1).”

It is worth elaborating and interpreting this definition too.

- This definition recognizes the importance of maintenance to software engineering.
- This definition considers that the approaches used must be systematic, disciplined, and quantifiable rather than just be based on computer science and mathematics.
- This definition includes in software engineering the study of and search for approaches to carrying out software engineering activities, i.e., the study of and search for methods, techniques, tools, etc.

Even after these two definitions, there are important components of software engineering, and indeed engineering, that have not been mentioned. As observed by Robert Veltre of the SEI, engineering clearly stands at least partially on the foundations of scientific knowledge. However, there exist huge gaps in knowledge between that produced by scientific research and that employed by engineers in engineering a product. Scientific knowledge is but one component, and it is usually a significant component only during *radical*, entirely new design, as opposed to *normal* design. Most engineering knowledge is derived from the following sources.

- Invention (radical design)
- Theoretical engineering research
- Experimental engineering research
- Design practice (reflective practice)
- Production
- Direct trial

In view of the two definitions and the discussions that follow, the following working definition of software engineering is adopted for the purposes of this document.

1. Software engineering is that form of engineering that applies:

    - a systematic, disciplined, quantifiable approach,
    - the principles of computer science, design, engineering, management, mathematics, psychology, sociology, and other disciplines as necessary,
    - and sometimes just plain invention,

    to creating, developing, operating, and maintaining cost-effective, reliably correct, high-quality solutions to software problems.

2. Software engineering is also the study of and search for approaches for carrying out the activities of (1) above.

In the interest of briefer sentences in the sequel, the phrase “quality software” means cost-effective, reliably correct, high-quality solutions to software problems. Since cost-effectiveness includes performance, “quality software” also means software that is performing adequately for its purpose. The word “producing” means creating, developing, operating, and maintaining; and “underlying principles” means principles of computer science, design, engineering, management, mathematics, psychology, sociology, and other disciplines as necessary.

2.2 Subfields

On the basis of the working definition, the software engineering field can be perceived as consisting of the following subfields.

1. Theory of programs and programming — mathematical theories of program verification, of program meaning with an eye toward verification, and of program construction, derivation, generation, and transformation

2. Formal methods — the application of the theory developed in (1) above to the production of quality software for actual use

3. Technology — the discovery, development, and validation of the effectiveness of software tools to help carry out one or more steps of the programming process, including the application of formal methods, for the purpose of improving the ability of the user to produce quality software

4. Methodology — the discovery, development, and validation of the effectiveness of nonformal but systematic manual procedures for the purpose of increasing the ability of the applier of the procedures to produce quality software

5. Management — the discovery, development, and validation of the effectiveness of managerial techniques to help people and groups of people produce quality software

6. Production of software artifacts — the actual development of particular instances of quality software

There are two common themes running through these field descriptions. All are concerned in some way with the achievement of quality software. All those involving tools, methods, and techniques talk about the discovery, development, and *validation of the effectiveness* of the tools, methods, and techniques; it is not really legitimate to claim that tools, methods, and techniques work unless something has been done to demonstrate that they work.

These subfields cover a rather large spectrum. Nonetheless, from the software engineering point of view, all subfields are necessary for a broad-based attack on the problem of producing quality software. The theory tells us at the very least what is and is not possible and provides a formal basis for any claim of correctness for any particular program. On the other hand, the formal methods do not seem to scale up well to large systems, and do not even apply to all aspects of these systems. Thus, software development technology, informal system development methods, and management techniques become important. The importance of management stems from the experimental observation that people issues have a bigger impact on the cost and time to produce quality software than do technical issues, and it is only by building particular software artifacts that we get to know if our tools, methods, and techniques work.

The subfields are listed in order of decreasing perceived academic legitimacy. The first two are quite well accepted as academically legitimate and, in fact, it seems that among the more theoretically inclined computer scientists, these two subfields are seen to comprise the whole of software engineering. The remaining subfields are in need of defense; accordingly, the rest of this article is essentially an explanation of why they are just as legitimate as their formal cousins.

3 Software Engineering Research

All research is supposed to be directed at solving problems, and software engineering research is no exception. The problem addressed by software engineering research is that quality software is very difficult to produce, and it is even more difficult to produce consistently enough to meet the growing demand for it. Section 5, dealing with why software engineering research is necessary, details exactly how difficult it is to produce quality software and discusses the properties of software that cause the difficulties. This present section deals with the content of the research, the paradigms for carrying it out, and the responsibilities of the researchers in these steps.

3.1 What is Software Engineering Research?

Software engineering research is concerned with the problems of systematic production of quality software. There are many skilled programmers, call them artists if you will, who produce such software more often than not. However, the demand for software far outstrips the productivity of these people. What is needed is that all professional programmers produce such software consistently. The research of software engineering focuses on finding ways by which the production process can be improved to the point of routine repeatability. It is a multipronged attack on all parts of the problem, with the hope that some of the prongs will yield an advance of some kind in the process.

3.2 Research Approaches

Among the prongs of the attack is research on the following aspects of software engineering.

1. Underlying theory
2. Software tools
3. Integrated software development environments
4. Software development methods
5. Management techniques

Each of these research approaches represents a different way to make the production of quality software more systematic and reproducible.

1. Research on underlying theory attempts to find more formal, and thus more sys-

tematic, ways of producing software. Errors are avoided by proving that the software at hand does what it is claimed to do.

2. Research on software tools attempts to find tools that can be applied to reduce the drudgery of software development, freeing the programmer for more creative thinking and reducing chances for errors. These tools generally do clerical tasks that are error prone because in the sheer size of real-life software, it is easy to overlook details that the tools can easily find.
3. Research on integrated software development environments attempts to combine such tools in a seamless fashion so that a whole collection of tools can be used over the entire life cycle. The environment remembers to do things that a programmer in his or her zeal to move on to the next step might overlook.
4. Research on software development methods attempts to find procedures and paradigms that programmers can follow to make software production more systematic and less error prone. They also help by suggesting steps to take when the programmer is stumped and has no idea of how to proceed.
5. Research on management techniques attempts to eliminate human interaction problems as an impediment to the production of quality software and, in fact, to marshall the power of groups of programmers to achieve what an individual cannot hope to achieve.

Occasionally, a researcher gets an idea for a new approach that may seem ridiculous to others, who discount it. However, in any area of active research, it is dangerous to discount any approach that has a reason that it might work and that has not been specifically disproved. No one can know ahead of time whether or not the approach will work. If we discount an approach that would have worked, we are cutting ourselves off from a benefit. Ultimately, we cannot discount anything that might yield a solution, anything reasonable that has not been demonstrated *not* to have *any* impact on the process or the software.

To use an extreme example, if it can be demonstrated that serving milk and cookies to programmers every morning before they begin working has a significant positive impact on the quality of the software produced by the programmers, then this discovery should be regarded as an important finding in software engineering. While the idea is admittedly far-fetched, the keywords here are “demonstrated” and “significant.” While strange ideas should not be discounted without evidence of their uselessness, it is equally clear that ordinary ideas cannot be counted without evidence of their usefulness.

3.3 Responsibilities of Researchers

When any approach is suggested by researchers, those researchers must, as part of their job, assess the effectiveness of the ideas and then determine if that assessment yields a statistically significant demonstration of the effectiveness of the approach. This evaluation is necessary, regardless of the nature of the approach, be it foundational theory, a tool, an environment, a method, or a technique. Even when the approach is theoretical and the theory can be proved sound, the researcher must demonstrate the relevance and usefulness of the theory and the effectiveness of its application to the production of reliable software.

