

Software Engineering Institute

Technical Report

ESD-TR-86-213

SEI-86-TM-6

May 1986

Beyond Programming-in-the-Large: The Next Challenges for Software Engineering

Mary Shaw

Approved for public release. Distribution unlimited.

Carnegie Mellon University

Pittsburgh, Pennsylvania 15213

This work was sponsored by the Department of Defense.

Copyright © 1986. Mary Shaw

This technical report was prepared for the


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

The ideas and findings in this report 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 report has been reviewed and is approved for publication.

FOR THE COMMANDER



Karl H. Shingler
SEI Joint Program Office

This document is available through the Defense Technical Information Center. Dtic provides access to and transfer of scientific and technical information for DoD personnel, DoD contractors and potential contractors, and other U.S. Government agency personnel and their contractors. To obtain a copy, please contact DTIC directly: Defense Technical Information Center, Attn: FDRA, Cameron Station, Alexandria, VA 22304-6145.

Copies of this document are also available through the National Technical Information Services. For information on ordering, please contact NTIS directly: National Technical Information Services, U.S. Department of Commerce, Springfield, VA 22161.

Beyond Programming-in-the-Large: The Next Challenges for Software Engineering

Mary Shaw

Abstract. As society's dependence on computing broadens, software engineering is being called upon to address new problems that raise new technical and nontechnical concerns. Aspirations and expectations for the application of computers appear to be unbounded, but present software development and support techniques will not be adequate to build computational systems that satisfy our expectations, even at very high cost. Each order-of-magnitude increase in the scale of the problems being solved leads to a new set of critical problems that require essentially new solutions. The next challenges for software engineering will deal with software as one of many elements in complex systems, which we call *program-as-component*, and with the role of software as an active participant in the software development process, which we call *program-as-deputy*.

Software engineering is concerned with finding practical solutions to computational problems. Over the next few years, software engineering will be required to

- respond to society's needs for software in constantly widening application areas
- accommodate constantly increasing levels of expectation for software capability and performance
- gain intellectual control over software development and support

The major challenges that arise from these requirements will be to broaden substantially software engineering's traditional scope of attention and to increase by orders of magnitude the scale of systems that can be constructed successfully. This will require significant changes in the character of the problems that we work on and the methods that we use to solve these problems.

As software engineering has matured, the range of tasks for which computers are useful has widened dramatically. Constantly increasing numbers of people have become computer users. The unprecedented utility of computing has caused demand to escalate beyond our capacity to produce the software to satisfy that demand. Software engineering must expand both the scope of problems it can solve and the scale of the systems it can develop. To do this, software engineers must become more aggressive about propagating new technology within the field and about adopting techniques from other fields.

This paper begins with a discussion of the effects of problem scale on software engineering. It defends this proposition with surveys of the kinds of issues now confronting software developers and of the events that are expanding the scope and scale of computing needs. It argues that software engineering must move from a labor-intensive basis to a technology-intensive basis rooted in sound models and theories. The paper concludes by proposing two new sets of issues that must be dealt with, *program-as-component* and *program-as-deputy*.

1. Effects of Scale on Software Engineering

Software engineering has progressed from solving small, simple problems to solving large, quite complex ones. Moreover, as the problem scale has increased, the essential character of the problems has changed. The tasks to be accomplished have become qualitatively more difficult, as seen for example in the progress from standalone computing, to multiprogramming, to timesharing, to distributed systems. At each stage in this history, the attention of the software engineering community has been directed toward some set of issues that can be understood as characteristic of the major issues of software development at that particular time. Each new generation of systems has been more ambitious than the previous, and new problems emerge as a consequence of this increase in scale. A significant increase in system scale and corresponding shift in the character of the critical problems seems to take place roughly every decade.

Each time there is an order-of-magnitude increase in the complexity of software systems, some different aspect of system development becomes the intellectual bottleneck. In the 1950's to mid-1960's the problem was writing understandable programs, and the solution was implemented through high-level languages. In the 1970's the problem was organizing large software system development, and the solution was implemented through tools for programming-in-the-large. When a shift of bottleneck takes place, the problems encountered with smaller systems remain, but the new bottleneck forces the field to attend to a new set of problems in a fashion that may be essentially different from the way we thought about previous problems. The earlier, smaller problems don't disappear, however; they usually remain as subproblems in the larger systems.

A number of different problems have held center stage in software engineering. In each period, however, the primary emphasis of the field has been shaped by a set of issues that arise in the most ambitious software that was ordinarily being developed at the time. Frequently, these issues had arisen in earlier systems; however, in systems that clearly press the limits of software engineering, the essentially different character of the resulting problems is often not recognized or dealt with, and the system developers cope with the problems on an *ad hoc* basis. It is when the problems begin to impede system development regularly that they are distinguished as arising from issues worthy of study in their own right.

A very early shift in the driving problems of software engineering took place before the term "software engineering" was coined. In the late 1950's and early 1960's, it was often a triumph simply to write a program that successfully computed the desired result. There was little widespread systematic understanding of program organization or of ways to reason about programs. In the mid-1960's, programming was influenced substantially by the recognition that programs could be the subject of precise, even formal reasoning. By establishing that algorithms and data structures could be designed and analyzed independent of their instantiation in any particular program and, indeed, independent of each other, Knuth established algorithms and data structures as fields of study [Knuth 68]. Dijkstra further refined our view of programming by arguing that we must simplify our programs in order to understand them [Dijkstra 68]. We might describe the resulting shift from *ad hoc* to systematic programming as a change from *programming-any-which-way* to *programming-in-the-small*.

