Technical Report
CMU/SEI-92-TR-34
ESC-TR-92-034

Academic Legitimacy of the Software Engineering Discipline

Daniel M. Berry

November 1992

Technical Report
CMU/SEI-92-TR-34
ESC-TR-92-034
November 1992

Academic Legitimacy of the Software Engineering Discipline

Daniel M. Berry
MSE Project

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This technical report was prepared for the

SEI Joint Program Office
ESD/AVS
Hanscom AFB, MA 01731

The ideas and findings in this document should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

Review and Approval

This document has been reviewed and is approved for publication.

FOR THE COMMANDER

(Signature on File)

Thomas R. Miller, Lt Col, USAF
SEI Joint Program Office

The Software Engineering Institute is sponsored by the U.S. Department of Defense.
This report was funded by the Department of Defense.

Table of Contents

List of Figures

Preface

The education-related projects at the Software Engineering Institute (SEI) help the SEI achieve its mission of improving the state of software engineering practice by finding ways it can help improve software engineering education. One way that the group tries to help is to encourage the development of software engineering programs at academic institutions. A number of us have had experience pushing such programs at our former academic institutions. Moreover, we have been involved in setting up a Master of Software Engineering program at Carnegie Mellon University.

When the members of the projects talk at conferences, SEI meetings, and one-on-one with other software engineers at other academic institutions, we hear complaints that faculty members are not being recognized as academically legitimate, that they are not being given promotions and granted tenure. We have come to recognize that the general problem of academic recognition for the discipline and its people is a major impediment to initiation and continuation of academic software engineering programs. Some of the members of the projects have personally experienced these problems. While we do not know the actual numbers, the prevalence of these complaints indicates that the problem is widespread.

Therefore, I have decided to write this report to help software engineers gain the recognition that they deserve for themselves and their academic (as opposed to computer) programs. I hope that it helps.

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 previous draft of this report. 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.

— D.M.B.

Academic Legitimacy of the Software Engineering Discipline

Abstract: This report 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 report 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 report 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 report 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 report 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.

2 Qualifications of the Author

My qualifications to write this document come from my experience dealing with most of the acceptance problems mentioned in the introduction. I was on the faculty of the Computer Science Department at the University of California in Los Angeles, California, USA. for 16 years. I slowly gravitated toward software engineering from programming languages, as software engineering itself was maturing. I left UCLA voluntarily with the rank of full professor to move to the same rank in the Faculty of Computer Science at the Technion in Haifa, Israel. One year after moving there, I was granted tenure. At the Technion, I am now helping to set up an undergraduate program in software engineering. At both UCLA and the Technion, I have done research in formal methods, software design methods and environments, requirements elicitation, and electronic publishing. I continue to be a visiting professor at the Technion while employed here at the SEI, where I am teaching in the Master of Software Engineering (MSE) program offered by the School of Computer Science (SCS) at Carnegie Mellon University. Currently, a discussion is underway to decide the place of the MSE program and its faculty in the SCS. Some of what is described here comes from this personal experience. The rest comes from others who have read and commented on earlier drafts, many of whom have had similar experiences.

3 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.

3.1 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 [Koen85a]

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.

3.2 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 report is essentially an explanation of why they are just as legitimate as their formal cousins.

Another way to divide the subfields is according to where in the working definition they lie. The theory of programs and programming deals with foundations; formal methods, technology, methodology, and management all deal with *how* to produce good software; and the production of software artifacts is the actual doing.

4 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 6, 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.

4.1 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.

4.2 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 systematic, 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.

Section 5.2 describes some of the contributions of this research over the years.

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 farfetched, 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.

4.3 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 9.) 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.

4.4 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.

4.5 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.

5 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.

5.1 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.

5.2 Typical Contributions of Subfields

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

A. Theory of programs and programming:

1. a new theory of program semantics
2. a new set of axioms for program verification
3. a new model of concurrency
4. a new major consistency or completeness theorem
5. a new axiom that makes a new class of programs verifiable

B. Formal methods:

1. a new class of programs subjectable to some formal method
2. a new formal method based on some theory
3. making a formal method significantly accessible to nonmathematicians

C. Technology:

1. a new tool that solves a class of problems not solvable before
2. a new tool that automates an error-prone manual task that was thought to require intelligence
3. controlled experimental demonstration of the effectiveness of a tool
4. integration of diverse tools into a software development environment in a manner significantly better, more seamless, or more complete than before
5. new paradigm for making tools
6. a tool-building tool

D. Methodology:

1. a new method that solves a class of problems not solvable before
2. a new method that systematizes an error-prone, nonautomatable, manual task that requires intelligence
3. controlled experimental demonstration of the effectiveness of a method.

E. Management:

1. a new technique that demonstrably brings a class of heretofore undoable systems into the doable range

2. a new technique that demonstrably overcomes a human or group barrier to software productivity

3. controlled experimental demonstration of effectiveness of a technique

F. Production of software artifacts:

1. solving with software a previously unsolved but hard problem, in particular a problem that has defied solution

2. 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

5.3 Specific Contributions

This section 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.

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:

1. the use of compilers to allow the use of high-level languages instead of machine or assembly language,

2. the use of linkers to allow separate compilation of parts of a program and subsequent combination of their object code into a single loadable program,

3. the use of libraries of already compiled subroutines in programs that are able to invoke them as though they were built-in operators,

4. the use of symbolic debuggers to allow debugging of programs in terms of the identifiers, expressions, statements, and labels used in the source program, and

5. 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:

1. the programming language C, a highly portable high-level language that allows addressing arbitrary machine locations when it is necessary to do so [Kernighan78, Johnson79],
2. the UNIX system in general and in particular,
    a. its portability, having been written almost entirely in C [Johnson78],
    b. the shell, a command processor that is just another invokable program, making its command language another useful interpretive language whose built-in data are the objects of the operating system [Bourne78],
    c. pipes that allow existing programs to be combined by having the output of one fed directly as the input of the next, enabling programmers to implement new functionality without having to write new programs (You should see the pipe that is set up to format this report, a pipe consisting of a half dozen or so programs, each doing a different part of the formatting job!) [Ritchie74],
3. the portable C compiler, which made it very easy to port the C compiler and C programs, such as UNIX, to other machines,
4. on-line full-screen editors that allow programmers to see much more of the program than could be seen with a line editor [Joy76, Stallman81], and
5. windowing systems that allow programmers to keep several views of the same program on screen all at once, views that include the source program, an interactive debugging session, the input of that session, and the output of that session.

The rest of this section describes specific contributions that were intended for software engineering. These contributions are divided into a number of categories:

1. Fundamental truths
2. Methodology
3. Formalisms
4. Tools and environments
5. Testing
6. Sociology and management

7. Metrics and economics
8. Experiments

Within each category, I have made some attempt to follow chronological order; however, showing relations between contributions takes precedence over strict chronological order.

5.3.1 Fundamental Truths

There are a number of documents that exposed fundamental truths, truths that were not accepted or even understood prior to publication of the document.

