Contents

Report of the Task Force on Military Software

Defense Science Board

September 1987

NOTE: This is a modified version of the original report. See note below. [1]

1 Executive Summary

Many previous studies have provided an abundance of valid conclusions and detailed recommendations. Most remain unimplemented. If the military software problem is real, it is not perceived as urgent. We do not attempt to prove that it is; we do recommend how to attack it if one wants to.

We do not see any single technological development in the next decade that promises ten-fold improvement in software productivity, reliability, and timeliness. There are several technical developments under way which together can be expected to yield one order of magnitude, but not two. Few fields have so large a gap between best current practice and average current practice; we concur with the priorities that DoD has given tc upgrading average practice by more vigorous technology transfer.

Current DoD initiatives in software technology and methodology include the Ada effort, the STARS program, DARPA's Strategic Computing Initiative, the Software Engineering Institute, and a planned program in the Strategic Defense Initiative. These five initiatives are uncoordinated. We recommend that the Undersecretary of Defense (Acquisition) establish a formal program coordination mechanism for them, (Rec. #2).

The big problems are not technical. In spite of the substantial technical development needed in requirements-setting, metrics and measures, tools, etc., the Task Force is convinced that today's major problems with military software development are not technical problems, but management problems. Hence we call for no new initiatives in the development of the technology, some modest shift of focus in the technology efforts under way, but major re-examination and change of attitudes, policies, and practices concerning software acquisition.

STARS

The DoD program for Software Technology for Adaptable, Reliable Systems, STARS, has made little progress in recent years and has had vague and ill-focused plans for the future. Service support and enthusiasm is lacking. Yet it is very important that such a project-independent methodology development effort proceed. We recommend that the STARS Joint Program Office be moved from the Office of the Secretary of Defense to the USAF Electronic Systems Division. (Rec. #1) We recommend that a general officer be given responsibility for STARS, the Ada Joint Program Office, and Software Engineering Tnstitute (whose contracting office is already in ESD). Deputies from the other Services should be appointed.

[1] This report is an adapted version of the original report from `https://apps.dtic.mil/dtic/tr/fulltext/u2/a188561.pdf` which has been semi-automatically converted to a computer-readable form for ease of access and reference in this modern Internet-based ecosystem. The conversion was performed by Michael Pyne. He worked with some level of care but may not have caught all transcription errors.

Ada

It is very important for DoD to have a standard programming language; Ada is by far the strongest candidate in sight. The 1983 mandate for Ada was technically premature. DoD commitment to Ada since that time has been weak. The state of Ada compiling technology is now such that it is time to commit vigorously and wholeheartedly. The directives 3405.1 and 3405.2 are right first steps - management follow-through on enforcement and support is now essential.

Ada embodies and facilitates a set of new approaches to building software, generally known as "modern software practices." We expect these practices, rather than yet another programming language, to make a real difference in software robustness, reusability, adaptability, and maintenance. Ada is not the only conceivtble vehicle for such practices, but if is here, it has been tailored for the embedded software problem, and multiple compilers have- been validated. We recommend against further waiting, language tuning, or subsetting.

Achieving the benefits of modern programming practices requires the development of unified programming environments. This work must continue to be pushed forward. Few program managers will want to take on the headaches of being first user of a new tool, yet it is essential that all major new programs be committed to that tool if it is to be effective. Only top-level DoD commitment and mandate can make that happen. We commend AJPO for its technical success in establishing the language definition and language validation procedures. We recommend that it be moved from OSD to a unified software joint program office in the USAF Electronic Systems Command (Recs. #6,7).

Acquisition

Mileu. The civilian software market has exploded in the past decade, so that the total civilian market for purchased software, not counting in-house-built application software, is now more than ten times larger than the DoD market. This requires a radical update in much DoD thinking. Some implications:

1. DoD can no longer create de facto standards and enforce them on the civilian market, as it was able to do with COBOL.

2. DoD must not diverge too far from whatever the civilian market is doing in programming methodology, else it will have to support its own methodology by itself, with little resource or training commitment from others. (The same thing is true of processor architectures.)

3. DoD should be aggressively looking for opportunities to buy, in the civilian market, tools, methods, environments, and application software. Whenever it can use these instead of custom-built software, it gets big gains in timeliness, cost, reliability, completeness of documentation, and training. But today's acquisition regulations and procedures are all heavily biased in favor of developing custom-built software for individual programs.