The most familiar example of a mode shift driven by increases in scale is the shift from programming to software system management that took place in the mid-1970's. This involved the recognition that constructing large, complex systems is not at all the same task as writing small individual programs – not even when the individual programs happen to require a large number of lines of code. Development of large systems requires the coordination of many people, maintenance and control of many versions, and remanufacture of old versions after the system has evolved. The problems associated with this shift had occurred in some systems many years earlier, and many of the problems were named in the conference that established the field of software engineering [Naur 68]. However, concrete prototypes of solutions did not become a significant focus of the field until a bit later. In the early 1970's, Parnas discussed techniques for modular decomposition [Parnas 72], and Baker investigated ways to organize teams of programmers [Baker 72]. DeRemer and Kron [DeRemer 76] addressed the task of describing large system structures and the essential differences between this task and the task of describing programs. They coined the terms *programming-in-the-small* and *programming-in-the-large* to identify the shift of attention from the problems encountered by a few people writing simple programs to the problems encountered by large groups of people constructing (and managing the construction of) large assemblies of modules. The significance of the distinction is that it is necessary to think about these two kinds of problems in essentially different ways, and DeRemer and Kron's contribution was to focus the attention of a significant fraction of the software engineering community on that new problem.

To explain the nature of this change, we can compare the shifts in several attributes of the problems and activities of programming-in-the-small and programming-in-the-large:

- **Characteristic problems:** The major focus shifted from emphasis on particular algorithms to emphasis on interfaces, system structures, and management of the people involved in system production.
- **Dominant data issues:** The chief concern about data shifted from data structures and data types to databases whose lifetimes transcend the execution of particular programs.
- **Dominant control issues:** The predominant view about flow of control shifted from the view that programs execute once and terminate to a view of an assembly of computational modules that are expected to execute continually.
- **Specification issues:** The shift in control issues led to a change in specification concerns. Whereas terminating programs can be specified as mathematical functions, the specification issue in a continually executing system deals with the sequence of states through which the system passes and the side effects of those states.
- **Character of state space:** The state space of a piece of software shifted from a small, easily comprehensible state space to a large state space with complex structure.
- **Management focus:** The management unit shifted from the individual effort to team efforts directed toward developing and maintaining large systems.
- **Tools and Methods:** Whereas the programmer-in-the-small uses data structures, compilers, linkers and loaders, the tools of the programmer-in-the-large are program-

ming environments, integrated tools, version control, configuration management, document production, and report generation.

One way we deal with increases in problem size and complexity is by finding ways to reduce the apparent complexity of the problem. For example, higher-level languages reduce the number of lines of program text (the apparent complexity) required to achieve a given functional capability (the actual complexity). However, there are limits to our ability to deal with increased complexity by abstracting existing methods and extending existing solutions. On occasion, we must focus on a different set of issues.

An order-of-magnitude increase in problem size sometimes escalates a new aspect of the software development process to the status of a bottleneck. The phenomenon of issue shift with increased scale is generally one of emphasis -- of which issues are the critical ones -- rather than one of discovering essentially new issues in the software development process. However, the newly-important issues usually have not been explored thoroughly, and new solutions are required. In these cases, as well in the cases when familiar aspects grow complex, the fruitful approaches are the introduction of systematic methods, the automation of routine detail, and the reduction of apparent complexity.

When solutions to these issues are found, they often turn out to be useful for problems of smaller size as well. For example, many individuals make good use of version-control tools on one-person projects. Even if the new tools are not critical for solution of the smaller problems, they may nevertheless be extremely useful.

2. The Nature of the Software Problem

Concerns about problems in the software development and support process have increased over the past several years, perhaps because increases in the size and complexity of software systems have made those problems more apparent. Today, software is a dominant factor in fielding advanced technology systems, and software reliability is a crucial issue. As a result of those concerns, numerous studies have highlighted critical software problems. In general, the problems are related to cost, management, and performance. The problems can be attributed to some characteristics of the state of the practice: labor-intensiveness and inadequate use of available technology. They are further compounded by the sheer magnitude of the applications.

There is a real temptation to single out a single issue as "The Software Problem" and marshal resources to attack that problem. However, that view is too simplistic. The problems associated with software are economic, managerial, and technical; they involve production, maintenance, and use of systems. This section examines several facets of this complex of software problems.

Software can be a critical factor in large system development and operation. Simple software errors can cause expensive failures in large complex systems, and delegation of system control to software may lead to failure when unexpected conditions or interactions arise. An example of the first kind of failure caused Gemini V to splash down 100 miles off course [Fox 83].

In essence, the programmer confused sidereal time with solar time and assumed that any location on earth would return to the same position relative to the sun every 24 hours. Since this ignores orbital motion, the accumulated error led to an off-course landing. An example of the second kind of failure was the complete loss of power in a Boeing 767 in August 1983 [AP 83]. A computer-controlled descent to the landing airport was optimized for fuel efficiency, but the low power levels allowed ice to build up in the engine and block airflow necessary for engine cooling. As a result, the engines overheated and had to be shut down. (The engines were restarted safely for landing.) A second example of failure due to unexpected interactions took place at the Crystal River nuclear power plant in February 1980 [Marshall 80]. For unknown reasons, a short circuit led to erroneous indications of low temperature in the reactor. The automated controls responded by speeding up the reaction in the core. As a consequence, the reactor overheated, the pressure in the core increased to the danger level, and the reactor automatically shut down. To release the pressure, the computer opened a pressure relief valve, but pressure dropped so quickly that a high pressure injection system was automatically activated, which flooded the coolant loop. The operator prevented further damage by closing appropriate valves manually.