1. With his now famous waterfall life-cycle model, Win Royce irrevocably changed the way that people viewed the software development process [Royce70]. Prior to the model, software development was viewed as a nonsystematic art. Following the presentation of the model, there was always some attempt to systematize the software development process. Certainly today, the model is viewed as a not-always-ideal ideal that never can be achieved and, in some cases, should not be achieved. Other life-cycle models are in vogue today, particularly incremental models and prototyping models. However, the presentation of the waterfall model gave people an ideal to shoot for and caused people to think in terms of life-cycle models from then on.
2. Laszlo Belady and Meir Lehman introduced the concept of large program and then, arguing from an assumption that correcting a fault in a large program has a small but nonzero (i.e., ε) probability of introducing even more faults, formally explained the observed phenomenon that a continually corrected large program eventually decays beyond usability. They showed that this decay was inevitable and could be avoided *only* by declaring that all remaining bugs are features and freezing the program [Belady71, Belady76].
3. Fred Brooks exposed the truth about software development, that it is very hard, and shattered the myths that had grown up about how to do it, seemingly natural techniques that actually are counterproductive [Brooks75]. The title of the book connotes the still-accepted myth that people and time are exchangeable in software development, when in fact putting more people on a late project is a good way to make it even later. Brooks followed this book with a paper in which he observes that software development is inherently difficult and that there are and will be no magic, quick technological solutions to software problems [Brooks87]. Section 6.1.3 summarizes the argument of this paper.
4. William Swartout and Bob Balzer laid to rest the idea that the life cycle could be as clean as the waterfall model suggests, by showing that specification and implementation of a system are hopelessly intertwined [Swartout82]. The specification will always be modified by what is learned during implementation, and we should make the best of it.

5.3.2 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.

1. Edsger Dijkstra described the difficulties that use of the goto causes to program comprehension [Dijkstra68a] and argues for what is now called structured programming.

2. Niklaus Wirth, Ole-Johann Dahl, Dijkstra, and Tony Hoare all wrote about how to do structured programming or stepwise refinement [Wirth71, Dahl72]. These spawned a concern for programming methods and led to almost universal adoption of that method in education and industry.

3. An early attempt to systematize system design in what might be called a higher level abstraction of structured programming was the book by Michael Jackson on top-down development of programs [Jackson75], soon followed by a book on system design using the same notation and ideas [Jackson83]. His main idea was not to write software but to build a model of the real-life situation for which the software is being built.

4. The programming language SIMULA 67 was among the first to have encapsulating modules, introduced inheritance, and in effect, spawned what is now called object-oriented programming [Birtwistle80, Dahl70].

5. David Parnas introduced the concept of information hiding and showed how it favorably affects maintainability of software [Parnas72]. This work spawned a wide variety of work in methods, languages, and tools for decomposing systems into modular pieces. In particular, Parnas himself described how to design the pieces with an eye to future use in other applications in the same domain [Parnas79].

6. Barbara Liskov and Steve Zilles cast into a concrete form the concept of hidden data abstraction that had been suggested by Parnas's method and Simula 67's classes. They introduced an encapsulating module that implemented abstract data types in a way that hiding the implementing data structure is enforced by the language [Liskov74].

7. Over the years Wayne Stevens, Glenford Myers, and Larry Constantine had observed that uses of certain programming features made program modifications easier and that uses of other features made program modifications harder. Based on these observations, they developed measures of the goodness of proposed decompositions and described a method of system decomposition (programming-in-the-large) in which these measures are used to arrive at good decompositions [Stevens74]. The method, composite design and analysis, turned out to yield almost the same decompositions as did information hiding.

8. Frank DeRemer and Hans Kron identified the fundamental differences between the kind of programming that takes place in individual modules or in programs small enough to be written by one person and the kind of programming that takes place at the system level for programs that are too big for one person [DeRemer76]. The former, programming-in-the-small, is done with conventional programming

languages and is concerned with algorithmic details of working with the data. The latter, programming-in-the-large, is done with module interconnecting languages and is concerned with the overall structure of the system's modules and the interfaces between these modules.

9. Nancy Leveson popularized the concept of software safety, showed that it was *not* the same as other concepts of program goodness, pointed out that it often conflicted with correctness, and suggested methods for building safety into systems by identifying safety problems at requirements engineering and system design time [Leveson86].

About some of these papers (i.e., those of Parnas, of Liskov and Zilles, of Constantine, Myers, and Stevens, of Kernighan and Ritchie, and of Dolotta, Haight, and Mashey) Tony Wasserman, of IDE, said to me "For me, these papers represent the best of what software engineering is all about: original research ideas that lead to widely accepted practical use far beyond the imagination of the inventors."

5.3.3 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.

1. In his seminal paper giving a notation and axiomatic basis for verifying that programs do what they are claimed to do, Hoare opened up the possibility of formally verifying the correctness of programs [Hoare69]. This work was a follow-up to Robert Floyd's earlier work using flowcharts [Floyd67a] and was the precursor to a large amount of work on axiom systems and techniques for verifying the behavior of programs.

2. Ralph London offered a formal verification of a program as a new kind of algorithm certification in the regular algorithm series of the *Communication of the ACM* [London70].

3. Ralph London formally verified a nontrivial compiler for a subset of LISP by showing that the code generated does the same things with any program that the LISP interpreter does [London72].

4. Hans Bekić, Dines Bjørner, Wolfgang Henhapl, Chuck Jones, and Peter Lucas took a different approach to formalize systems, namely that of axiomatizing the transitions of a state machine [Bekić74, Bjørner82]. That approach has come to be known as VDM (Vienna Development Method). Jean-Raymond Abrial, Ian Hayes, Steve King, John Nicholls, Ib Sorensen, Mike Spivey, Bernard Sufrin, Jim Woodcock, and John Wordsworth took a similar approach with the specification language Z [Spivey89, Hayes87]. It is my understanding that these languages and their corresponding software development methods are in use in developing real systems. In particular, an Oxford University group used Z to specify large portions of CICS, IBM's transaction processing system [Nix88a] and won the Queen's award for this work.

5. Following Liskov and Zilles's work on abstract data types, William Wulf, Ralph London, and Mary Shaw showed how an encapsulating abstract data type construction module could be annotated with assertions specifying the properties of the abstraction and how the body of the module could be proved to be consistent with these assertions [Wulf76a].

6. Richard DeMillo, Richard Lipton, and Alan Perlis pointed out the essential differences between proofs done in mathematics to advance mathematical knowledge and proofs done to help ensure program correctness [DeMillo79]. In particular, the lack in the latter of the social processes that surround the former increases the chances of program correctness proofs being incorrect. The three point out the danger of relying on the result of a mechanical program proof—which after all is merely a software-generated binary value, "yes, it is correct" or "no, it is not"—as certification of the correctness of a program, particularly when lives are at stake. The implication of these observations is that the value of program proofs can be increased by recognizing that a proof is but one tool among many that can be used by the entire programming team to ensure that the software they are writing is correct. The proofs should be used as a vehicle for capturing and forcing examination of redundant information about the software being written. Used this way, program proofs then become immersed into a social process not unlike that undergone by mathematical proofs.

5.3.4 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.

1. Stuart Feldman's make program that allows specification of compile-time dependencies between modules. When the user asks for a specific load module to be made, the program tries to do the minimal amount of recompilation, compiling only those modules that have been changed since the last construction or are dependent on a module that has been changed or recompiled [Feldman78, Feldman79].

2. Mark Rochkind's SCCS (Source Code Control System) is a collection of routines that allow a group of people to maintain more than one version of a program in a history of revisions, so that at any time any version and any revision of any module can be inspected and updated with minimal interference with the work of others [Rochkind75a].

3. Walter Tichy improved the organization of the preserved files and the algorithms to develop RCS (Revision Control System) [Tichy81, Tichy85].

These three pioneering tools have spawned other configuration management tools. They are used in far more than just program development. For example, this report is maintained by RCS and is generated by make. These tools have led also to a large interest in software tools and software development environments. Many tools and environments are available; some of the more notable ones are listed in the continuation of this numbered list.

4. Probably the first widely distributed CASE tool was Dan Teichroew and E. A. Hershey's PSL/PSA (Problem Statement Language/Problem Statement Analyzer) a relational database for maintaining system requirements and describing their relations to subsequent design and implementation documents [Teichroew77].

5. The original UNIX environment was called the PWB (Programmer's Workbench) [Dolatta78a] and was regarded as the first integrated software development environment, with pipes as the main way to make the output of one tool be the input of another without having to save that output in a file. Furthermore, the structure of the system makes the addition of each new tool strengthen the total power of the environment even more than one would expect by just the addition of another tool. It is another example of when the whole is greater than the sum of its parts.