Too often, software engineering researchers propose an idea, apply it to a few, toy-sized examples, and then grandiosely jump across a gap as wide as the Grand Canyon to an unwarranted conclusion that the idea will work in every case and that it may even be the best approach to the problem it purports to solve. Some of these researchers then embark on a new career to evangelically market the approach with the fervor of a religious leader beckoning all who listen to be believers. (See Section #snake.oil#.) While historically, methodologists have the biggest reputations for this behavior, they certainly have no monopoly on it. Theoreticians are equally guilty of proposing a formal method of software development, showing that it works for a small, previously solved, formally definable problem, and then expecting the rest of the world to see that it will work for all problems. When the rest of the world does not see how to apply it to large, real-world problems and wonders why the theoretician does not go ahead and apply it to such problems, the theoretician throws his or her hands up in the air in fit of despair over the sloppy way programmers work.

3.4 How to Assess an Approach

No matter the source and the formality of an idea, it is incumbent upon its believers to evaluate it. The builder of a theory, a tool, an environment, a method, or a management technique is obliged to:

1. describe it,
2. implement it completely, if it is a tool or an environment,
3. apply it to large software development projects under controlled experiments, and
4. soberly and critically evaluate its effectiveness, being prepared to admit that it does not really work.

Note that the failure of the idea to work, while disappointing, is an important result. If the idea was reasonable, others are likely to think of it; reporting the failure saves them from wasting effort and frees them to try other ideas.

Often the design of the experiment and the evaluation of the results is the hardest part of the research. For this reason, these are often not done properly. Many times, a statistically meaningful controlled experiment is infeasible. It may be too expensive to commit enough teams to develop the same software under different conditions. Even if some teams can be committed, the number may not yield statistically significant results. Even if a large number of teams can be committed, they still may not yield statistically significant results, perhaps because other unforeseen but independent factors dominate. In such cases, smaller experiments with careful, critical-to-a-fault introspection or interviewing must be substituted. Of course, then the results do not carry as much credibility.

When proposing a software tool or environment as a solution to one or more software development problems, it is obviously insufficient to simply propose the tool or environment and to proclaim that it solves the problems. No matter how persuasive the proposal or how convincing the logical arguments as to why the tool or environment will work, the tool or environment must be built before the following questions can be answered.

1. Can the tool or environment be built? Some tool descriptions have contained non-computable parts or have appealed to technology, e.g., expert systems or artificial intelligence, that is just not quite up to the task yet.
2. If it can be built, it is really effective as a tool or environment?
3. If it is effective, up to what sized problems can the tool or environment be effectively used?

Thus the tool or environment must at least be built. Moreover, it must be built to something resembling production quality or at least to beta-test level so that the test for effectiveness will be performed under realistic conditions. This works in both directions. First, we want to give the tool or environment every chance to succeed; and if it does not have all of its capabilities, it may fail due to the lack of some critical capability that it would normally have. Second, full-strength tools and environments are known to have more performance problems than their significantly lighter weight prototypes, and a slow-responding tool or environment is often considered worse than none. It is for these reasons that I, personally, have come to have little patience with less than fully implemented tools.

Notice that the above argument does not preclude prototyping as long as the prototype is not used for more than a proof of concept or as a means to determine the requirements for the production version. In particular, the effectiveness of the tool or environment cannot be fully assessed with the prototype, and that fact must be recognized in any claimed results arising from the prototype.

The evaluation of a tool, environment, method, or technique must be made relative to one or more bases for comparison. The new way must be compared to current practice, to doing the same work manually, to using other existing tools, environments, methods, or techniques, or to combinations thereof. It is not a priori clear that use of a tool is better than carrying out the process manually because, sometimes, there is much to be gained from the thinking that a human does when doing the task manually, and this gain will be lost if the process were automated.

3.5 Physics and Software Engineering

The situation in software engineering research is not unlike the situation in physics, where there is a strong theoretical component and a strong experimental component, neither of which can function without the other. No theoretical result is accepted unless its implications are borne out by observation. Moreover, since many of the observations predicted by the theory involve differences that are smaller than observational tolerances, the theory is also needed to derive observable implications. Conversely, empirical observations are used as the basis for induction to theory. As an event happens that cannot be predicted by current theory or that runs counter to current theory, the creative physicist tries to see a pattern in order to develop new theory.

In software engineering, the theory is used to predict methods, technology, and techniques that will have a positive impact on the software development process. It is necessary to test these theories. Many times, the testing of a theory, especially that about technology, requires building entirely novel software systems. These must be built to sufficient completeness that a meaningful test can be carried out. In addition, building novel software systems and playing with them suggests new methods, technology, and techniques. For example, the field of requirements engineering is new enough that few methods, technology, and techniques have been suggested. My favorite research paradigm is to build a tool that extrapolation from observations has suggested, and then to play with the tool in a complete requirements engineering effort. As the requirements are engineered, we introspect in order to identify methods and techniques to be applied in general and in conjunction with the tool, and we often identify other tools that should be built. Thus, software engineering, like physics, needs its theoretical and experimental components.

4 Contributions

The basic criteria for judging the merit of work in any academic field is whether or not it has made an original significant contribution to the knowledge of the field.

Work in the theory of programs and programming can be judged in the normal way for theoretical contributions. The hard question is how to evaluate a contribution in the how-to subfields, formal methods, technology, methodology, management, and in the production of software artifacts. The following subsection attempts to answer this question. The subsections after that describe typical contributions of the various subfields and list specific noteworthy contributions.

4.1 Evaluating Contributions

One of the most difficult questions a software engineering academic faces is how to evaluate work results in a how-to subfield. If the result is a tool, an environment, a method—formal or not—, a technique, or anything else that is designed to improve the programming process, the natural question that is asked, and in fact that *should* be asked, is “How does one evaluate the tool, environment, method, or technique?”

Ultimately the evaluation is based on professional and expert judgement of the peers, referees, or thesis committee members in the subfield. This is no different from what happens in mathematics and formal computer science. Who is to say that the theorem proved for a thesis is a significant advance? Who is to say that the proof is done correctly? Who is to say that the proof is done well? Who is to say that the whole thing is creative? Other, expert mathematicians and formal computer scientists are the ones to say!

In the software engineering area, the peers or committee members use their expertise to answer the following questions.

1. Is the tool, environment, method, or technique a significant advance?
2. Is the tool or environment implemented correctly?
3. Is the tool or environment implemented well?
4. Is the tool or environment effective for its purpose?
5. Can the method or technique be applied under normal circumstances, and in those circumstances does it have a high probability of being successful?
6. Is the whole thing creative?

Of course, it is the researcher's responsibility to explain in publication how the tool, environment, method, or technique is to be evaluated and to carry out that evaluation in as sober, scholarly, and fair manner as possible. The researcher must identify the goal of the tool, environment, method, or technique and check it against the goal. In cases in which an objective measure is possible, e.g., for performance, the researcher must provide the measurement or complexity assessment. In cases in which the satisfaction of a goal is at best subjective, the researcher must be as fair as possible in an introspective evaluation and may even have to do some kind of experiment to see how the average programmer uses the tool, environment, or method. In fact, it is this last evaluation requirement that makes research in software engineering more difficult than doing research in mathematics. Evaluation methods must be pioneered along with the tools, environment, methods, and techniques.