Software is an ever-increasing part of the computing problem. A growing percentage of computer-related costs is attributable to software, and both the total cost and the fraction attributable to software are expected to increase dramatically. For example, Figure 1 shows a projection of costs of computing involving software for mission-critical applications in the US Department of Defense [EIA 80].

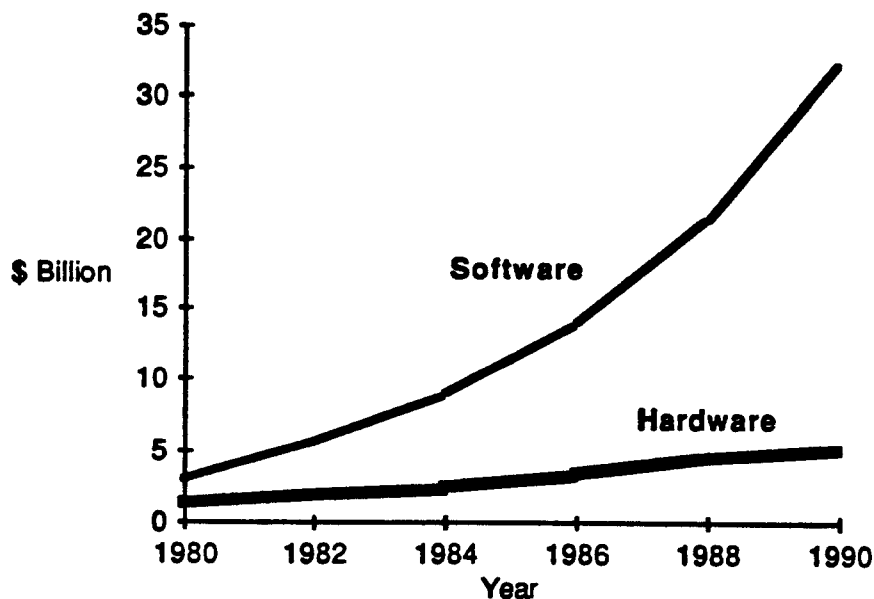


Figure 1: Projected Growth of Software and Hardware Costs

Computing technology is only one dimension of the problem. As system complexity has increased, managerial and professional issues have assumed increasing importance. These are coming to be widely recognized and addressed, but they have not yet achieved the same stature

as the technological issues. In addition, economic and legal considerations are playing an increasing role as the value of software as intellectual property grows.

Qualified personnel are scarce and productivity growth is low. Software costs for both development and maintenance are still largely labor-related, yet the supply of computer professionals is insufficient to meet the demand, and the supply and productivity of existing professionals are not increasing fast enough. Figure 2 compares the demand for software in one application area [Tomayko 85] with the overall productivity growth of programmers [Boehm 81]. Software maintenance activities are particularly understaffed, resulting in degraded operational support. Similarly, qualified personnel for management functions are scarce.

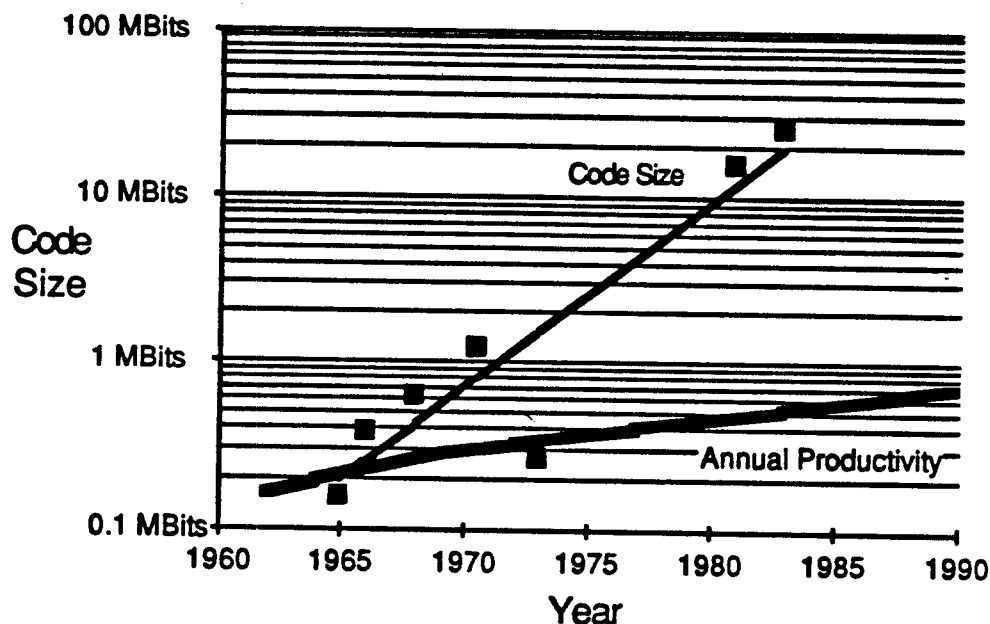


Figure 2: Onboard Code Size for Manned Spacecraft vs Annual Programmer Productivity

Current technologies may not scale up to new problems. The major concern about a software system is often whether the system can be implemented at all, not the cost of implementation. Software engineering is continually developing tools to aid in the software development process. Some of the most important tools of current interest address the management of large and changing configurations of software. These tools are of some interest for systems of all sizes (programming-in-the-small), but they become critical only when system size reaches some critical threshold (programming-in-the-large). Large-scale systems already stretch our ability to achieve desired results. Since our aspirations grow faster than our productivity, we continually set out to build systems that are more complex than any we have built before; these can be heterogeneous real-time systems in which the software is a minor component interacting with electronic or electromechanical components. As a result, we can expect to need new tools for reducing the apparent complexity of very large systems; it is reasonable to expect that these tools will be useful but not critical for systems of sizes we can manage now.