6. Warren Teitelman and Larry Masinter's INTERLISP was also regarded as an integrated LISP programming environment, integrated by the fact that all programs were written in LISP and communicate with list structures [Teitelman81].

7. Wayne Cowell and Lee Osterweil put together the first integrated programming environment for FORTRAN programs [Cowell83].

8. STATEMATE is a state-transition diagramming system with interactive graphics input, simulation tools, and code-generating tools [Harel88]. Its major advance over earlier attempts is its careful attention to semantics of the notation, providing distinct notation for distinct concepts, and hierarchical composability to get around the problems of paper, screen, and brain size limiting the size of the model that can be shown at any time.

9. Heloísa Penedo and Don Stuckle described the requirements for a database in support of the software life cycle [Penedo85].

10. Osterweil also noted that descriptions of software development steps made with the help of tools in an environment constitute a program of sorts, specifically a process program. He proposed a process programming language which can be characterized as a strongly typed make with full algorithmic capabilities [Osterweil87].

11. Gail Kaiser and Peter Feiler suggested using knowledge-based data management to build more intelligent integrated programming environments [Kaiser87].

12. Steven Reiss showed how to connect software tools in an environment by using message passing [Reiss90]. This technology is now used in Hewlett-Packard's SoftBench, Sun's ToolTalk, and Digital Equipment Corporation's Fuse.

5.3.5 Testing

Contributions in program testing have been much harder to come by.

1. Lee Osterweil and Lloyd Fosdick built DAVE as a system for testing and validation of FORTRAN programs [Osterweil76]. It found wide use in industry and served as inspiration for many later environments.

2. Glenford Myers was the first to identify the true purpose of program testing, that is to show that the program has faults and *not* to show that it has no faults. His idea was that one should look for *failed* tests and not successful ones [Myers79]. Obviously, the theory tells us that testing cannot be used to show absence of faults and can be used to show their presence. However, the real issue is psychological, one of maximizing the success and effectiveness of testing by having the appropriate goals.

3. John Goodenough and Susan Gerhart attempted to find a formal basis for program testing by exploring the possibility of identifying for a program a finite number of test cases such that the program behaving correctly on them is a formal proof of the correctness of the program [Goodenough75]. Although the problem is, in general, too difficult to solve for an arbitrary program, the work offers guidelines for a systematic selection of test data.

4. Bill Howden defined the concept of reliability of a set of test data, backed up that definition with a usable method to analyze the reliability of path testing, and showed that the method is applicable to a well-known published set of incorrect programs [Howden76].

5. This work was followed by a more complete mathematical framework by John Gourlay for characterizing testing [Gourlay83].

The goal of research in software testing is to find a way to automatically generate test data that test a program sufficiently to find all faults in the program. Testing all possible data is out of the question because for all but the most trivial programs, the number of possibilities is too large. Therefore, a representative sample of test data is used. It is certainly easy not to test enough. Ideally one would like to test only the test data that will expose all the faults in the program. However, that is impossible to arrange as it requires knowledge in advance of the faults; and if they are known in advance, there is no need to test. So we settle for test data that in some way cover the program sufficiently. Moreover, it has been recognized that test case *generation* cannot be automated much beyond random generation. Random test case generation is viable only when the problem domain is very well-structured, and in any case, a major problem with randomly generated test cases is that there is no convenient way to know what the correct answer should be. Researchers have settled for automatic checking, according to some criteria, of human-generated test data. Probably, the most significant work is in the area of test-data adequacy criteria and comparing the coverage of proposed criteria.

One class of methods for generating test data is the black-box methods based only on specifications of the code to be tested. This class includes generation of random data as well as functional testing.

The other class of methods, for both generating test data and evaluating test data coverage, is the structural methods based on the code to be tested. This class includes the path coverage methods, such as causing every statement to be executed once, causing every branch to be followed once, and causing every path to be executed once. Also in this class are mutation testing and methods based on data flow.

6. William Howden described methods for generating test cases from a specification of the function of a program [Howden75]. These methods later were classified as functional testing [Howden80].

7. Richard DeMillo, Richard Lipton, and Fred Sayward [DeMillo78a] and, independently, Dick Hamlet [Hamlet77a] proposed the widely used program testing technique called mutation testing. A program and mutants, each obtained by modifying the program in one statement, are run on the same test data in order to expose faults.

8. Janusz Laski and Bogdan Korel found a way to use the analysis of a program's data flow to find a testing strategy for the program, that is, to determine which parts of the program are worth testing [Laski83]. Heretofore, data flow analysis had been used only in compiler optimization, and structured test case generation methods had focused only on control flow to identify paths to be tested.

9. Susan Rapps and Elaine Weyuker took the data flow idea one step further by using the information determined in data flow analysis to generated test data [Rapps85].

10. Testing all paths suffers from combinatorial explosion. One way to cut down on the number of paths to test is to consider only paths between the definition and use of variables. Even with this reduction, the theoretical upper bound on the number of paths is 2^d where d is the number of decisions in the program. However, Weyuker was able to demonstrate empirically that in practice this upper bound is not reached; the actual number of definition-use paths tends to be proportional to d [Weyuker88]. This observation makes definition-use-path coverage an eminently practical test case generation technique.

The next contribution represents a completely different kind of testing.

11. Michael Fagan described a testing technique that has become known as the inspection. An inspection is similar to a walkthrough, but it is more formal, involving five steps: an overview, preparation, the inspection itself, rework, and follow-through. The inspector's purpose is to find and report as many faults as possible. The correction of the faults is left to the author, although the inspector makes sure that the faults are corrected. While inspections were originally applied to program design, they are now widely used for any product of the software life cycle as a means to find errors in that product [Fagan74, Fagan76]. Fagan backed his methodological claims with an experimental verification that formal design inspections, as he described them, are effective in finding errors than less formal design walkthroughs. For the application development whose inspections he measured, he found that 67% of all faults that were ever found were found during the formal inspection; and during actual use, the inspected product exhibited 38% fewer faults than did a comparable product that was subjected only to walkthroughs.

5.3.6 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.

1. Gerald Weinburg's study of the psychology of computer programmers and programming showed that the psychological issues are so important that projects are made or broken according to how they are handled [Weinberg71].

2. One of the earliest attempts to put some structure into the team of programmers working on one program and to systematize their behavior was the chief programmer team popularized by Terry Baker and Harlan Mills [Baker72, Mills71]. What was most interesting about this approach was that the top-down team structure matched the top-down programming method that was recommended by Baker and Mills.

3. Ben Shneiderman's early book on designing user interfaces pointed out the importance of getting the user involved in the process and described prototyping as a key technique for doing just that [Shneiderman84].

5.3.7 Metrics and Economics

A little more accepted than the sociological and management work has been the work in software metrics and software engineering economics.

1. Until data were kept, software people were in the dark as to where time, effort, and thus money was being spent in software development. Slowly, as the data were gathered we began to learn the truth. In 1976, Barry Boehm observed that the cost of hardware was shrinking and the cost of software development and maintenance was growing [Boehm73]. He predicted that by 1985, the cost of software development and maintenance would grow to be 85% of the total cost of computing systems, with hardware only 15%, and that maintenance would grow to be 73% of the cost of software. He was essentially right. The impact of this finding was to tell us on what we should be focusing our research, namely in making software development and maintenance, maintenance in particular, more effective.

2. Albert Endres of IBM Germany studied the testing of DOS for the 360 and found that a majority of the errors were confined to the half dozen or so largest modules of the system, which were in all likelihood the most complex [Endres75]. The results showed that fault detection should be focused on complex modules and that a good way to eliminate faults is to reduce complexity.