Individual works in the artifacts subfield are specific software artifacts. The evaluation of an artifact is similar to that of a tool, environment, method, or technique. It is done by expert judgment, and focuses on whether or not the artifact is a contribution both to the application area and to software engineering. It is a contribution to the application area if it solves a hard problem that has not successfully been solved before and it does so in a creative manner. It is a contribution to software engineering if the particular tool, method, or technique provides the leverage to solve the problem, leverage without which the problem would not have been solved as well or at all. It is a contribution also if its production were done in a way that helps to regularize the production of similar or related artifacts. Note then that from this point on, production of a similar or related artifact ceases to be a major contribution. When the artifact is a new software engineering tool or environment, the tool or environment must be evaluated against its purpose.

Artifacts are often used as research vehicles in areas in which the problem is to automate something not automated before or to build an active model of a real-life phenomenon. Examples of the former are in the areas of computer graphics, computer music, and electronic publishing. Examples of the latter are in the areas of artificial intelligence and simulation. In artificial intelligence, it is usual to build a program that mimics some aspect of human behavior in order to study that behavior. Such artifacts are usually bigger contributions in their application areas than in software engineering, as nothing particularly noteworthy was done to make the software development significantly easier.

In many of these cases, the problem itself is initially so poorly understood that a major part of the software development involves defining the problem. It may even be that the purpose of writing the program is to arrive at a suitable definition of the problem. Many programs in artificial intelligence are of this nature. The development of the program is equivalent in effort to developing a new theory from the ground up. Other times, a program is but the carrier of a particular software engineering technology idea, i.e., the program is a tool that implements a particular technology that is claimed to be effective. The claim cannot be tested unless the tool is built. Furthermore, building the tool makes the technology available, and that availability is a significant contribution. The make program is of this nature.

4.2 Typical Contributions of Subfields

Having explained how to evaluate contributions, this section lists typical good contributions for each subfield.

1. Theory of programs and programming: a new theory of program semantics; a new set of axioms for program verification; a new model of concurrency; a new major consistency or completeness theorem; or a new axiom that makes a new class of programs verifiable.
2. Formal methods: a new class of programs subjectable to some formal method; a new formal method based on some theory; or making a formal method significantly accessible to nonmathematicians.
3. Technology: a new tool that solves a class of problems not solvable before; a new tool that automates an error-prone manual task that was thought to require intelligence; controlled experimental demonstration of the effectiveness of a tool; integration of diverse tools into a software development environment in a manner significantly better, more seamless, or more complete than before; new paradigm for making tools; or a tool-building tool.
4. Methodology: a new method that solves a class of problems not solvable before; a new method that systematizes an error-prone, nonautomatable, manual task that requires intelligence; or controlled experimental demonstration of the effectiveness of a method.

5. Management: a new technique that demonstrably brings a class of heretofore undoable systems into the doable range; a new technique that demonstrably overcomes a human or group barrier to software productivity; or controlled experimental demonstration of effectiveness of a technique.
6. Production of software artifacts: solving with software a previously unsolved but hard problem; in particular a problem that has defied solution; or production of a new program to solve a problem whose previous software solutions were poor and in which the discovery and implementation of the current solution involved direct applications of software engineering or served as an effectiveness demonstration of a software engineering technology, method, or technique.

4.3 Specific Contributions

Section 5.3 of the report lists what I consider significant contributions in software engineering over the years. (I compiled this list with input from others, all of whom are listed in the acknowledgements. Since the ultimate decision was mine, I take all blame for the inevitable omissions and apologize for them.) The current section only summarizes this full list, and the reader is encouraged to consult that list.

In this summary, in the interest of keeping the size of the bibliography to the suggested maximum of 25 citations and of scholarly fairness, the only citations given are to those documents that readers of the *Communications* are not likely to have enccountered. Thus, well-known computer science concepts are mentioned with no citation. The reader interested in seeing the citations and the full bibliography should consult the report.

A number of major contributions to software engineering were made long before the term was coined and, in fact, long before computer science existed as a field. These include the development and use of compilers, linkers, libraries of already compiled subroutines, and symbolic debuggers to simplify the basic programming and debugging tasks plus the ability to write device-independent programs that read from and write to files specified at invocation time. I personally do not know who invented these ideas, but whoever they are, my hat is off to them.

There is an additional, attributable contribution predating computer science that has proved to be a basic observation underlying most of the complexity management in software engineering, the methods, tools, environments, and approaches. Miller observed that human mind is capable of handling at most seven, plus or minus two, distinct items of information at any time [Miller56]. We are able to handle more only by aggregating several items under a single abstraction that summarizes them or abstracts away their differences.

There are a number of more recent contributions that were made not specifically for software engineering purposes but, nevertheless, have had a major impact on the way of programming. These include the programming language C; the UNIX system in general and in particular, its portability, the shell, and pipes; the portable C compiler;

7. on-line full-screen editors; and windowing systems.

The report lists specific contributions that were intended for software engineering. These contributions are divided into a number of categories:

1. Fundamental truths: These truths deal with the ideal and real software life cycle, the large program and its evolution, and the way large programming projects really work or do not work. Often, these truths were not accepted or even understood prior to publication of the document.
2. Methodology: Most of the work was in devising methods that were claimed to, and in some cases did, help programmers produce quality software more consistently.
3. Formalisms: Some of the work on formalism has had a significant impact in showing us how to determine the meanings of the programs we write and showing us more systematic ways to produce high-quality software.
4. Tools and environments: The second largest group of contributions is in technology, tools, and whole environments of tools that help the programmer to program more effectively.
5. Testing: Contributions in program testing have been much harder to come by and have focused on finding algorithms for generating covering test data, or failing that, of testing the coverage of proposed test data.
6. Sociology and management: Probably the least accepted work in software engineering is the work in the sociology and psychology of programmers and the management of groups of people and projects. It had been long understood that social, psychological, and management issues were important in software development. It was always hard to quantify their effect and to put one’s finger on the exact effects. Also, perhaps, these issues were looked upon as diluting the field with nontechnical issues. Nevertheless, there were a few seminal books and papers that got these issues out on the table and set the standard for the future.
7. Metrics and economics: A little more accepted than the sociological and management work has been the work in software metrics and software engineering economics. It has focused on finding measurable characteristics of programs on which predictions of needed resources can be made.
8. Experiments: An important kind of research has been experiments aimed at showing that the methods, techniques, approaches, and ideas presented above do, in fact, work.

5 Why Software Engineering Research Is Needed

Software engineering research is necessary because software production is hard, much harder than many people seem to appreciate. Some generalize from the kinds of programs developed for completely specified classroom assignments, which cannot take more than a semester to complete and which are never run after they are handed in, to the belief that all software is straightforward, is only few dozen pages long, and is just a matter of implementing the obvious, complete requirements of a single-person customer. Some generalize from the kinds of programs that are formally specifiable and whose compliance to these specifications is formally verifiable to the belief that all software systems are formally specifiable and verifiable. However, the fact is that real software developed to solve real problems is several orders of magnitude more difficult than the above toy problems. First, a real problem is generally not well enough understood to specify completely, let alone to specify formally. Such programs are systems for controlling real-life processes for which the number of variables affecting the system exceed the manageable and for which the full set of variables cannot even be known. Indeed, disasters in software-based systems are generally caused by unforeseen events or by highly unlikely combinations of individually foreseen events for which the program has no pre-planned response [Neumann86]. When these failures are analyzed, it is often found that the root causes are not even mentioned in the system specification.