Techniques for managing projects are inadequate. The software development process is not as well understood as other kinds of development efforts, and it is therefore more difficult to plan, schedule, and manage. A major problem is a lack of good quantitative information and models for interpreting it. In some cases, the management plans are not sufficiently detailed to account for the intricacies of the development process, do not provide a sound basis for management of the development effort, or are based on poorly defined requirements. Although considerable progress has been made in the last decade, techniques still in widespread use do not cope well with changes in requirements and specifications, which are the most significant drivers of cost and schedule growth. Figure 3 illustrates the kind of problem that frequently arose under these older management strategies. It compares the history of the estimates of total project cost (black diamonds) to the history of total cost to date (white circles) for a large software project; values are normalized to total project cost. This example suggests that initial estimates are unrealistic and that estimates are not updated until reality sets in, as apparently happened in months 9 and 24 of this example [Devenney 76].

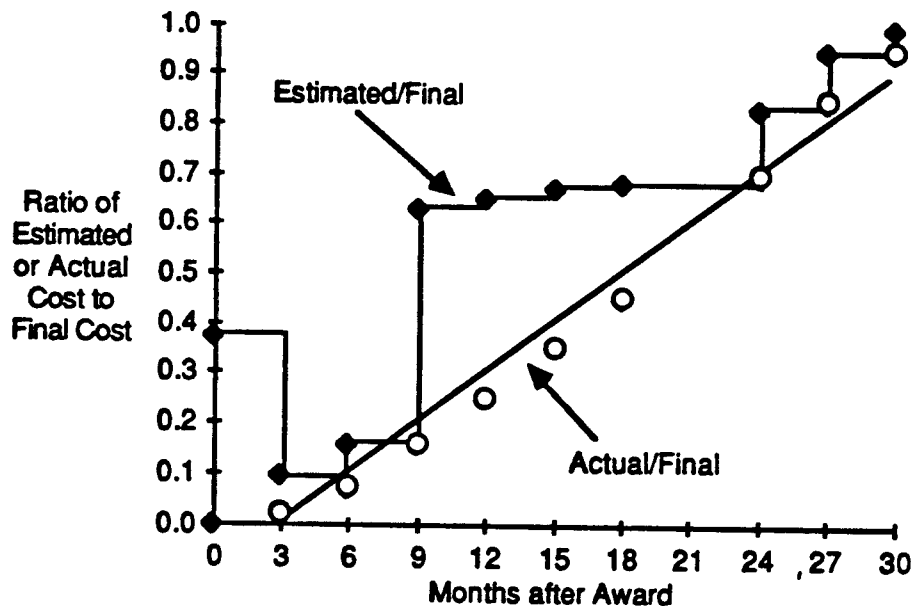


Figure 3: Growth of Estimated Total Project Cost and of Actual Accrued Costs (as fractions of final cost)

Change is a fact of life. Because the underlying technology is evolving rapidly, software tools will change continually. So will the underlying paradigms for software development. The solutions of software engineering must make allowances for change, including the ability to upgrade systems regularly and to tolerate modest amounts of inconsistency as the upgrades take place. A more serious problem is that the rapidity of change means that we are always on the leading edge of the learning curve for the current technology. By the time we have assimilated the technology and found ways to exploit it, new technology has introduced new problems.

New technology is adopted slowly. Even though software engineering is evolving rapidly,

technology transition still takes up to two decades from concept formation to widespread practical use [Redwine 84]. Reliability and integrity of systems are often limited by continued use of decade-old technology, and systems are often obsolete before they are released. In many organizations, development and support tools are insufficient, out-of-date, inefficient, and often mutually incompatible; reuse of existing code is not practiced sufficiently; and lack of standardization leads to unmanageable systems. Observed rates of technology transition in software engineering are rapid compared to some other fields, but there is considerable room for improvement. Figure 4 shows the rate at which some familiar software engineering techniques have moved toward practice [data from Redwine 84].