3. Tom McCabe introduced the cyclomatic complexity measure, basically the number of decisions made in a module, as a useful predictor of costs and faults [McCabe76]. Like others, it was not well received at first, but experimentation has shown the metric to be useful.

4. James Elshoff also collected and analyzed data in 1976 and reported that already maintenance accounted to more than 60% of the cost of software [Elshoff76].

5. Maurice Halstead attempted to identify fundamental measures about software stemming from the fact that software is written with programming languages and programming languages, like all other languages, have operators, operands, and

characteristic ways that they are combined and used [Halstead77]. The basic measures for any program are the numbers of operators and operands and the numbers of unique operators and operands. Halstead derived formulae predicting software effort based on these measures. Initially, this software science was considered even more a pseudo-science than methodology work; however, there has been experimental verification of the accuracy of the predictions of software science and thus of the theory itself.

6. Closely related to this work was the attempt to determine the actual source of errors. In 1979, Boehm reported that between 60% and 70% of all faults detected in large software systems are specification or design faults [Boehm79]. That is, the faults were programmed right into the systems from the beginning. Boehm also observed that the cost to fix a fault grows exponentially with the stage of the life cycle in which it is found [Boehm80]. A fault that costs $1.00 to fix in the specification stage costs $10.00 to fix during implementation and $100.00 to fix during maintenance. These results drove home the importance of finding errors early.

7. Boehm's study of the economics of software development showed how poorly managers were estimating the resources needed for software development, showed the consequences of these poor estimates, and showed how to estimate more accurately [Boehm81].

8. In his landmark book on software engineering economics [Boehm81], Boehm introduced the COCOMO model of software cost-estimation. The model consists of a number of formulae relating an estimated product length (in delivered source instructions), a rough level of difficulty (high, medium, low), and 15 software development effort multipliers to obtain a nomimal effort in person-months. He gave typical values for the multipliers in the formulae, derived presumably from his own considerable management experience at a successful software production company. Although the formulae and numbers might seem as though they were pulled out of a hat, the data from 63 projects in a variety of application areas have validated the predictions given by the model to be accurate within 20%. No other predictor has been shown to be better, and many are worse.

5.3.8 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.

After many years of wild methodological claims and debate as to which programming language features were helpful or harmful, a number of researchers began to do experimental verification of software science, effectiveness of methods and language features, and in general of the folklore about software development.

1. H. Sackman, W. Erickson, and E. Grant attempted to compare the effectiveness of programmers working interactively to that of programmers working in batch mode [Sackman68]. The folklore says that working interactively is better than working in batch mode. While the data were inconclusive for the original question, the experiment did manage to show that there were differences of up to 28 to 1 in the competence of programmers that appeared to be equal in experience, i.e., years of

programming and application areas. This result showed the difficulty, indeed the virtual impossibility, of drawing significant conclusions from so-called controlled experiments involving human programmers. Individual skill differences dominate all other factors that can be controlled; moreover, the obvious methods of controlling skill differences do not control them.

2. John Gould and Paul Drongowski were able to show experimentally that debugging proceeds faster when the debuggers have had previous experience with the program, that assignment bugs are harder to find than others, and that the debugging tools they tested did not really help programmers debug faster [Gould74].

3. Generally, attempts to determine if the folklore about good features and bad features is correct have not yielded conclusive results. That itself is a significant contribution because it confirms that the whole language feature issue is not really important in software engineering; lousy programs can be written in any language, even the cleanest; and good programs can be written in any language, even FORTRAN and COBOL!

4. John Gannon, in one of the few successful experiments about language features, showed that strong typing can be helpful [Gannon77].

5. While testing programmers was found to be problematic, testing the validity of measures of programs proved to be more fruitful. Bill Curtis, Sylvia Sheppard, and Phil Milliman showed that complexity measures of Halstead and of McCabe are better predictors of time to find the source of a fault than is the number of lines of code [Curtis79].

6. John Bailey and Vic Basili were able, by compiling data on 18 large NASA programs, to develop a cost prediction metric accurate to one standard deviation [Bailey81a] and to show that the cost of reusing existing code was about 20% of the cost of developing new code for the same function. Of course, the actual numbers of these results are specific to the organization, its applications, and its personnel; but the technique can be carried out anywhere to get useful predictive data for that place.

7. Mark Weiser showed that debugging programmers use slicing; that is, they routinely strip away parts of the program that do not influence the problematic variable at the problematic statement [Weiser82].

8. Basili and David Hutchens showed that various complexity metrics are not much better than numbers of lines of code in predicting faults [Basili83].

9. Allan Albrecht and John Gaffney defined function points, a measure of the size of a software product, in terms of the numbers of inputs, outputs, master files, and interfaces used by the product. They reported some experiments that showed that function points form a better basis for predicting effort to build a product that do lines of code [Albrecht83].

10. Basili, Richard Selby, and Hutchens describe a framework, based on iterative learning, for classifying, planning, carrying out, and evaluating experiments in software engineering in order to ensure that the experiment is testing what it purports to and the results are interpreted properly [Basili86].

11. Having multiple copies of identical units is a popular and effective way to increase hardware reliability and to prevent the failure of a single component from bringing down the whole system. At any time, the surviving copies of any unit vote on the result to be reported. There was an attempt to apply the same idea to software. Since multiple copies of the same program will always behave the same, the idea was modified to have the participants in the voting be programs that were independently developed from the same specifications. For the voted result to be more correct more often than any one program's result requires an assumption of complete independence in the different developments from the same specifications. Nancy Leveson and John Knight showed with a carefully designed, large-scale experiment that this assumption of independence is not justified [Knight86].

12. Selby, Basili, and Terry Baker compared the Cleanroom software development method with the then standard team approach [Selby87]. Although the data were generally inconclusive, the products developed with the Cleanroom method passed a higher percentage of the test cases than those developed with the other methods.

13. Basili and Selby compared black-box and glass-box testing and one-person code reading for their effectiveness in finding faults [Basili87]. The participants were professional programmers and advanced computer science students. There were differences between the way the different kinds of programmers performed, but generally code reading found more interface faults than the other methods, and black-box testing found more control faults than the other methods. Basili and Selby used a technique called fractional factorial experiment design to factor out differences in the ways different participants tested their different programs [Basili84].

5.3.9 Best Papers

It is interesting to list the papers that have received the best paper awards at the International Conference on Software Engineering. These awards started at the ninth conference, and for the first two years, the award was presented to a paper from the same conference. Starting with the eleventh conference, it was decided to present the award to a paper from the tenth previous conference.

The recipient papers were

9. Dewayne Perry's "Software Interconnection Models" (from ninth conference) [Perry87],

10. Harel et al's "STATEMATE: A Working Environment for the Development of Complex Reactive Systems" (from tenth conference) [Harel88],

11. Rochkind's "The Source Code Control System" (from first conference) [Rochkind75b],

12. Wulf, London, and Shaw's "An Introduction to the Construction and Verification of Alphard Programs" (from second conference) [Wulf76b],

13. Parnas's "Designing Sofware for Ease of Extension and Contraction" (from third conference) [Parnas78], and

14. Tichy's "Software Development Based on Module Interconnection" (from fourth conference) [Tichy79].

Many of these papers or later versions thereof have been cited earlier for their specific contribution.

6 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.

6.1 Programming Is Hard

> "Software engineering ... is the part of computer science that is too difficult for the computer scientists."
>
> — F.L. Bauer [Bauer71a]

6.1.1 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].

6.1.2 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 continues by describing the process of software development for P-type and E-type programs. He dismisses the rigid life-cycle models, e.g., the waterfall model, as not capturing the reality of continuous feedback, evolution, upgrading, and bug fixing that go hand-in-hand with software that is part of the system it controls and that is never finished.