This section attempts to show just how hard software development is—in effect to show that it is absolutely necessary for software engineering research to address making software production more systematic and repeatable.

5.1 Programming Is Hard

> “Software engineering ... is the part of computer science that is too difficult
> for the computer scientists.”
>
> — F.L. Bauer [Bauer71]

5.1.1 The Experience of Knuth in Writing TEX

At the 1989 conference of the International Federation of Information Processing (IFIP) in San Francisco, Donald E. Knuth was invited to give a keynote address [Knuth89, Knuth91]. Knuth is well known for his work in programming languages and algorithms. His most famous programming language papers are “The Remaining Trouble Spots in ALGOL 60” [Knuth67], “Semantics of Context-Free Languages” [Knuth68], and “Structured Programming with goto Statements” [Knuth74b]. The second of these papers has spawned a major area in formal semantics of languages, attribute grammars. The algorithms presented in this paper have found their way into the work on term-rewriting systems. Knuth’s landmark work in algorithms is the, to date, three-volume encyclopedia on algorithms [Knuth69, Knuth71, Knuth73], which is targeted to be seven volumes. Each volume has proved to be the definitive book in its area, clearly elucidating all issues of each algorithm presented, the correctness, the complexity and performance, and the pragmatics. Many of the exercises in these books turn out to be major problems in computer science, prompting many a doctoral candidate to tackle them as PhD topics. Knuth received the prestigious ACM Turing Award in 1974 for all his work prior to that date [Knuth74a]. He has even ventured into the esoteric topic of axiomatization of number systems [Knuth74c]. If there was ever anyone with academic legitimacy, it is he.

Yet over the past decade or so, his main work has been in the development of two major programs for use in document typesetting. What started out as an attempt to make sure that all his subsequent books, to be printed with computer-driven typesetters, would look as good as his earlier hand-typeset books, eventually mushroomed into a ten-year effort yielding, to date, two releases each of TEX and METAFONT. The former is a program for typesetting mathematical and technical documents and the latter is a program for producing fonts to be used by TEX and other typesetting programs.

The paper accompanying the IFIP keynote address (and presumably the address too) explains that one of the lessons learned from the development of this typesetting system is that “software is hard.”

> What were the lessons I learned from so many years of intensive work on the practical problem of setting type by computer? One of the most important lessons, perhaps, is the fact that SOFTWARE IS HARD.... From now on I shall have significantly greater respect for every successful software tool that I encounter. During the past decade I was surprised to learn that the writing of programs for TEX and for METAFONT proved to be much more difficult than all the other things I had done (like proving theorems or writing books). The creation of good software demands a significantly higher standard of accuracy than those other things do, and it requires a longer attention span than other intellectual tasks.

Knuth’s remark that writing software is more difficult than proving theorems and that significantly more accuracy is required for writing software than for proving theorems merits closer examination. The point is the difference in audience. The programmer is writing for an incredibly stupid and literal audience that has no understanding to guide it through minor incompleteness. The mathematician is writing proofs for a human audience, and a highly professional one at that. The readers of the theorem can be counted on to fill in unimportant details that the author has left out. The author can count on the reader’s abstraction ability to supply missing details. No computer can fill in unimportant and missing detail. Past attempts to get machines to infer what the user means have failed except in very limited domains. Moreover, many published theorems contain, plainly and simply, mistakes. When these are minor, such as a typographical error or a proof step misstated, the competent human reader can be expected to correct the problem while reading the proof. No computer is that forgiving, as all programmers can attest. A computer does what you told it to do, and *not* what you thought you told it to do. Interestingly, Knuth himself, has had to publish two consecutive corrections to the proof of a theorem about attribute grammars presented in [Knuth68].

5.1.2 Lehman on the Nature of Programming

Meir Lehman, in his recent “Software Engineering, the Software Process and Their Support” [Lehman91], captures the essence of hard software problems. He classifies programs according to the S-P-E scheme he devised with Les Belady.

> An *S-type* program is one required to satisfy some *pre-stated specification*. This specification is the sole, complete and definitive determinant of program properties.... In their context *correctness* has an absolute meaning. It is a specific relationship between specification and program.
>
> A *P-type* program is one required to produce an acceptable *solution* of some ... problem. If a complete and unambiguous statement of that problem has been provided it may serve as the basis for a formal specification.... Nevertheless, program correctness relative to that specification is, at best, a means to the *real end*, achievement of a satisfactory solution.... In any event, when *acceptability* of the solution rather than a relationship of correctness relative to a specification is of concern the program is classed as of type P....
>
> An *E-type program* is one required to solve a problem or implement an application in some *real world* domain. All consequences of execution, as, for example, the information conveyed to human observers and the behaviour induced in attached or controlled artifacts, together determine the acceptability of the program, the value and level of satisfaction it yields. Note that *correctness* cannot be included amongst the criteria.... Moreover, once installed, the system and its software become part of the application domain. Together, they comprise a feedback system that cannot, in general, be precisely or completely known or represented. *It is the detailed behavior under operational conditions that is of concern.*

Generally, the kinds of programs with which the theoretical branches of computer science deal are S type. These are programs for which assertions about their behavior can be verified formally. Almost all of theoretical computer science assumes that all programs are S type. Much of programming language research is dedicated to making languages so perspicuous that more and more programs can be S type.

However, both forefront industry and software engineering research deal mainly with E-type programs and the process of their development. Most new software developed at the frontiers of operating systems, distributed systems, concurrent systems, expert systems, intelligent systems, embedded systems, avionics systems, etc., is E type. For example, consider the fly-by-wire software to fly aircraft that are, for the benefit of fuel efficiency, so unstable that they cannot be flown solely by a human pilot. The software has become an integral, inseparable part of the aircraft in that the plane itself ceases to function if the software does not function. The versions of these E-type systems under development currently contain significant portions that have never been developed before. For these portions there is little or no existing software to use as the basis of a formal model. The reason that software engineering research deals mainly with the problems of the production of E-type programs is that the less difficult types of programs are not as much in need of methodological assistance.

Software engineering research, when it develops tools and environments, is developing mainly E-type software. Initially, the software engineers, doing their own software development, perceive a need for a tool or an environment to do some of the more clerical parts of their task, to manage the software through its entire life cycle. From this perceived need comes a vague idea of the functionality of the tool or environment. However, this idea cannot be made less vague until the tool or environment is actually used. Thus, the software engineer builds a near-production quality prototype and tries it out, perhaps in the construction of the next version. Thus, the tool or environment has become an inextricable part of its own and other software life cycles.

It is these impossible-to-specify and thus impossible-to-verify programs that are the subject of software artifact engineering research, and it is the regularization of the process of producing these artifacts that is the subject of software engineering methodology research.

Perhaps, here we see the basis for the theoretician's condemnation of software engineering research. The kinds of programs they work with are S type. Perhaps they are not even aware of the existence of P-type and E-type programs, and they believe that all programs are S type or are easily made S type. If all one knows about are S-type programs, then it is quite reasonable to doubt the necessity of methodological research; it suffices to formalize the problem and the program rolls right out of the formalization. Certainly, S-type programs can be implemented quite systematically every time, and there is no need for software project management, for example, to build them. It is for E-type programs that the most help is needed; and if a technology, method, or technique is judged as useful, it is because it makes the production of E-type programs more systematic and repeatable.