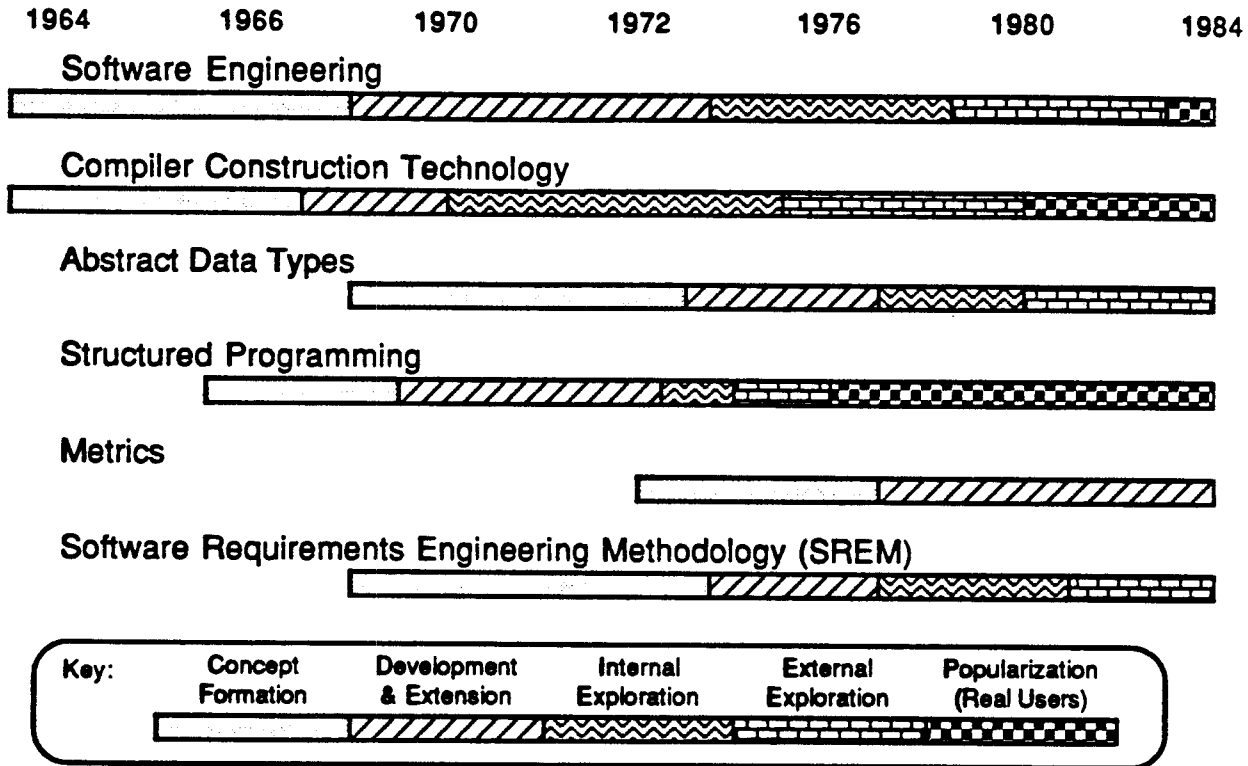


Figure 4: Progress of transition for some software technologies

Software is often the pacing component of an integrated system. Software is often embedded in a large heterogeneous system. It is frequently on the critical path of such systems, and software slippage translates directly into delays and compromises in release of the system. Moreover, major capital outlays may be required for the non-software components; in such cases, the direct cost of software delay is dwarfed by the cost of maintaining unusable hardware.

Intellectual control over software development is a central issue. Modern software is extremely complex, and its complexity rises as aspirations for system performance rise. Systems of the scale required over the next decade will be created successfully only if the software structure, the development process, and the maintenance process are understood in a precise and systematic fashion.

In summary, there is no single software problem; rather there are many problems that combine to form a large, complex set of issues. These issues span development and support tools, management techniques, strategies for buying and selling software, and availability of qualified software professionals.

3. Broadening Scope of Computing Systems

The nature of computing, and hence of software engineering, is changing rapidly. In order for software engineering to be prepared to address the problems of the 1990's, we must try to understand the forces that are shaping the field and to anticipate the roles that computers and software will play in the future. This section points out some of the trends that will affect the field over the next decade and describes some of the new phenomena and issues that may arise.

Computers are becoming smaller and cheaper, and they are being distributed across a wider and more varied population. Important current trends include:

- Decreasing hardware costs
- Increasing share of computing costs attributable to software
- Increasing expectations about the power and reliability of application systems
- Increasing range of applications, particularly those on which lives will depend
- Increasing use of software as component in integrated heterogeneous systems
- Increasing development of distributed computing and convenient network access
- Widening view of computers as an information utility and a basis for electronic publishing
- Increasing quality of interfaces to humans (voice, high-performance graphics)
- Increasing availability of computing power, especially in homes
- Increasing exposure of naive people to computers, both at home and at work
- Increasing general reliance on computers for routine day-to-day operations
- Continuing or increasing shortage of qualified professionals
- Increasing importance of "intelligent" systems requiring complex judgemental decisions
- Increasing legal and economic concerns about proper treatment of software

On the basis of these trends, we can extrapolate some future developments:

- **Pervasive Consumer Computing:** Computers will be extremely widespread, both as multiple-purpose machines in homes and offices and as dedicated machines for applications such as environment control and hazard monitoring. Most of the users of these machines will be naive--certainly the majority of them will not be programmers. As a result, most of the users of programs will not be creators of programs.
- **Information Utility:** We will come to think of computers primarily as tools for communicating and for accessing information, rather than primarily as calculating machines. Networks will provide a medium for making available numerous public

databases, both passive (catalogs, library facilities, newspapers, bulletin boards) and active (newsletters, electronic mail, individualized entertainment). Real-time control applications will become more prevalent.

- **Broad Range of Applications:** The range of applications will continue to broaden, and an increasing number will be applications in which unreliable software could lead to risk of human life. As a result of this and widespread use by nonprogrammers, much of the software will provide packaged services that require little, if any, programming. There will be substantial economic incentives for producing general systems that can be tuned to individual, possibly idiosyncratic, requirements.
- **Changes in the Workplace:** Distributed systems and networks will facilitate a distributed workplace, but we doubt that the norm for office workers will be to work at home instead of in an office. Electronic communication will speed communication, but computers will not replace human interaction for decision-making. Electronic workstations will change the nature of work that now depends on paper flow, and robotics will change manufacturing substantially.
- **Massively More Complex Computers:** Some computer networks and large computers will be replaced by or evolve into massive computer systems whose capacity is orders of magnitude greater than that of any system now available. Such systems already are emerging: current systems include very large databases (nationwide banking records, interactive consumer catalogs, traffic information for ocean-going vessels, etc.). The first steps have already been taken by airline and hotel reservation systems. Eventually, these systems may be used by millions of people simultaneously.
- **Intelligent Systems:** Intelligent software systems will provide intellectual multipliers that substantially increase professional productivity in some areas. Intelligent robots will take over an increasing percentage of the industrial workload and perhaps even make a dent in the household chores. Increasingly sophisticated systems will lessen the need for programmers, and they may increase everyone's need for a basic understanding of computers. Further, today's expert systems may be tomorrow's oracles.
- **Effect on GNP:** The fraction of the GNP represented by computing and information handling--already large--will increase as our society becomes as dependent on information as on grain or metal.