Life-cycle model. DoD Directive 5000.29 and STD 2167 codify the best 1975 thinking about software, including a so-called "waterfall" model calling for formal specification, then request for bids, then contracting, delivery, installation, and maintenance. In the decade since the waterfall model was developed, our discipline has come to recognize that setting the requirements is the most difficult and crucial part of the software building process, and one that requires iteration between the designers and users. In best modern practice, the early specification is embodied in a prototype, which the intended users can themselves drive in order to see the

consequences of their imaginings. Then, as the design effort begins to yield data on the cost and schedule consequences of particular specifications, the designers and the users revise the specifications.

Directive 5000.29 not only does not encourage this best modern practice, it essentially forbids it. We recommend that it be revised immediately to mandate and facilitate early prototyping before the baseline specifications are established (Rec. #23).

DoD-STD-2167 likewise needs a radical overhaul to reflect best modern practice. Draft DoD-STD-2167A is a step, but it does not go nearly far enough. As drafted, it continues to reinforce exactly the document-driven, specify-then-build approach that lies at the heart of so many DoD software problems.

For major new software builds, we recommend that competitive level-of-effort contracts be routinely let for determining specificatlons and preparing an early prototype (Rec. #26). The work of specification is so crucial, and yet its fraction of total cost is so small, that we believe duplication in this phase will save money in total program cost, and surely save time. After a converged-specification has been prepared and validated by prototyping, a single competitively-bid contract for construction is appropriate.

Incentives. Defense procurement procedures discourage contractor investment in the development of new software methodology. Any such contractor investment made today promises low return. We recommend that the DoD rights-in-data policy be revised to distinguish software rights from other rights, and that the policy as it applies to software be designed to encourage contractor investment, both with private and IRD funds, in tools, methods, and programming environments (Recs. #17-22).

Similarly, today's policies actively discourage the reuse of software modules from one system in another. We recommend a variety of policy changes, each designed to encourage reuse, and indeed, the establishment of a public market in reusable software parts (Recs. #29-33).

Personnel

It appears that the number of software-qualified military officers has been essentially constant over the past decade, despite exponential growth in software. Many studies have recommended actions that need to be taken re training, specialty codes, career paths, etc., to address the shortage of uniformed specialists. Some of these have been taken. Nevertheless, the number has not increased.

We doubt that it will. The powerful civilian demand for such persons will, we expect, continue to drain them away from the Services as fast as they reach first retirement age, or before.

Therefore we recommend that the Services now assume that there will not be more such people, and concentrate effort on how best to use those they have (Rec. #34). The application-knowledgeable, technically skilled leaders are the military's limiting resource in using today's computer technology.

We observe that in the best military software programs, the number of customer software people engaged in the acquisition and program oversight approximates 10% of the number of contractor personnel. This number does not seem too high. Few program offices are staffed so well, however, largely due to the shortage of qualified people. Meanwhile one observes some substantial software-building efforts under way within the Services, usually done by a combination of civilian and uniformed personnel, generally managed by software-qualified officers. This is a second-best use of the available specialist officers.

We recommend phasing out this practice and concentrating the available knowledgeable officers on acquisition (Rec. #35). We see no other way that the exponential growth in needed military software can be met.

2 Introduction

2.1 The Charge to the Task Force

Abbreviated Terms of Reference, (Appendix A2 contains the full text)

A. Assess and unify various recent studies.

B. Examine why software costs are high.

C. Assess STARS for military software; discuss the priority of its components.

D. Recommend how to enlist industry, Service, and university efforts in a productivity thrust.

E. Assess STARS, etc., for U.S. international competitiveness.

F. Recommend how to apply R&D funds to get the most increase in military software capability.

G. Recommend how to implement an incremental and evolutionary approach to (F).

H. Assess the wisdom of the Ada plan, especially in view of "Fourth-Generation" languages.

What the Task Force Did Not Address

Problem Seriousness Sizing. It would be presumptuous, and appear to be self-serving as well, for this Task Force to tell the Service commanders and the DoD civilian authorities that your mission-critical software problem ranks high on your present or future critical problem list. Other studies have sized the cost and recounted software-caused delays and system malfunctions. Your own experience will have to put this problem into proper perspective among all your difficulties.

What the Task Force is qualified to do for you is to

- characterize software, its problems, and its technology
- identify trends that will, in the course of time, make today's problems worse or better,
- suggest actions to address today's problems and avert tomorrow's calamities.