Lehman's article explores the reasons for this fundamental difficulty, and the reader is encouraged to read the original or the summary of this in the report.

5.1.3 Brooks on There Being "No Silver Bullet"

In "No Silver Bullet" [Brooks87], Fred Brooks explains that fashioning complex constructs is the essence of software development and explores the possibility that technology or technique can be found to make software development fundamentally easier. He builds an analogy between the terror of software complexity and that of the werewolf. Whereas the legendary werewolf could be eliminated by an equally legendary silver bullet, the software crisis has no corresponding magical, quick solution. Brooks says basically that he sees no single development, in technology or technique, that promises an order-of-magnitude improvement in programmer productivity, software reliability, and software simplicity. Moreover, the very nature of software makes the discovery or production of such a silver bullet unlikely ever. No breakthrough will do for software what has been done for hardware by electronics, transistors, and large-scale integration. The surprise is not that software progress is slow; it is that hardware progress has been so fast. In the past 30 years, not only have costs come down by 3 orders of magnitude, but speed and capacity have gone up by 6 orders of magnitude, and size has gone down by 6 orders of magnitude. In no other technology has the improvement been so large and so quick.

The difficulties of software can be divided into two groups,

1. essence, those which are inherent in the nature of software and
2. accidents, those which arise from the production of software.

The essence of software is that a program is a network of interlocking data structures, relations among these data, algorithms, and procedure invocations. This network exists no matter what specific representations are used, simply because they exist in the abstractions that the software represents. As suggested by a diagram nearly identical to that of Figure 5-1 on page 24, this network increases exponentially in complexity as the size of the problem grows. It is unmanageable except for the smallest, toy programs.

All software is an attempt to implement some abstraction, a collection of capabilities desired by the client. Brooks believes that the hard part of building software, i.e., the essence, is the specification, design, and testing of the abstraction and not the labor of constructing the program and testing its agreement to the abstraction, which is only the accident. He considers syntax errors, and perhaps even logical errors in the code, in getting the representation right to be fuzz compared to conceptual errors in getting the abstraction right. Technology and methodology have focused on the process of constructing and testing the program, i.e., the accident. If Brooks's belief is true, then software will always be hard to build and there will *never* be a silver bullet.

The properties of the essence that makes software difficult are:

1. **Complexity**: Software is more complex than any other entity constructed by humans "because no two parts are alike (at least above the statement level)" [Brooks87].
2. **Conformity**: Whenever a software-based system is built, if anything needs to be bent to get the hardware, software, firmware, and peopleware to conform with each other, the software is the one chosen to be bent.
3. **Changeability**: Software is subject to change far more than other technology; once delivered, computer hardware, buildings, and vehicles undergo changes only rarely.
4. **Invisibility**: Brooks observes that "Software is invisible and unvisualizable."

The accidents of software are the difficulties in the production at one particular point in time of one particular representation of a desired software abstraction, that is, the current difficulty of programming it for one particular platform in one particular programming language. Surely, the production of cleaner hardware or higher level or cleaner programming languages makes the development of a program easier. Certainly the development of tools that help manage some of the complexity of developing software in a particular language and/or on a particular platform makes the development of a program easier. However, the best program in the world is of no use if the essential purpose of the program, the abstraction it is to implement, is not understood. Thus, taming the accidents of software do not help much in taming the essence of software.

The fact is that nearly all the advances mentioned in Section 4.3 and Section 5.3 of the report do no more than tame some accidents of software. Certainly, the more concrete of these, the tools or the algorithms for generating test cases, merely attack one particular accident. Only the more amorphous of the contributions, such as Parnas's method for modular decomposition, begin to attack the essence.

Basically, there cannot be as dramatic an improvement in software simply because the raw material that makes it, the human brain, cannot be improved by the orders of magnitude that are required. We humans have basically the same brains that we had 5000 years ago. The brain is not really any smarter; it is just more experienced and it builds on more history and technology these days than it did 5000 years ago. It has learned to tame some of the accidents but cannot yet really cope with the essence.

5.1.4 The Formal Difficulty of Programming

As another assessment of the difficulty of programming, it is useful to understand the formal, computational complexity of programming. Writing a correct program that meets a given specification is computationally at least as difficult and formally as unsolvable as proving a theorem.

Back in the early '70s, work by Manna and Waldinger [Manna71] showed that writing a correct program to satisfy a given input-output predicate can be reduced to constructively proving the existence of a solution to the problem represented by the specification. The statements in the program can be extracted from the steps of the proof. In addition, various researchers have developed prototype automatic programming systems that successively refine formal specifications of a desired program into a program meeting the specifications [Partsch83, Elspas72]. All these systems use at least a rudimentary verifier. The more powerful the theorem prover, the more powerful the refiner, the more programs that can be generated. Thus, writing a program is no harder than proving a theorem. As a matter of fact, it appears that no nontoy program has ever been generated with such a system. Certainly, I have never seen a report suggesting otherwise.

It is well known that the existence of a program to satisfy a given specification is undecidable. It is also well known that *whether* a given program satisfies a given specification is also undecidable. Therefore, it cannot be algorithmic to generate a program to satisfy a given specification. The existence of a program that satisfies a given specification can be demonstrated only by a special-case proof. That a program satisfies a specification can be demonstrated only by a special-case proof, and a program to satisfy a specification can be generated only by a special-case process. All these special-case proofs and processes are *not* algorithmic. Therefore, proving a theorem is no harder than writing a program that meets a given specification. Thus, proving theorems and writing programs are tasks of equal formal difficulty.

Yet, there is some sense in which program writing is easier than theorem proving. After all, many more people seem to learn how to program than learn how to prove theorems. Perhaps this phenomenon comes from the fact that if a program is wrong, you tend to learn that pretty quickly as the program is being run. Proofs generally require the presence of a competent mathematician to spot errors; at least, it is more common for the author of a program or proof not to see errors than it is for a computer running the program with test data not to show errors. Consider the oft-heard lament of programmers, "... but there's no way that bug can be happening!" indicating that the mental model of the program does not match reality. This more complete feedback with programming should not be taken as a sign that programming is inherently any easier than proving theorems; it is merely proof that humans learn better when the reinforcement, positive or negative, is swift and consistent.

5.1.5 Real Programs and Mathematical Theory

Many of the problems that software engineers solve are more difficult than anything mathematicians will deal with. A mathematician will not deal with a problem *unless* the problem itself can be formally stated. Software engineers routinely write software for systems that are not well enough understood to specify, let alone formalize. Typically, such software developments follow an iterative life cycle in which the evolving specification and software feed back on each other. The result is a more complete understanding of the problem, a statement of the requirements, and at least one formal model of the system, namely the software. This sort of software development is equivalent to the development of a new theory about the system from the ground up.

Just because we do not understand software or the problems we solve with software well enough to prove theorems about them does not mean that they are not worth studying; quite the contrary—software presents a much harder, broader, deeper problem. Certainly, theory is essential in attacking many software problems. However, it appears to many practicing software engineers that some theoretical software engineers are taking baby steps by working on problems whose domains are well enough understood to solve by formal methods. Practicing software engineers just do not buy the claim of the formalists that the methods scale up to solve the problems that the practicing software engineer encounters on a daily basis. Moreover, it appears that no formalist has successfully taken up the challenge of producing an industrial-strength, full-scale operating system, process controller, editor, etc. with formal methods; moreover, these are not the really hard programs because they are well understood.