Even if this projection is inaccurate in its details, there will nevertheless be a substantial qualitative shift in the role of computers in the world at large. This view of the future raises a number of issues:

- **Consumer Concerns:** The use of computers by large numbers of nontechnical people, together with the increasing number of sensitive applications that involve computers, will raise issues about the responsibilities of vendors towards their products. These will certainly include analogs of the familiar problems associated with product and professional liability, merchantability and warrantability (guarantees), usability and reliability, licensing, copyrights, and product safety (e.g., development of an analog to the certification that Underwriters Laboratories provides for electrical products). Other problems, such as security and privacy concerns, undoubtedly will arise from the special nature of computers.
- **Production and Distribution:** An expanding role for computers and computer-related products and services in the retail marketplace will introduce new problems in manufacturing, sales and service, equitable methods of charging for shared

resources, and industry compatibility standards. Another class of problems will center on how to create software for a mass market, perhaps including some notion of mass production of software (e.g., by tailoring packages rather than by writing code and by using software to construct software).

- **Safety and Security:** In addition to the consumer-safety issues, we can expect questions concerning licensing, product and professional liability, and the trustworthiness of integrity of data provided via public databases. Existing concerns about security and privacy will increase. These concerns will be particularly acute when human life is at stake.
- **Economic Impact:** The economic impact of these major innovations will be widespread. Of particular concern for the computing industry will be the interplay between technological development and limiting factors, such as productivity, on the growth of the information sector. Accurate software cost estimates and well-considered marketing policies will be vital as the computer industry matures. Some of the most important economic changes will involve personnel, especially when unskilled positions are eliminated through automation or replaced by jobs requiring a high level of technical expertise.
- **Human Issues:** Currently, people deal directly with computers primarily by choice. As computers become pervasive, people will interact with them through necessity. There will be a variety of sociological consequences, including the necessity of systems designed for naive users, personnel dislocation caused by technical change, and major shifts in the content and style of education.
- **Social Issues:** The computer age could bring about a new underprivileged class of the computer illiterate. Women and minorities might make up the majority of this new class because of insufficient technical education. To prevent such a situation, computer scientists must be aware of the social implications of their work, and the society must be aware of the implications of this new technology.

The thrust of these examples is that software engineering must be able to respond to problems that involve naive users, highly heterogeneous systems, and increased requirements for product-level performance. We turn now to the software problems posed by these demands.

4. The Science of Software Engineering

Traditional methods of software development are *ad hoc* and labor-intensive. They will not be adequate to satisfy the increased demands on computing systems and the complexity of the resulting systems. Software engineering must move to a technology-intensive basis that draws on scientifically-based models and theories; it must be prepared to take advantage of advances in these areas as they become available. The education of software engineers is critical to this progress, for good ideas achieve practical utility only in the hands of people who use them wisely.

Over the past two decades this kind of shift in technology has taken place in many aspects of programming-in-the-small. Some of the earliest formal models supported the analysis of algorithms. Our understanding of algorithms for certain problem domains is now well structured, we can analyze the performance of specific algorithms, and we know theoretical limits on performance in many cases. Similarly, a theory to support abstract data types emerged during the

1970's. In the late 1960's computer scientists recognized the importance of good representations and their associated data structures. Refining this insight to derive a theory of abstract data types took about a decade; it required advances in formal specification, programming languages, verification, and programming methodology. Undergraduate computer science students should now routinely master algorithmic analysis and abstract data types; it is now reasonable but not entirely realistic to expect the material to be applied in routine practice.

Sound theories can also contribute significantly to our ability to construct software systems. For example, the compiler for a programming language is a medium-sized system with a structure that is now well understood. Whereas in the early 1960's the construction of a compiler was a significant achievement, compilers now often are constructed routinely. Good theoretical understanding of syntax led to effective techniques for constructing parsers, first manually and more recently automatically. Similarly, good theories for programming language semantics and type structures are leading to automation of other stages of compiler construction.

Although programming-in-the-large has a somewhat shorter history, formal models are beginning to emerge for the information management problems in that domain. For example, configuration management and version control began on an *ad hoc* basis with simple tools for organized (and often massive) recompilation, but at least a few models of system configuration and remanufacture are guiding the construction of software tools. The theoretical basis not only shows how to manage dependency information to reconstruct a system correctly, it also supports more efficient strategies of system reconstruction by avoiding unnecessary steps (e.g., recompilation of modules in which the only changes were comments or which depend only on unchanged portions of modules that were changed).

These examples give the flavor of the progress toward sound foundations for software engineering. There are clearly many areas in which the models, theories, and methodologies are still primitive. However, the power of soundly based theories in at least a few areas offers encouragement for developing and refining theories in other areas.