He offers the LST model [Lehman84], on which the ISTAR [Lehman86a] environment was based, as a model of the software process. The software process is a multistep sequence in which, in each step, a human transforms a base into a target. For each step, the base is a specification and the target is an implementation of the specification. Only the first step has no formal basis. It must rely on informal verbal descriptions of the domain and its

objects, attributes, relations, events, and activities. In the case of an E-type system, the model produced in this first step involves

> an act of mental abstraction.... This is a transient, unobservable, act that cannot be completely recorded, controlled, or, in general, revisited. Even if the representation produced by this act can be completely formalised, the mental abstractions that must be undertaken to create the finite, discrete representation of the unbounded, continuous real world are tantamount to a nonformal representational component. For this reason alone, the development process of E-type systems must always display some characteristics of nonformal development.... The E-type program development paradigm may ... be described as *the transformation of a nonformal model of a real world application in its application domain to a formal representation of the programmatic part of a solution system in association with a nonformal support system, the whole operative in the real world solution domain.*

Thus, carrying out verifications of implementations with respect to specifications in the software process is insufficient in the case of E-type software. Consistency of the implementation and the specification says nothing about whether the specification and the implementation are in fact solutions to the problem at hand. At each step along the way, each new implementation representation must be validated against the original problem to be solved.

Lehman gives his estimate as to the difficulty of programming E-type systems. He argues that from the LST process model it follows that an E-type program may be considered a model of a model of a model of ... a computer application in the real world. Turski's view is that the mathematical concept of model dominates the theory and practice of software development [Turski81]. At one extreme, the real-world application and the final implementation are considered models of the specification, which then serves as a bridge between concept and realization. At the other extreme, the steps form a strict chain, in which each successive basis-target or specification-implementation pair forms a theory-model pair. It then follows, Lehman argues, that every E-type program is Gödel incomplete. The properties of an E-type program cannot be completely known from the set of formal models generated during program development. Thus, the humans involved in system development and usage become an integral and active part of the system, and these humans will constantly discover new properties of the system, either that they did not know that it had or that it now needs. "The degree of satisfaction that the operational system yields is ultimately determined by their judgment, action, and inaction. Thus, neither developers nor users can fully know system properties." This incomplete knowledge is true of even P-type programs like editors and shells. How many of you have had the experience of discovering some neat new tricks that you can do with the commands already in your favorite editor or shell? How many of you have grown weary of dealing with intellectual inertia against learning a new version of your favorite operating system each time the producer releases one? This incomplete knowledge is certainly true of E-type programs. There

is much less known about the art of writing E-type programs for any given application, simply because there is *no* history with the application.

Thus, clearly programming E-type systems is intellectually as difficult as building an entirely new formal system from first principles, discovering theorems as you go, and discovering a clean development from a minimal set of axioms.

Lehman goes on to discuss the uncertainties involved with software development in order to explain why E-type software is subject to constant changes. It is the need to cope with these changes that gives software engineering researchers the greatest challenge and the most fertile source of problems to be solved.

First, from the Gödel incompleteness mentioned above comes what Lehman calls a Gödel-like uncertainty, an uncertainty arising from the fact that neither developers nor users can fully know system properties. Secondly, there is a Heisenberg-like uncertainty that starts as the system is being developed and grows during use. The very processes of development and of use change the perception and understanding of the application itself. The installation of the system irrevocably changes the operational environment in which the application resides. The organization can never go back to the original unautomated system because it begins to schedule and commit itself to a pace allowed by the automation. Users find a mismatch between their expectations and the reality both in performance and interface and begin to demand change. In addition, the same users begin to see better ways to perform existing functions (i.e., better interfaces or more efficient procedures, and new useful functions) and begin to demand their implementation. Note that it is not until a system is running that users will even conceive of these new functions, simply because these new functions are meaningless in the old, unautomated way of doing things. Since the criteria of acceptance of type-E systems is user satisfaction, E-type systems must change or face an inevitable deterioration of acceptance as users become less and less satisfied. Therefore, as Lehman states, “Software evolution is intrinsic and not, as sometimes thought, primarily due to shortsightedness of developers or users.”

The point is that all man-made systems evolve and, in particular, software evolves. The rate of evolution of a *used* software system will far exceed the rate of evolution of a biological or physical system. This is because the fundamental laws of biological and physical systems do not change, while the laws of software, being entirely man-made and so malleable, can be and are changed at a moment’s notice. Unfortunately, while it is easy to *decide* to change software, it is another thing entirely to actually change it. Anyone who has changed software, no matter how small the change, has learned the hard way how intricately interconnected all software is and how the slightest change in one part of a program can have a major impact on the rest of the program. In fact, it is clear that the term “slightest” change, implying a mathematical continuity reminiscent of natural systems, is an oxymoron, as software is notoriously discrete, noncontinuous, and unstable in response to change. Each change, to be made safely, requires an understanding of the total system, an understanding which, as we have seen, is not there. In fact, in a sense, here is the fundamental contradic-

tion. If we knew enough about the program to make the change safely, we probably would know enough about the program and the application not to even have to make the change. As Lehman puts it, "The fact that the inexperienced believe software change to be trivial because it does not require expenditure of physical effort and because the intellectual effort needed to make them correctly is under-estimated, compounds the problem, increasing the pressure for rapid change. As already remarked, response to this pressure cannot be neglected without decline in the satisfaction that the system yields. This is the underlying cause of the high level of software maintenance activity associated with all serious application of computers."

Because changes are not easy to make, there is a delay between the onset of pressure for a change and the successful completion and integration of the change. This is compounded by and is compounding the pressure for change. The discovery of new needed or desired changes continues even as changes are being made. With staff limitations, later requests are delayed even more. These delays increase the pressure for change. There is thus no escape from mismatch between user expectations and user perceptions about the system. Since the behavior of the user cannot be predicted, there is no way to predict the degree of satisfaction that a change will elicit. Each change temporarily decreases knowledge about the system, even though the purpose of the change was to increase that knowledge, or rather to decrease the lack of knowledge. These conflicting knowledge tendencies give rise to uncertainty about system behavior relative to users' expectations. This uncertainty may be called *Heisenberg-like uncertainty*.

The third kind of uncertainty is also present in all human endeavors, *process uncertainty*. There is no way to predict the effectiveness of the humans in carrying out any process, especially one involving thinking. Thus there is no way to predict how well human programmers will carry out their intents with respect to the quality of the software they produce, independent of whether they correctly understand what the software is to do. In other words, even if the software is S type, we cannot predict how well the code will meet the specification. Even if a verification is carried out, a mistake could occur that compensates for a mistake in the software itself.

These three uncertainties make software production an extremely hard problem, even harder than most mathematical problems. Basically, the domain of an E-type program for a real-world application is not definable; it is unbounded, continuous, and changing dynamically in unpredictable ways. Any solution program is finite, discrete, and, if left alone, static and unchanging. Therein lies a gap that can never be closed. Even as software is changed, what is produced is static until the next change. In the meantime, the real-world domain has been changing continuously. Moreover, one can never be sure that the new program satisfies its requirements as well as the previous one. Lehman concludes with what he calls the basic *uncertainty principle of computer application*. "The outcome, in the real world, of software system operation is inherently uncertain with the precise area of uncertainty not knowable." Solving these problems in the face of these uncertainties to produce

solutions that satisfy, in spite of not knowing what it is that they satisfy, is the heart and essence of software engineering.

6.1.3 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 6-1 on page 46, 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]. Good software practice mitigates against duplication of parts since two parts that are similar are collapsed into a single callable routine in which the differences have been parameterized. In other more physical, human-built entities (bridges, buildings, cars, computers, etc.) the most obvious characteristic is the repetition of parts.