Many problems addressed by software engineering are considered by many to be *wicked* *problems* [Rittel72]. A wicked problem is one with the following characteristics.

1. The problem's definition and solution must be carried out concurrently.
2. There is no unique definition or unique solution for the problem.
3. There is always room for improvement in any problem definition and solution.
4. The problem is complex because it is composed of many interrelated subproblems.
5. The problem has not been solved before and is unlike any other that has been solved before. It thus requires new approaches, and the resulting solution is not likely to be applicable elsewhere.
6. Many parties with differing priorities, values, and goals have a stake in the problem and its definition and solution.

Wicked problems defy formalization; furthermore, whenever a formalization is available, it can always be improved. It is unlikely that the formalization can build on the existing body of theory, and thus the formalization is built from the ground up for each new problem. These properties make the work harder than that usually performed by mathematicians.

Hence, in many senses, industrial-strength programming is harder than proving theorems, mainly because it often deals with problems that have yet to be formalized.

5.1.6 Classroom Exercises and Real Programs

A false sense of easiness is conveyed in introductory and intermediate programming classes. The programming problems solved in these classes are necessarily small, as no assignment can last more than the semester or quarter, and most of the assignments can be finished in two or three weeks. These problems are trivial compared to industrial-strength problems that take groups of programmers several years to produce and for which person-year is the unit of work. The point is that the work involved is a function of complexity, and complexity is a function of the interaction between parts of the program. Given this fact, the complexity of software grows exponentially with its size. Figure 5-1 shows the number of possible interaction paths between parts as the number of parts grows from one to five. Compounding the problem is the fact that as the number of people required for a project grows, the volume of communication necessary to keep people abreast grows exponentially for precisely the same reason. Thus, normal linear extrapolations of difficulties experienced in a classroom programming assignment to those that will be experienced in industry just do not work. The evidence that people extrapolate only linearly comes from the fact that people consistently underestimate the effort involved for a new bigger project.

In addition, classroom exercises are woefully unrealistic in terms of the quality assurance and maintenance activities they require. Class programs are tested only enough to make sure that they will pass the grading test. They are forgotten once they are handed in, never to be maintained. It is well known that testing and maintenance account for about 60% of the cost of program production [Boehm81]. These two activities are difficult precisely because they involve the paths of interaction between parts. When tracking down the source of a bug, which usually shows up nowhere near the source, all possible paths of interaction must be followed backward from where the bug is observed. Moreover, each time a change is made, all possible impacts of that change must be explored. Because these interactions grow on an exponential scale, human creativity becomes an absolute necessity to cut through the combinatorial explosion to focus on the most likely places of interaction. Thus, the classroom programming exercises simply do not show the full scale of intellectual difficulty involved in software production. Those who generalize from software developed in the classroom come to unrealistic conclusions.

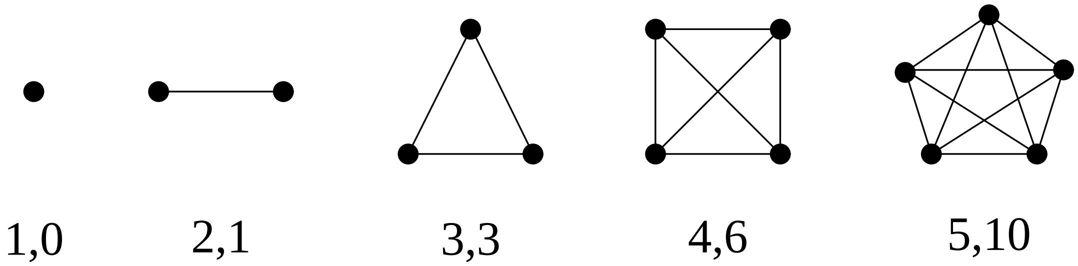

Figure 5-1: Numbers of parts and of possible interactions between them

The funny thing is that even these classroom-style exercises are harder than people think. There are several cases of authors promoting a certain systematic or formal way of working in a published paper containing a smallish, classroom-style toy example, only to end up red-faced as readers found and published corrections to their supposedly correct example.

Peter Naur published a paper describing what would be called *proof-assisted structured programming* [Naur69]. To demonstrate the method, he gave an informal specification of a small formatting program and then constructed a program implementing the specifications while informally verifying its correctness. The informal specification consists of four natural language sentences, three of which behave as axioms, and is not unlike a specification given to a class for a programming assignment. The program has about 25 lines of Algol. It was published proved but not tested. Burt Leavenworth reviewed the paper for *Computing Reviews* [Leavenworth70] and found a fairly trivial boundary condition fault that would have been spotted easily in a test. Still later, Ralph London found three other faults [London71], again boundary condition faults that would easily have been found in tests. London offered a new program for the same specification. He *formally* proved his program correct and dared to publish it without testing it! (I guess that he thought that the object lesson of Naur's experience was that the proof must be done formally!) As might be expected, John Goodenough and Sue Gerhart found still three more faults [Goodenough75], which London had missed. These, too, would have been detected had London tested his program.

There was another comedy of errors going on at the same time concerning the specifications themselves. Of the 7 faults found after initial publication, 2 can be considered specification faults, that is, situations not even mentioned in the specifications. Accordingly, Goodenough and Gerhart produced new set of specifications about 4 times longer than Naur's, still in natural language. Still later, Bertrand Meyer detected 12 faults in Goodenough and Gerhart's specifications and attributed them to the use of ambiguity-laden natural language [Meyer85]. He gave a formal specification using mathematical notation that corrects these problems. Recognizing that these formal specifications are hard to read, Meyer also gave a natural language paraphrase of the formal specifications. As the reader might now expect, this was not the end of the story. Steve Schach, reporting on the above history to stress the importance of testing and the difficulty of getting the specifications right, reports an additional fault, an ambiguity, in Meyer's natural language specification [Schach90].

Certainly each of the authors in this history has an object lesson in his or her paper, and each, except perhaps the last has been embarrassed to have been made the object of a lesson. At the risk of subjecting myself to a future object lesson, I now add another lesson. My lesson is that even classroom-style programs are extremely difficult if not impossible to get right, even when lots of good people are involved, and even when there is all the time in the world to do it (we all know how publication takes *forever*). If classroom-style, relatively trivial exercises are so difficult to do right, what hope is there for any real-life, industrial-strength or at-the-frontier program to be done right? Programs are complex animals, and the study of methods to manage that complexity is an intellectual challenge even greater than that of programming and mathematics, which are only tools of the process.

5.2 Necessity of Nontechnical Solutions

It is a sad fact that purely technological solutions have not solved the problem of producing quality E-type software. The software crisis continues [BUGS90, Kitfield89], and it is recognized that technology is neither the problem nor the solution [ASB90]. It is well known among those conducting controlled experiments into the effectiveness of programming tools, methods, and techniques that it is hard to obtain statistically significant results because individual differences among the programmers and groups dominate [Boehm84]. Differences of 28 to 1 in programmer and group productivity have been found [Sackman68, Boehm81]. Thus, it is critical to consider nontechnical issues such as human intelligence and creativity, individual and group behavior, management, psychological, and sociological issues [Weinberg71].