Non-Mission-Critical Software. The Task Force largely limited itself to mission critical systems, those wherein military software most differs from civilian-market software. Our recommendations with respect to procurement, however, apply to all DoD acquisition of software. In Section 6 we categorize DoD software according to the degree to which it must be non-standard because of its military function.

Service-Specific Personnel Problems. We did not address Service-specific personnel and skills problems. These have been adequately addressed in earlier studies. The career path and skills-retention problems continue to be very real in all the Services.

SEI. We did not review the Software Engineering Institute, other than to hear a briefing on its objectives. It was in the process of being established and finding a permanent director during our study; any review would have been premature.

SDI. The same was true of the SDI plan for developing software methodology. At the time we were briefed by the SDI office, there was no plan to review.

SCI. We had only one briefing on the DARPA software methodology efforts encompassed within the Strategic Computing Initiative. These efforts are properly aimed at producing results a decade hence. The approaches are sufficiently bold that little in the way of directly applicable short- and mid-term results can be expected.

New Technological Initiatives. We do not recommend any new initiatives or funding for new specific research or technology-development programs. We support the recent technological initiatives, but today's major unaddressed problems are not technical, but managerial.

2.2 Military Software

Role. Software plays a major role in today's weapon systems. The "smarts" of smart weapons are provided by software. Software is crucial to intelligence, communications, command, and control. Software enables computerized systemns for logistics, personnel, and finance. The chief "military software problem" is that we cannot get enough of it, soon enough, reliable enough, and cheap enough to meet the demands of weapon systems designers and users. Software provides a major component of U.S. war-fighting capability.

Growth. DoD software-intensive systems have grown exponentially, reaching an annual software expenditure level in mission-critical computer systems of about $9 billion in 1985, with projections of $30 billion annually by 1990 [Taft, 1985]. This continuing growth has strained the ability of the DoD to manage their development. Because software controls system function, deficiencies in software development affect over-all weapon system performance and cost quite out of proportion to the software cost itself.

Like Civilian Software. Military software is fundamentally like advanced civilian software, only more so. That is, the properties of real-time operational software in civilian banking, airline reservations, or process control, are the same as those of weapon-system software. Big civilian database and file systems look essentially like the military logistics, finance, and personnel software. In the operation of a ship or a base, one finds many small computers whose tasks are essentially the same as those in civilian businesses.

Only More So. Mission-critical military software is more universally real-time, communications-oriented, and resource-constrained than its civilian counterparts. At any given time, the demands of weapon systems stress the state of the software art more severely than do civilian demands.

Timeliness and Reliability. Although the cost of military software is commonly seen as the major problem, and is emphasized in our Terms of Reference, both previous studies and our briefers suggest that software timeliness and reliability are even more critical problems today.

Software development cycles are long, relatively unpredictable, and come at the end of total weapon system development. Thus they frequently encounter delays, delays usually on the critical path to operational capability. It also takes too long to adapt running software to changing hardware or operational requirements.

Software reliability is equally of concern. Since operational software is complex, it usually contains design flaws, and these are hard to find and often painful in effect.

Requirements-Setting Is The Hardest Part. As is true for complex hardware systems, the hardest part of the software task is the setting of the exact requirements, including numbers for size and performance, and including the relative priorities of different requirements in the designers' inevitable trade-offs.

We have no technology and only poor methodologies for establishing such requirements. There are not even good ways in common use for even *stating* detailed requirements and trade-off priorities. Misjudgements in requirements badly hurt effectiveness, cost, and schedule. Such misjudgements abound. Most common is the specification of over-rich function, whose bad effects on size and performance become evident only late in the design cycle. Another common error is the mis-imagination of how user interfaces should work.

In our view the difficulty is fundamental. We believe that users *cannot*, with any amount of effort and wisdom, accurately describe the operational requirements for a substantial software system without testing by real operators in an operational environment, and iteration on the specification. The systems built today age just too complex for the mind of man to foresee all the ramifications purely by the exercise of the analytic imagination.

This inherent difficulty is unnecessarily compounded in DoD by the presence of too many intermediaries between the ultimate user and the software specifier.

The Big Problems Are Not Technical. In spite of the substantial technical development needed in requirements-setting, metrics and measures, tools, etc., the Task Force is convinced that today's major problems with military software development are not technical problems, but management problems. Hence we call for no new initiatives in the development of the technology, some modest shift of focus in the technology efforts under way, but major re-examination and change of attitudes, policies, and practices concerning software acquisition.