   Digital computers have large numbers of states, many more than most other things we build. However, there is still only a finite number of states. A typical program has orders of magnitude more states than hardware; and in many cases the number of states in the abstraction is unbounded and if it involves real numbers, uncountable. In addition, the very attempt to fit this abstraction with an unbounded number of states into a machine with a finite number of states adds still more complexity to the software and is a fruitful source of faults.

   When a bigger bridge, building, car, or computer is built, the complexity does not go up significantly because the construction just requires larger numbers of the same units that are used to build their smaller versions. Doubling the size of a program means quadrupling the relationships. If a program can be made bigger simply by repeating some code already in it, then it will not be made bigger; that repeatable code will be made a procedure that will be invoked as many times as necessary at the cost of only one new line per repetition.

   Advances in mathematics and the physical sciences have come by construction of simplifying models that ignore complexities not relevant to the phenomenon being studied. In software, ignoring complexities yields an incomplete program that cannot deal with the real-world situation for which it was constructed. Certainly, abstraction (ignoring details) is useful for decomposing the software into manageable pieces. However, ultimately these ignored details must be considered to complete the program. The complexity of software simply cannot be escaped.

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. Certainly, software is far more malleable than are hardware, firmware, and people. Moreover, the software is usually the last component of the system to arrive on the scene and is usually the only one developed on site. That is, the off-the-shelf hardware and firmware come with predetermined behavior, and the behavior of human users is fairly well set, to the extent that bibliographical descriptions of human behavior are still valid. Thus, it is the software that is adapted to conform to the rest of the system.

   Compounding the difficulty is the fact that this conformity is nonuniform. The interface to the hardware and the firmware is different from the interface to the users. Each thing that the software must conform to has been developed by a different independent agent. Brooks states a belief that conforming to a variety of man-made objects is far more difficult than what must be done in nature; his belief, shared by many scientists, is that nature has some unifying simplicity. The

work of a scientist is to find this simplicity. The work of a programmer is to conform to complexities.

3. **Changeability**: Software is subject to change far more than other technology. Once delivered, computer hardware, buildings, and vehicles undergo changes only rarely. The cost of changes in these is a large enough fraction of the cost of the original construction that replacement is usually chosen as more cost-effective than change. In practice, the cost of a change in software can be very high. The cost of maintaining a program over its lifetime is more than the cost of developing it; also because of potential ripple effects, the cost of making a minor change can be major. However, the perception is that the cost to change a statement or a routine is small.

   This perception and the very malleability of software subject it to pressures for change from two main sources.

   1. As a system is used successfully, the users find new functions that the system should have to make it even more useful.
   2. A good program survives the hardware platform for which it was built; hence it must, over time, be adapted to new hardware, new input and output devices, new networking technology, etc.

   These pressures to change make the abstraction that the software represents, the complex animal that must be understood before the software can be built, a moving target.

4. **Invisibility**: Brooks observes that "Software is invisible and unvisualizable." For objects with a geometric reality, such as buildings, diagrams are faithful, complete, and useful representations from which inconsistencies and omissions can be gleaned. For computer hardware, circuit diagrams have the same completeness property. However, software has no physical reality. Thus, save for the code itself, it has no complete representation. Diagrams at best capture some aspects of a system but not all of them. Thus, we have data flow diagrams, control flow diagrams, module interconnection diagrams, etc. None captures all aspects of the system; and when we have a collection of such diagrams, their mutual consistency is not at all assured. Moreover, since these diagrams are not complete, our ability to detect inconsistencies and omissions in them is limited, especially if the inconsistency or omission falls between the cracks between what is represented by the various diagrams.

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 5.3 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.

6.1.4 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 [Manna71a] 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.[1]

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.

6.1.5 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.

6.1.6 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 6-1 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

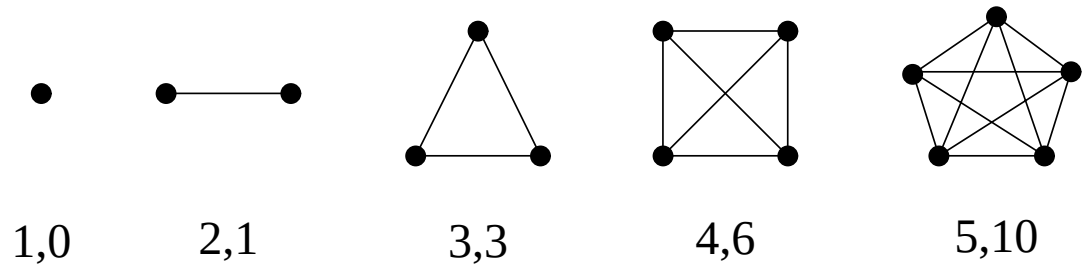

Figure 6-1: Numbers of parts and of possible interactions between them

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.

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* [Leavenworth70a] 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.

6.2 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.

6.3 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.

7 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.

8 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.

9 Snake-Oil Salespeople

A common complaint heard is that there are many people selling software engineering snake oil and many charlatans who do no substantial work. These people are described as evangelists for their own methods, which they claim will solve *all* the world's problems. Probably they have consulting companies that sell the method for megabucks, and they are interested only in advancing the fortunes of the company.

This complaint is true to some extent. However, there is no reason for good research to be discounted because of the existence of poor work. Every field has its share of bad work, papers that should never have seen the light of day, charlatans who sell snake oil instead of substantial results, evangelists who promote themselves or their companies, incompetents who feed off the work of others, and all the other misfits that seem to thrive in an academic environment. (Do not get me wrong; I *am* an academic, but I do recognize the drawbacks of academic life.) Yes, there is poor work done in *every* field. Poor work is a reflection on the person who makes it and the people who accept it and not on the field itself or the people in it who do good work.

10 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.

Bibliography

Albrecht83 Albrecht, A.J. & Gaffney, J.E. Jr. "Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation." *IEEE Transactions on Software Engineering SE-9*, 6 (November 1983): 639–648.

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

Bailey81 Bailey, J.W. & Basili, V.R. "A Meta-Model for Software Development Resource Expenditures," pp. 107–116. *Proceedings of the Fifth International Conference on Software Engineering*. San Diego, CA, March 1981.

Baker72 Baker, F.T. "Chief Programmer Team Management of Production Programming." *IBM Systems Journal 11*, 1 (1972): 56–73.

Basili83 Basili, V.R. & Hutchens, D.H. "An Empirical Study of a Syntactic Complexity Family." *IEEE Transactions on Software Engineering SE-9*, 6 (November 1983): 664–672.

Basili84 Basili, V.R. & Weiss, D.M. "A Methodology for Collecting Valid Software Engineering Data." *IEEE Transactions on Software Engineering SE-10*, 6 (November 1984): 728–738.

Basili86 Basili, V.R.; Selby, R.W. Jr.; & Hutchens, D.H. "Experimentation in Software Engineering." *IEEE Transactions on Software Engineering SE-12*, 7 (July 1986): 733–743.

Basili87 Basili, V.R. & Selby, R.W. Jr. "Comparing the Effectiveness of Software Testing Strategies." *IEEE Transactions on Software Engineering SE-13*, 12 (December 1987): 1278–1296.

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

Bekić74 Bekić, H.; Bjørner, D.; Henhapl, W.; Jones, C.B.; & Lucas, P. *A Formal Definition of a PL/I Subset, Parts I and II* (Technical Report 25.139). Vienna: IBM Laboratory, December 1974.

Belady71 Belady, L.A. & Lehman, M.M. *Program System Dynamics or the Metadynamics of Systems in Maintenance and Growth* (Research Report RC3546). Yorktown Heights, NY: IBM, September 1971.

Belady76 Belady, L.A. & Lehman, M.M. "A Model of Large Program Development." *IBM Systems Journal 15*, 3 (1976): 225–252.