I once gave a talk entitled “Software Engineering Myths” to a group of programmers at a company in Israel. After the talk, I stayed for about an hour to answer questions. Not one question was technical, even though I had been told to expect questions about software tools. All the questions were about how to get upper management to allow them, the programmers, to apply the methods and tools that they already knew about and not to lock them into unreasonable straitjackets. It seems that their operating environment and local politics, along with unreasonable expectations and deadlines were getting in the way of their effective functioning. This is clear testimony to the importance of nontechnical issues in software engineering.

So what should be done about the fact that the nontechnical issues introduce a degree of fuzziness that is uncommon in computer science and other technology-based fields? My advice is to accept it as a challenge. It is a challenge to devise methods to state these issues as requirements because one must be able to measure compliance to requirements. It is a challenge to devise methods to test whether these nontechnical approaches work.

Indeed, experimental methods developed and perfected in psychology, sociology, and management will have to be borrowed, refined, and used. Exploration of the effectiveness of these nontechnical approaches may even be acceptable PhD topics, *if* the thesis addresses the issue of assessment and carries out the assessment in a manner that leaves even the technically expert, skeptical reader convinced of the effectiveness of the approaches.

5.3 Classical Engineering and Software Engineering

Steve Schach observes that bridges collapse less frequently than do operating systems [Schach90]. Given that bridges are built by civil engineers and operating systems are built by software engineers, and presumably both kinds of engineers practice engineering, Schach then asks, “Why then cannot bridge-building techniques be used to build operating systems?” In effect, he is asking why civil and software engineering are different. The answer, as Schach puts it, is “bridges are as different from operating systems as ravens are from writing desks.”

These are the properties of a bridge collapsing.

1. The damage is major, unrepairable, and usually life threatening.
2. The collapse is an indication that the original design or construction was faulty, because the bridge as designed and constructed did not withstand at least one of the conditions to which it was exposed.
3. Very little of the original bridge itself is reusable, so, it will be necessary to design and build a new bridge from scratch.

These are the properties of an operating system collapsing.

1. The damage is minor, repairable, and usually not life threatening.
2. The collapse is an indication that the original design or construction was faulty,

because the system as designed and constructed did not withstand at least one of the conditions to which it was exposed.

3. Usually the system can be rebooted, and it suffices to do so. If the problem is transient, it may not happen again; and in any case, usually there is nothing else that can be done because the source of the fault was not identified. If at some point the source of the fault is identified, an attempt is made to fix that fault by local modifications of the code, reusing almost all of the previous code. Almost never is the whole system thrown out and rebuilt from scratch.

Even though item 2 in both lists above are essentially identical, the differences between physical and thought media and their malleability make all the differences in the world between the items 1 and 3.

Another essential difference between bridges and operating systems is their different concepts of fault anticipation and fault tolerance. It is normal to over-engineer a bridge so that it can withstand every anticipatable condition such as destructive weather, flooding, and excessive traffic. However, beyond the agreed upon upper bound of the force that the bridge is subjected to, there is no attempt to keep the bridge from collapsing and to allow it to collapse gracefully; we just make sure that that upper bound is far beyond what it will ever be subjected to. On the other hand, it is accepted that there is no hope of anticipating all possible conditions to which an operating system can be subjected. Instead, we try to design operating systems so that if they fail, they fail in a way in which the damage is minimal and is easily recovered from.

Why can we not anticipate all possible conditions to which the operating system will be subjected? We cannot because, as noted in several preceding sections, the set of conditions to which an operating system can be subjected is constantly growing as we get more ambitious about what we automate. We can anticipate the conditions to which a bridge is exposed because the environment that can affect a bridge is not changing.

The final essential difference between bridges and operating systems is the way maintenance is approached. Bridges deteriorate, and maintenance is restricted to repairing it in a way totally consistent with its original design to restore it as closely as possible to its original state. No one would consider moving a bridge to another location (the Arizona tourist attraction London Bridge notwithstanding). Operating systems do not deteriorate. Existing flaws are corrected and new features are added, the result being that the original design is modified as the system is moved away from its original state. Moreover, one thinks nothing of moving an operating system to a new machine even if it has a different architecture and instruction set.

Thus software engineering, which works with ideas, is different from more classical kinds of engineering, which work with physical substances and objects. Those differences are what make software so complex and software engineering so intellectually challenging.

6 Academic Discipline of Software Engineering

An academic discipline requires a body of knowledge, a continual supply of important hard problems to solve, and research to solve these problems. The performance of this research is a major part of the academician's job and constitutes an on-the-job initiation rite for the PhD candidate in the discipline. The other major part of the academician's job is to teach the body of knowledge, both that which is established and that which is being discovered by the research.

The body of knowledge of software engineering is, in effect, the solutions to problems that are believed to have been solved. The problems are those of the production of quality software, and the research is the discovery of approaches that systematize the production of quality software.

7 Publications

In academia, publications are critical to the peer evaluation and academic promotion process. They are take as *prima facie* evidence of significant contributions, as their purpose is to describe the results that are claimed to be significant contributions. I have sat on enough promotion case decisions to know that "publish or perish" is no joke and that most of the deliberation in these decisions concerns the candidate's publication record.

I have heard comparisons of published papers in conferences and journals of theoretical computer science and published papers in conferences and journals of software engineering. The claim is made that the theoretical papers involve much more work to bring to final published form than do the software engineering papers. This can be substantiated partially by the longer delays between submission and appearance for the theoretical papers. In addition, it takes much more work to get the first submitted version written. This observation may be true, but it misses part of the point. The way theoreticians work, the paper is the *whole* work. When an idea comes to the theoretician, he or she begins writing a paper. The development of the theory is the writing of the paper. On the other hand, even before a paper in software engineering can be written about a particular tool, environment, or software artifact, the tool, environment, artifact must be implemented, installed, tested, and used. Even before a paper can be written about experiences using a software method, management technique, tool, environment, or program, the tool, environment, or program must be implemented, installed, and tested; the users must be trained in the method, technique, tool, environment, or program; they must be left to apply the method, tool, environment, or program; and finally the authors must decide what has been learned. If one counts the work that must be completed before writing can be started, it is doubtful that the theoretician is spending more time to produce a paper than is the software engineer. Indeed, the labor intensiveness of software engineering research is the reason that the publication list of a good software engineering academic will not be anywhere near as long as that of a good theoretician.

8 Conclusion

It has been observed that computer science is the science of complexity. Nearly everything computer scientists work on is geared more or less to reducing or managing the complexity of some system, be it hardware, software, firmware, or people. Software is the most malleable of the wares that are the subject of computer science; its very malleability is a continual enticement to attempt more and more ambitious projects that are beyond what can be done by special-purpose hardware and firmware and what can be done by people. The ambition leads to attempting more and more complex tasks for which the only hope for solution lies in reducing and managing that complexity.

Managing software complexity demands a deep understanding of software. It also demands a good understanding of hardware and firmware. Because software is created by people and groups of people, managing software complexity demands also a good understanding of people and groups, and that understanding pulls in elements of psychology, sociology, and management. Moreover, if someone claims that software engineering is no more than psychology, sociology, and management, simply ask this person if he or she would want the air traffic controller software that lands his or her next flight to have been written by a psychologist, sociologist, or manager who does not also have a deep understanding of software in particular and computer systems in general. Can you, the reader, imagine how someone without an understanding of how a tiny change to a program can cascade into dozens of seemingly unrelated bugs, of how algorithms can have different orders of complexity, and of what abstraction tools and concepts have been developed to contain complexity can possibly be relied upon to produce quality software for critical applications on which all of our lives depend?