5. The Next Challenges

In the decade since software engineering recognized programming-in-the-large as a significant issue, the complexity of software systems has grown by another leap, and another shift is now taking place. Software engineers must now deal with complex systems in which software is one of many components in a large heterogeneous system and in which the software is expected to serve as a surrogate for a human programmer, taking an active role in the development and control of software systems. We will describe those new modes of operation as *program-as-component* and *program-as-deputy*, respectively. Their relation to programming-in-the-small and programming-in-the-large, as well as to the *ad hoc* programming of the 1960s, is suggested by Figure 5.

Identification of these new modes recognizes a change in the character of the problems that depend on computational solutions as well as a change in the character of the software development and support process:

Attribute	1960+5 years <i>Programming- any-which-way</i>	1970+5 years <i>Programming- in-the-small</i>	1980+5 years <i>Programming- in-the-large</i>	1990+5 years <i>Program-as- component</i>	1990+5 years <i>Program-as- deputy</i>
Characteristic Problems	Small programs	Algorithms and programming	Interfaces, management, system structures	Integration of heterogeneous components	Incorporation of judgement
Data Issues	Representing structure and symbolic information	Data structures and types	Long-lived data bases, symbolic as well as numeric	Integrated data bases, physical as well as symbolic	Knowledge representation
Control Issues	Elementary understanding of control flow	Programs execute once and terminate	Program assemblies execute continually	Control over complex physical systems	Programs learn from own behavior
Specification Issues	Mnemonics, precise use of prose	Simple input-output specifications	Systems with complex specifications	Software as component of heterogeneous system	Extensive reuse of design
State Space	State not well understood apart from control	Small, simple state space	Large structured state space	Very large state with dynamic structure and physical form	State includes development as well as application
Management Focus	None	Individual effort	Team efforts, system lifetime maintenance	Coordination of integration and interactions	Knowledge about application domain and development
Tools and Methods	Assemblers, core dumps	Programming languages, compilers, linkers, loaders	Environments, integrated tools, documents	Tools for real-time control, dynamic reconfiguration	Program generators, expert systems, learning systems

Figure 5: Emergence of Software Problems with Growth in System Complexity

- They are not necessarily amenable to algorithmic solution, and heuristic approaches may be important.
- They involve judgemental elements such as selecting among competing, non-absolute preferences.
- They depend on problem-specific knowledge that must be consulted dynamically.
- They are so complex that solutions cannot be specified a priori but must be evolved through experience.
- They involve integration of a heterogeneous set of system components including hardware as well as software.
- They require graceful accommodation of unreliable data and other vagaries of physical systems.
- They may involve external constraints that arise from the physical system being controlled rather than from the logical function of the system.

The role of *program-as-component* arises in large heterogeneous systems. Such systems include programs in multiple languages for significantly complex hardware systems; they may have mechanical constraints, produce noisy data, or impose real-time constraints on operation. To capture the nature of this shift of attention, we can consider the same attributes as before:

- **Characteristic problems:** The major focus of design is shifting from algorithms and interfaces to the integration of the system as a whole.
- **Dominant data issues:** We need integrated data bases that include not only symbolic and numeric information but also information about the physical status of the system that may in fact be a physically distributed system – in which communication is a very significant issue.
- **Dominant control issues:** Software must now provide control over complex systems that may include data subject to physical or mechanical constraints as well as the usual purely symbolic data.
- **Specification issues:** Software specifications must address interfaces with non-software elements of the system as well as with other software elements.
- **Character of state space:** The state space of a large heterogeneous system may be very large. In addition, the structure may be dynamically reconfigured, and it may contain physical elements as well as symbolic elements.
- **Management focus:** The heterogeneous character of these systems increases demands on management to coordinate design, development, construction, and integration schedules which have very different characters.
- **Tools and Methods:** These systems require real-time control and interfaces for lay users; they must be capable of running complex control problems with very little human intervention.

The role of *program-as-deputy* arises when large, creative portions of the program development process are delegated to software. This shift has been taking place gradually ever since the first symbolic assembler assigned addresses to variables. As time has passed, more and more expertise about the software development process has been incorporated in programs that perform increasingly creative subtasks within the software development and management process. By

delegating these subtasks to tools or program generators, we will raise the level of software reusability from code fragments to design elements. The attributes of this activity are:

- **Characteristic problems:** The major focus is on incorporating expertise and judgement in software tools. The shift that makes this an issue now is the attempt to incorporate into the automated tools judgements that may be partially subjective.
- **Dominant data issues:** Since software is now automatically carrying out some aspects of the software development process, we must represent not only the data of the application domain but also knowledge about that domain's specific expertise and the state of the software process. Further, we must choose representations that support learning -- retaining information about specific developments and using it to improve the overall performance of the tools.
- **Dominant control issues:** Programs must not only encode expertise but must also learn from their own prior use.
- **Specification issues:** Emphasis has shifted from reuse of code to reuse of design through automation.
- **Character of state space:** We must extend the state space to cover the development processes as well as the application domain.
- **Management focus:** Both qualitative and quantitative knowledge about the software development process and about the application domain must be acquired and managed.
- **Tools and Methods:** Program generators are significant early tools. As time passes, they are being joined by learning systems and knowledge representation systems.

These shifts reflect only the changes in the technology of software development and support. As system scale has increased, issues from several other areas also have become critical.