2.3 Why Is Software Technology Developing So Slowly?

Participants and observers in the computer game often marvel that the software technology develops so slowly, especially in comparison with computer hardware technology. In our Terms of Reference we are charged with examining the underlying nature of the software process so as to explain high costs and slow development.

Hardware Technology Is So Fast.

The remarkable fact is not the slow rate of development of computer software technology, but the fast rate of hardware technology, a fact especially striking to those of us who do both. Today's hardware offers at least a 10,000-fold gain in price-performance over that of 30 years ago, and one can choose at least 1000-fold of that gain in either price or performance! No other technology has come even close to that rate of development. It reflects the shift of computer hardware from an assembly technology to a process technology.

Software Is Labor-Intensive.

Software development is and always will be a labor-intensive technology. The work and the time is all in development, not production. Development is always labor-intensive. Moreover, in the ultimate, one is developing conceptual structures, and although our machines can do the dog-work and can help us keep track of our edifices, concept development is the quintessentially human activity.

The Essence Is Designing Intricate Conceptual Structures Rigorously.

In Appendix A5, we analyze the software task. We argue that its essence is the designing of intricate conceptual structures, rigorously and correctly. The part of software development that will not go away is the crafting of these conceptual structures; the part that can go away is the labor of expressing them. The task is made more difficult by three other properties of software products: (1) the necessity for them to conform to complex environmental, hardware, and user interfaces; (2) the necessity for them to change as their interfaces change; (3) and the invisibility of the structures themselves.

We believe a significant fraction of software development effort today is expended on this essential labor, rather than on the task of expressing the designs.

The Removal of Expression Difficulties Has Brought Much of the Past Gain.

The essential labor itself has not always taken most of the effort. Much of the work was formerly spent on non-essential, incidental difficulties in the expression of the conceptual structures. The three big breakthroughs in software methodology each have consisted of removing one of these incidental difficulties.

First was the awkwardness of machine language. High-level languages removed this difficulty and improved productivity ten-fold.

Second was the loss of mental continuity occasioned by slow turn-around batch compilation and execution. Time-sharing removed this difficulty, improving productivity 2-5 times.

Third was the utter incompatibility of files, formats, and interfaces among various software tools. Integrated programming environments such as Unix and Interlisp overcame this difficulty, again doubling (or better) productivity.

What's In the Cards?

There are stll non-essential expression difficulties, but they do not account for most of the development effort in modern software shops. Future methodological imprcvements will have to attack the essence - conceptual design itself.

Examination of the most promising technological developments shows no single technique that can be expected to yield as much as a 10-fold improvement in productivity, timeliness, and robustness in the next ten years.

On the other hand, all of the various technological developments on the horizon together should easily yield a 10-fold improvement in the next decade. It is not likely that all those developments together will yield a 100-fold improvement.

2.4 Current Software Trends

Five developments in the past decade have revolutionized the software scene. DoD software practices evolved in the 60s and 70s, and they neither take into account nor utilize these advances.

The Microcomputer Revolution and the Personal Computer

The microcomputer, both as a component, and by its incorporation into personal computers, has totally changed the computer field and the software field. Every procedure for computer acquisition, etc., must now define a floor in machine size below which it is not applicable, and machines below the floor should be treated as commodities, components, and spare parts. (Not all procedures have yet been so revised.)

Obviously software standards such as the Ada mandate must have such a floor as well. The constraints on embedded microprocessors are such that their software often must be in machine language. We do not address microprocessor software.

We likewise do not deal with the software problems of personal computers. DoD, like every large enterprise, needs some standards as to how such machines are to be supplied, how they will be equipped with standard-function programs, and how they are to interchange information. Such standardization should be minimal and light-handed. We should not recommend that the Ada mandate cover personal computers.

America's greatest comparative military advantage is the individual initiative and ingenuity of our Service people. We are therefore greatly encouraged to see the Services making personal computers readily available to individual units so that individuals can solve their own simple computing problems their own way. A personal computer and an electronic spread sheet make a powerful combination, sufficient for countless tasks.[2]

A Mass Market for Software

The personal computer revolution has explosively fueled the development of a mass market for third-party developed software. This is the most important development in the software field in our time.