Software engineering is intellectually deep and is a vital area of academic study. People who engage in this study should be afforded the same academic respect that is given to other, more established disciplines.

Acknowledgments

I thank Rich Adrion, Mark Ardis, Vic Basili, Len Bass, Orna Berry, Manfred Broy, Dave Bustard, Lionel Deimel, Jerry Estrin, Thelma Estrin, Stuart Feldman, Gary Ford, Nissim Francez, Peter Freeman, John Gannon, David Garlan, Carlo Ghezzi, Norm Gibbs, Leah Goldin, Harvey Hallman, Dick Hamlet, David Harel, Susan Horwitz, Gail Kaiser, Shmuel Katz, David Kay, Don Knuth, Manny Lehman, Nancy Leveson, Carlos Lucena, Luqi, Linda Northrop, David Notkin, Lee Osterweil, Lolô Penedo, Linda Pesante, Keith Pierce, Raphi Rom, Steve Schach, Richard Schwartz, Avner Schwarz, Mary Shaw, Rob Veltre, Tony Wasserman, Peter Wegner, Elaine Weyuker, and Jeannette Wing for reading and commenting on a drafts of the report on which this article is based. David Kay and Mary Shaw, in particular, provided many juicy quotes that I have shamelessly lifted for inclusion in this document. Nancy Leveson and David Notkin were so critical of an earlier draft that I had to go visit them and their students just to sort out this criticism; they were right and the report is much better for it. Nancy also stuck with me through several other revisions.

Bibliography

ASB90 Army Science Board. *Final Report: 1989 Ad Hoc Subgroup on Software in the Army*. Army Science Board, July, 1990.

Bauer71 Bauer, F.L. “Software Engineering,” pp. 530–538. *Information Processing (IFIP) '71*. Amsterdam: North Holland, 1971.

Boehm81 Boehm, B.W. *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice-Hall, 1981.

Boehm84 Boehm, B.W., Gray, T.E., and Seewaldt, T. “Prototyping vs. Specifying: A Multi-Project Experiment,” pp. 473–484. *Proceedings of the Seventh International Conference on Software Engineering*. Orlando, FL, May, 1984.

Brooks87 Brooks, F.P. Jr. “No Silver Bullet.” *Computer 20*, 4 (April, 1987): 10–19.

BUGS90 Subcommittee on Investigations and Oversight. *Bugs in the Program, Problems in Federal Government Computer Software Development and Regulation*. Subcommittee on Investigations and Oversight, April, 1990.

Elspas72 Elspas, B., Levitt, K.N., Waldinger, R.J., and Waksman, A. “An

Assessment of Techniques for Proving Program Correctness." *Computing Surveys 4*, 2 (June, 1972): 81–96.

Ford90 Ford, G. *1990 SEI Report on Undergraduate Software Engineering Education* (Technical Report, CMU/SEI-90-TR-3, DTIC: ADA223881). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, March, 1990.

Goodenough75 Goodenough, J.B. and Gerhart, S.L. "Toward a Theory of Test Data Selection." *IEEE Transactions on Software Engineering SE-1*, 2 (June, 1975): 156–173.

IEEE91 "IEEE Standard Glossary of Software Engineering Terminology," *IEEE Software Engineering Standards Collection*, Spring 1991 Edition. New York, NY: IEEE, 1991.

Kitfield89 Kitfield, J. "Is Software DoD's Achilles Heel?" *Military Forum* (July, 1989): 30 ff.

Knuth67 Knuth, D.E. "The Remaining Trouble Spots in ALGOL 60." *Communications of the ACM 10*, 10 (October, 1967): 611–618.

Knuth68 Knuth, D.E. "Semantics of Context-Free Languages." *Mathematical Systems Theory 2*, 2 (1968): 127–145.

Knuth69 Knuth, D.E. *The Art of Computer Programming: Fundamental Algorithms*. Reading, MA: Addison-Wesley, 1969.

Knuth71 Knuth, D.E. *The Art of Computer Programming: Seminumerical Algorithms*. Reading, MA: Addison-Wesley, 1971.

Knuth73 Knuth, D.E. *The Art of Computer Programming: Sorting and Searching*. Reading, MA: Addison-Wesley, 1973.

Knuth74a Knuth, D.E. "Computer Programming as an Art." *Communications of the ACM 17*, 12 (December, 1974): 667–675. 1974 ACM Turing Award Lecture.

Knuth74b Knuth, D.E. "Structured Programming with goto Statements." *Computing Surveys 6*, 4 (December, 1974): 261–302.

Knuth74c Knuth, D.E. *Surreal Numbers*. Reading, MA: Addison-Wesley, 1974.

Knuth89 Knuth, D.E. *Theory and Practice* (Report No. STAN-CS-89-1284). Palo Alto, CA: Computer Science Department, Stanford University, 1989.

Knuth91 Knuth, D.E. “Theory and Practice.” *Theoretical Computer Science 90*, 1 (1991): 1-15.

Koen85 Koen, B.V. *Definition of the Engineering Method*. Washington, DC: American Society for Engineering Education, 1985.

Leavenworth70 Leavenworth, B. “Review #19420.” *Computing Reviews 11*, (July, 1970): 396–397.

Lehman91 Lehman, M.M. “Software Engineering, the Software Process and Their Support.” *IEE Software Engineering Journal 6*, 5 (September, 1991).

London71 London, R.L. “Software Reliability through Proving Programs Correct,” pp. 125–129. *Proceedings of the IEEE International Symposium on Fault-Tolerant Computing*, March, 1971.

Manna71 Manna, Z. and Waldinger, R.J. “Toward Automatic Program Synthesis.” *Communications of the ACM 14*, 3 (May, 1971): 151–165.

Meyer85 Meyer, B. “On Formalism in Specification.” *IEEE Software 2*, 1 (January, 1985): 6–26.

Miller56 Miller, G.A. “The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information.” *The Psychological Review 63,* (March, 1956): 81–97.

Naur69 Naur, P. “Programming by Action Clusters.” *BIT 9*, 3 (1969): 250–258.

Neumann86 Neumann, P.G. “Risks to the Public.” *Software Engineering Notes* (1986). Column in nearly every issue since January 1986.

Partsch83 Partsch, H. and Steinbrüggen, R. “Program Transformation Systems.” *Computing Surveys 15*, 3 (September, 1983): 199–236.

Rittel72 Rittel, H. *On the Planning Crisis: Systems Analysis of the ‘First and Second Generations’* (Bedriftsokonomen, NR. 8). Norway: 1972.

Sackman68 Sackman, H., Erickson, W.J., and Grant, E.E. “Exploratory Experimental Studies Comparing Online and Offline Programming Performance.” *Communications of the ACM 11*, 1 (January, 1968): 3–11.

Schach90 Schach, S.R. *Software Engineering*. Boston, MA: Aksen Associates & Irwin, 1990.

Weinberg71 Weinberg, G.M. *The Psychology of Computer Programming.* New York, NY: van Nostrand Reinhold, 1971.