- **Professional Issues:** Software engineering will experience a significant personnel shortfall for at least the next 5-10 years. Attention to education, career paths, and professionalism will help to take up the slack.
- **Legal Issues:** Software is unlike either hard products or books. As a result, neither patent law nor copyright law is quite appropriate for software products and tools. Intellectual property law for software must deal with such issues as software protection, product liability, impediments to dissemination of new technology, and rights in technical data.
- **Economic Issues:** Costs of software development arise from many sources, and software consumes an increasing fraction of corporate resources. Software engineers often fail to appreciate cost components other than the ones directly associated with creating the software. The public marketplace rewards software contributions imperfectly, especially in cases where software must be modified for reuse. In addition, accounting rules for software influence corporate decisions about innovation.
- **Managerial Issues:** Management concerns have interacted with software technology ever since we recognized the issues of programming-in-the-large. As systems grow larger, managerial issues expand to include improved costing and estimating techniques, the visibility into software development necessary for effective control, adequate performance measures for human organizations, and incentives and risk reduction measures to encourage more productive software technology.

Although these areas generally have not been covered in software engineering education, their role now requires attention.

Conclusions

The issues and solutions discussed here cover an area somewhat larger than the one software engineering traditionally encompasses. They include many of the problems of end users, such as consumer-level product quality and systems in which software is an integral (and dedicated) component of a larger system. They also include material that usually is considered part of other parts of computer science, such as artificial intelligence, and computer architecture. For software engineering to respond to the problems it will confront over the next decade, it must cope with problems of

- **Scale:** Both the magnitude and the complexity of our systems will continue to increase, so we must seek ways to make the software a participant in managing its own development
- **Scope:** Computing will play an ever-broader role in specific domains, so we must learn to deal with software as a component of integrated heterogeneous systems

Software engineering should be broad enough to address the software design and integration issues and the software development and maintenance elements of those problems, including both technological and non-technological elements. It must be broad enough to shift focus to new issues, such as program-as-component and program-as-deputy, as the field grows and technical bottlenecks shift.

Acknowledgements

My understanding of software engineering has come from many discussions with other computer scientists, especially my colleagues at Carnegie-Mellon. Particular insights in this paper came from discussions with Bill Wulf, Allen Newell, Nico Habermann, and Jim Horning.

In addition to the direct citations, the following served as major sources: Some of the development of "The Software Problem" was prepared for [Barbacci 85]. The discussion of "Broadening Scope of Computer Systems" was originally presented in [Shaw 85]. The section on "The Science of Software Engineering" was prepared for [Shaw 86]. The author's view of the software problem was strongly influenced by the three-day retreat reported in [Musa85].

References

- [AP 83] Associated Press story in Los Angeles Times, 24 August 1983, p. 1. Reported in *Software Engineering Notes* 8, 5, October 1983.
- [Baker 72] F. T. Baker. "Chief Programmer Team Management of Production Programming." *IBM Systems Journal*, 11, 1, 1972, pp. 56-73.
- [Barbacci 85] Mario Barbacci, A. Nico Habermann, Mary Shaw. "The Software Engineering Institute: Bridging Practice and Potential." *IEEE Software*, 2, 6, November 1985, pp. 4-21.

- [Boehm 81] Barry W. Boehm. *Software Engineering Economics*. Prentice-Hall, 1981.
- [DeRemer 76] Frank DeRemer and Hans H. Kron. "Programming-in-the-Large versus Programming-in-the-Small." *IEEE Transactions on Software Engineering*, 2, 2, June 1976, pp. 80-86.
- [Devenney 76] Thomas J. Devenney. *An Exploratory Study of Software Cost Estimating at the Electronic Systems Division*. MS dissertation, Air Force Institute of Technology, July 1976 (approved for public release).
- [Dijkstra 68] Edsger Dijkstra. "GOTO Statement Considered Harmful." *Communications of the ACM*, 11, 3, March 1968, pp. 147-148.
- [Fox 83] Joseph M. Fox. *Software and Its Development*. Prentice-Hall, 1983, pp. 187-188. Reported in *Software Engineering Notes* 9, 1, January 1984.
- [EIA 80] Electronic Industries Association, Government Division. *DoD Digital Data Processing Study -- a Ten Year Forecast*. 1980.
- [Knuth 68] Donald E. Knuth. *Fundamental Algorithms. The Art of Computer Programming*, Vol. 1, Addison-Wesley 1968.
- [Marshall 80] Elliott Marshall. "NRC Takes a Second Look at Reactor Design." *Science*, 207 (28 March 1980), pp. 1445-48. Reported in *Software Engineering Notes* 10, 3 (July 1985).
- [Musa 85] John D. Musa. "Software Engineering: The Future of a Profession." *IEEE Software*, 2, 1, January 1985, pp. 55-62.
- [Naur 68] Peter Naur and Brian Randell (eds). *Software Engineering. Report on a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 October 1968*.
- [Parnas 72] David L. Parnas. "On the Criteria for Decomposing Systems into Modules." *Communications of the ACM*, 15, 12, December 1972, pp. 1053-1058.
- [Redwine 84] Samuel T. Redwine, Louise Giovane Becker, Ann B. Marmor-Squires, R. J. Martin, Sarah H. Nash, William E. Riddle. *DoD Related Software Technology Requirements, Practice, and Prospects for the Future*. Institute for Defense Analysis, IDA Paper P-1788, June 1984.
- [Shaw 85] Mary Shaw (ed). *The Carnegie-Mellon Curriculum for Undergraduate Computer Science*. Springer-Verlag, 1985.
- [Shaw 86] Mary Shaw. "Education for the Future of Software Engineering." *Proc. of Software Engineering Institute Software Engineering Education Workshop*, Springer-Verlag 1986 (to appear).
- [Tomayko 85] James Tomayko. Personal communication.