Birtwistle80 Birtwistle, G.M.; Dahl, O-J.; Myhrhaug, B.; & Nygaard, K. *Simula Begin*. Lund, Sweden: Studentlitteratur, 1980.

Bjørner82 Bjørner, D. & Jones, C.B. *Formal Specification and Software Development*. Englewood Cliffs, NJ: Prentice-Hall, 1982.

Boehm73 Boehm, B.W. "Software and its Impact: a Quantitative Assessment." *Datamation 19*, 5 (May 1973): 48–59.

Boehm79 Boehm, B.W. "Software Engineering, R&D Trends, and Defense Needs," *Research Directions in Software Technology*. ed. P. Wegner. Cambridge, MA: MIT Press, 1979.

Boehm80 Boehm, B.W. "Developing Small-Scale Application Software Products: Some Experimental Results," pp. 321–326. *Proceedings of the Eighth IFIP World Computer Congress*, October 1980.

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

Boehm84 Boehm, B.W.; Gray, T.E.; & 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.

Bourne78 Bourne, S.R. "UNIX Time-Sharing System: The UNIX Shell." *Bell System Technical Journal 57*, 6 (June 1978): 1971–1990.

Brooks75 Brooks, F.P. Jr. *The Mythical Man-Month: Essays on Software Engineering*. Reading, MA: Addison-Wesley, 1975.

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.

Cowell83 Cowell, W.R. & Osterweil, L.J. "The Toolpack/IST Programming Environment," pp. 326–333. *Proceedings of Softfair, A Conference on Software Development Tools, Techniques, and Alternatives*. Crystal City, VA, July 1983.

Curtis79 Curtis, B.; Sheppard, S.B.; & Milliman, P. "Third Time Charm: Stronger Prediction of Programmer Performance by Software Complexity Metrics," pp. 356–360. *Proceedings of the Fourth International Conference on Software Engineering*. Munich, FRG, September 1979.

Dahl70 Dahl, O-J.; Myhrhaug, B.; & Nygaard, K. *Common Base Language (SIMULA 67)* (Publication No. S-22). Oslo, Norway: Norwegian Computing Center, 1970.

Dahl72 Dahl, O.-J.; Dijkstra, E.W.; & Hoare, C.A.R. *Structured Programming*. London: Academic Press, 1972.

DeMillo78 DeMillo, R.A.; Lipton, R.J.; & Sayward, F.G. "Hints on Test Data Selection: Help for the Practicing Programmer." *Computer 11*, 4 (April 1978): 34–41.

DeMillo79 DeMillo, R.A.; Lipton, R.J.; & Perlis, A. "Social Processes and Proofs of Theorems and Programs." *Communications of the ACM 22*, 5 (1979): 271–280.

DeRemer76 DeRemer, F. & Kron, H.H. "Programming-in-the-Large vs. Programming-in-the-Small." *IEEE Transactions on Software Engineering SE-2*, 2 (June 1976): 80–86.

Dijkstra68 Dijkstra, E.W. "Go To Statement Considered Harmful." *Communications of the ACM 11*, 3 (March 1968): 147–148.

Dolatta78 Dolatta, T.A.; Haight, R.C.; & Mashey, J.R. "UNIX Time-Sharing System: The Programmer's Workbench." *Bell System Technical Journal 57*, 6 (June 1978): 2177–2197.

Elshoff76 Elshoff, J.L. "An Analysis of Some Commercial PL/1 Programs." *IEEE Transactions on Software Engineering SE-2*, 2 (June 1976): 113–120.

Elspas72 Elspas, B.; Levitt, K.N.; Waldinger, R.J.; & Waksman, A. "An Assessment of Techniques for Proving Program Correctness." *Computing Surveys 4*, 2 (June 1972): 81–96.

Endres75 Endres, A. "An Analysis of Errors and their Causes in System Programs." *IEEE Transactions on Software Engineering SE-1*, 2 (June 1975): 140–149.

Fagan74 Fagan, M.E. *Design and Code Inspections and Process Control in the Development of Programs* (Technical Report IBM-SSD TR 21.572). IBM Corporation, December 1974.

Fagan76 Fagan, M.E. "Design and Code Inspections to Reduce Errors in Program Development." *IBM Systems Journal 15*, 3 (1976): 182–211.

Feldman78 Feldman, S.I. "Make — A Program for Maintaining Computer Programs," *UNIX Programmer's Manual, Seventh Edition*. Murray Hill, NJ: Bell Laboratories, 1978.

Feldman79 Feldman, S.I. "Make — A Program for Maintaining Computer Programs." *Software—Practice and Experience 9*, 4 (April 1979): 224–265.

Floyd67 Floyd, R. "Assigning Meanings to Programs," pp. 19–31. *Proceedings of Symposium on Applied Mathematics*, Vol. 19. Providence, RI: American Mathematics Society, 1967.

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.

Gannon77 Gannon, J.D. "An Experimental Evaluation of Data Type Conventions." *Communications of the ACM 20*, 8 (August 1977): 584–595.

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

Gould74 Gould, J.D. & Drongowski, P. "An Exploratory Study of Computer Program Debugging." *Human Factors 16*, 3 (1974): 258–277.

Gourlay83 Gourlay, J.S. "A Mathematical Framework for the Investigation of Testing." *IEEE Transactions on Software Engineering SE-9*, 6 (November 1983): 686–709.

Halstead77 Halstead, M.H. *Elements of Software Science*. New York, NY: Elsevier North-Holland, 1977.

Hamlet77 Hamlet, R.G. "Testing Programs with the Aid of a Compiler." *IEEE Transactions on Software Engineering SE-3*, 3 (July 1977): 279–290.

Harel88 Harel, D.; Lachover, H.; Naamad, A.; Pnueli, A.; Politi, M.; Sherman, R.; Shtul-Trauring, A.; & Trakhtenbrot, M. "STATEMATE: A Working Environment for the Development of Complex Reactive Systems," pp. 396–406. *Proceedings of the Tenth International Conference on Software Engineering*. Singapore, April 1988.

Hayes87 Hayes, I. *Specification Case Studies*. Englewood Cliffs, NJ: Prentice-Hall, 1987.

Hoare69 Hoare, C.A.R. "An Axiomatic Basis for Computer Programming." *Communications of the ACM 12*, 10 (October 1969): 576–580,585.

Howden75 Howden, W.E. "Methodology for Generation of Test Data." *IEEE Transactions on Computers C-24*, 5 (May 1975): 554–559.

Howden76 Howden, W.E. "Reliability of the Path Analysis Testing Strategy." *IEEE Transactions on Software Engineering SE-2*, 3 (September 1976): 208–216.

Howden80 Howden, W.E. "Functional Program Testing." *IEEE Transactions on Software Engineering SE-6*, 2 (March 1980): 162–169.

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

Jackson75 Jackson, M.A. *Principles of Program Design*. London: Academic Press, 1975.

Jackson83 Jackson, M.A. *System Design*. Englewood Cliffs, NJ: Prentice-Hall, 1983.

Johnson78 Johnson, S.C. & Ritchie, D.M. "Portability of C Programs and the UNIX System." *Bell System Technical Journal 57*, 6 (June 1978): 2021–2048.

Johnson79 Johnson, S.C. "A Tour through the Portable C Compiler," *UNIX Programmer's Manual, Seventh Edition*. Murray Hill, NJ: Bell Laboratories, January 1979.

Joy76 Joy, W. *An Introduction to Display Editing* (Technical Report). Berkeley, CA: Electrical Engineering Computer Science Department, University of California at Berkeley, 1976.

Kaiser87 Kaiser, G.E. & Feiler, P.H. "An Architecture for Intelligent Assistance in Software Development," pp. 80–88. *Proceedings of the Ninth International Conference on Software Engineering*. Monterey, CA, March 1987.

Kernighan78 Kernighan, B.W. & Ritchie, D.M. *The C Programming Language*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1978.

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

Knight86 Knight, J.C. & Leveson, N.G. "An Experimental Evaluation of the Assumption of Independence in Multiversion Programming." *IEEE Transactions on Software Engineering SE-12*, 1 (January 1986): 96–106.

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.

Laski83 Laski, J.W. & Korel, B. "A Data Flow Oriented Program Testing Strategy." *IEEE Transactions on Software Engineering SE-9*, 3 (May 1983): 347–354.

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

Lehman84 Lehman, M.M.; Stenning, N.V.; & Turski, W.M. "Another Look at Software Design Methodology." *Software Engineering Notes 9*, 2 (April 1984): 38–53.

Lehman86 Lehman, M.M. "Model Based Approach to IPSE Architecture and Design." *Software Engineering Notes 11*, 4 (August 1986): 49–60.

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

Leveson86 Leveson, N.G. "Software Safety: What, Why, and How." *Computing Surveys 18*, 2 (June 1986): 125–164.

Liskov74 Liskov, B.H. & Zilles, S.N. "Programming with Abstract Data Types." *SIGPLAN Notices 9*, 4 (April 1974): 50–60.

London70 London, R.L. "Certification of Algorithm 245[M1] Treesort 3: Proof of Algorithms—A New Kind of Certification." *Communications of the ACM 13*, 6 (June 1970): 371–373.

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.

London72 London, R.L. "Correctness of a Compiler for a LISP Subset." *Proceedings of an ACM Conference on Proving Assertions about Programs, SIGPLAN Notices 7*, 1 (January 1972): 121–127.

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

McCabe76 McCabe, T.J. "A Complexity Measure." *IEEE Transactions on Software Engineering SE-2*, 4 (December 1976): 308–320.

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.

Mills71 Mills, H.D. *Chief Programmer Teams: Principles and Procedures* (Report No. FSC 71-5108). IBM Federal Systems Division, 1971.

Myers79 Myers, G.J. *The Art of Software Testing.* New York, NY: Wiley-Interscience, 1979.

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.

Nix88 Nix, C.J. & Collins, B.P. "The Use of Software Engineering, Including the Z Notation, in the Development of CICS." *Quality Assurance 14*, (September 1988): 103–110.

Osterweil76 Osterweil, L.J. & Fosdick, L.D. "DAVE—A Validation, Error Detection, and Documentation System for FORTRAN Programs." *Software—Practice and Experience 6*, (1976): 474–486.

Osterweil87 Osterweil, L.J. "Software Processes are Software Too," pp. 2–13. *Proceedings of the Ninth International Conference on Software Engineering.* Monterey, CA, March 1987.

Parnas72 Parnas, D.L. "On the Criteria to be Used in Decomposing Systems into Modules." *Communications of the ACM 15*, 2 (December 1972): 1053–1058.

Parnas78 Parnas, D.L. "Designing Software for Ease of Extension and Contraction," *Proceedings of the Third International Conference on Software Engineering.* Atlanta, GA, May 1978.

Parnas79 Parnas, D.L. "Designing Software for Ease of Extension and Contraction." *IEEE Transactions on Software Engineering SE-5*, 2 (March 1979): 128–138.

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

Penedo85 Penedo, M.H. & Stuckle, E.D. "PMDB — A Project Master Database for Software Engineering Environments," pp. 150–157. *Proceedings of the Eighth International Conference on Software Engineering.* London, August 1985.

Perry87 Perry, D.E. "Software Interconnection Models," pp. 61–69. *Proceedings of the Ninth International Conference on Software Engineering.* Monterey, CA, March 1987.

Rapps85 Rapps, S. & Weyuker, E.J. "Selecting Software Test Data Using Data Flow Information." *IEEE Transactions on Software Engineering SE-11*, 4 (April 1985): 367–375.

Reiss90 Reiss, S. "Connecting Tools Using Message Passing in the Field Environment." *IEEE Software 7*, 4 (July 1990): 57–66.

Ritchie74 Ritchie, D.M. & Thompson, K.L. "The UNIX Time-Sharing System." *Communications of ACM 17*, 7 (July 1974).

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

Rochkind75a Rochkind, M.J. "The Source Code Control System." *IEEE Transactions on Software Engineering SE-1*, 4 (December 1975): 364–370.

Rochkind75b Rochkind, M.J. "The Source Code Control System," *Proceedings of the First International Conference on Software Engineering.* Washington, DC, September 1975.

Royce70 Royce, W.W. "Managing the Development of Large Software Systems: Concepts and Techniques," *Proceedings of WesCon*, August 1970.

Sackman68 Sackman, H.; Erickson, W.J.; & 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.

Selby87 Selby, R.W. Jr.; Basili, V.R.; & Baker, F.T. "Cleanroom Software Development: An Empirical Evaluation." *IEEE Transactions on Software Engineering SE-13*, 9 (September 1987): 1027–1037.

Shneiderman84 Shneiderman, B. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Reading, MA: Addison-Wesley, 1984.

Spivey89 Spivey, J.M. "An Introduction to Z and Formal Specification." *Software Engineering Journal 4*, 1 (January 1989): 40–50.

Stallman81 Stallman, R.M. "EMACS: The Extensible, Customizable, Self-Documenting Display Editor," pp. 147–156. *Proceedings of the ACM*

SIGPLAN/SIGOA Symposium on Text Manipulation. Portland, OR, June 1981.

Stevens74 Stevens, W.P.; Myers, G.F.; & Constantine, L.L. "Structured Design." *IBM Systems Journal 13*, 2 (1974): 115–139.

Swartout82 Swartout, W. & Balzer, R. "The Inevitable Intertwining of Specification and Implementation." *Communications of the ACM 25*, 7 (July 1982): 438–440.

Teichroew77 Teichroew, D. & Hershey, E.A. III "PSL/PSA: A Computer-Aided Technique for Structure Documentation and Analysis of Information Processing Systems." *IEEE Transactions on Software Engineering SE-3*, 1 (January 1977): 41–48.

Teitelman81 Teitelman, W. & Masinter, L. "The Interlisp Programming Environment." *Computer 14*, 4 (April 1981): 25–34.

Tichy79 Tichy, W. "Software Development Based on Module Interconnection," pp. 29–41. *Proceedings of the Fourth International Conference on Software Engineering*. Munich, FRG, September 1979.

Tichy81 Tichy, W. *Revision Control System* (Distributed Software). Lafayette, IN: Purdue University, 1981.

Tichy85 Tichy, W. "RCS—A System for Version Control." *Software—Practice and Experience 15*, 7 (July 1985): 637–654.

Turski81 Turski, W.M. "Specification as a Theory with Models in the Computer World and in the Real World." *Infotech State of the Art Report 9*, 6 (1981): 363–377.

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

Weiser82 Weiser, M. "Programmers Use Slices When Debugging." *Communications of the ACM 25*, 7 (July 1982): 446–452.

Weyuker88 Weyuker, E.J. "An Empirical Study of the Complexity of Data Flow Testing," pp. 118–195. *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*. Banff, Alberta, CANADA, July 1988.

Wirth71 Wirth, N. "Program Development by Stepwise Refinement." *Communications of the ACM 14*, 4 (April 1971): 221–227.

Wulf76a Wulf, W.A.; London, R.L.; & Shaw, M. "An Introduction to the Construction and Verification of Alphard Programs." *IEEE Transactions on Software Engineering SE-2*, 4 (December 1976): 253–265.

Wulf76b Wulf, W.A.; London, R.L.; & Shaw, M. "An Introduction to the Construction and Verification of Alphard Programs," p. 390. *Proceedings of the Second International Conference on Software Engineering.* San Francisco, CA, October 1976.