Each of several computer architectures (the properties of a computer that determine what programs it will run) define a market. The biggest are those for IBM PC-compatibles, Apple-compatibles, Macintosh, DEC VAX-compatibles, Unix-compatibles, and IBM 370 compatibles. For each of these markets literally hundreds of packages are available, covering an immense spectrum of functions and costing from a few dollars to a few hundred thousand. The markets are fiercely competitive.

Technology for Software Modularization and Reuse

Techniques for designing software in little modules, for defining the module interfaces precisely, and for using common file formats have come into standard use during the decade. These methods, the backbone of so-called "modern programming practices", radically improve the structure and adaptability of large programs. They also define modules, whose reuse often costs one-tenth as much as writing another module to do the same function. Reuse is also much quicker, and it yields better tested, more reliable code.

The Ada programming language is designed to make such modularization natural, and to provide very powerful facilities- for linking modules. Integrated programming environments, such as Unix, provide the same kind of facility at another level, that of the shell-script linking whole programs together.

Rapid Prototyping and Iterative Development

As people have recognized that the requirements, and especially the user interface, require iterative development, with interspersed testing by users, there has developed a technology for constructing "rapid" prototypes. Such a prototype typically executes the main-line function of its type, but not the countless exceptions that make programming costly. It usually does not have complete error-handling, restart, or help facilities. The prototype is often built using a lash-up of handy components that swap performance for rapid interconnect ability. It is usually run on a computer that is bigger and faster than the target machine.

Commercial packages enable one to prototype graphics interfaces, for example, so that user testing can be done quite early in the development.

[2] Jones and Brooks had the opportunity to observe a Blue-Flag simulated Air Force-Army-Marine tactical exercise. We saw a number and variety of personal computers that have been integrated effectively into unit operational functions; we were pleased to see a light dependence on massive computer systems.

Professional Humility and Evolutionary Development

Experience with confidently specifying and painfully building mammoths has shown it to be simplest, safest, and even fastest to develop a complex software system by building a mninimaal version, putting it into actual use, and then adding function, enhancing speed, reducing size, etc., according to the priorities that emerge from the actual use. Software engineers must recognize that we cannot specify mam moths right the first time. In practice, Version 2 is usually under development before Version 1 is delivered, so Version 3 may be the first to be affected by actual experience.

This procedure speeds first delivery. It also provides for the iterative setting of requirements. It minimizes the specification of heavy function whose performance penalties have not yet been weighed. It tends to concentrate development effort where it will make the most difference. Seeing the minimal version run does wonders for the morale of the development team and substantially boosts their communication as to further development.

Evolutionary development is best technically, and it saves time and money. It plays havoc with the customary forms of competitive procurement, however, and they with it. Creativity in acquisition is now needed.

2.5 Current DoD Programs on Software Technology

Besides some substantial efforts in individual Service laboratories, DoD has under way five programs aimed at enhancing software methodology:

STARS. The program for Software Technology for Adaptable, Reliable Systems, managed by the Undersecretary of Defense (Acquisition), was started in ii80 to address all aspects of modern software methodology as applied especially to mission-critical computer systems.

ADA. The Ada program, managed by the Ada Joint Program Office under the Undersecretary of Defense (Acquisition), was started in 1975 to define and develop a standard high-level language suitable for embedded computer systems.

Software Engineering Institute. Founded in 1964, the SEI mission is focused on technology transfer - bringing the best modern methodology into actual practice in the Services and among DoD contractors. The SEI is operated by Carnegie-Mellon University under contract from the USAF Electronic Systems Command.

Strategic Computing Initiative. A software component under the DARPA Strategic Computing Initiative is aimed at developing radically new methods and tools, especially those based on expert systems and other artificial intelligence techniques. The program aims at results a decade ahead of modern practice.

Strategic Defense Initiative. A software component under the Strategic Defense Initiative is aimed at providing methodological advances for the building of the massive distributed, ultra-high-performance software system demanded by the SDI.

2.6 Recent Previous Studies

The Task Force reviewed the available studies done since 1982, starting with the monumental 1982 Druffel study, which in turn summarized the results of 26 previous studies. Appendix A4 lists the recent ones.

To a surprising degree, the conclusions of these studies agree with each other and remain valid; the recommendations continue to be wise. The chairmen of the several study groups briefed us. All had one message: very little action, has been taken to implement the recommendations. If the military software problem is real, it is not perceived as urgent by most high military officers and DoD civilian officials. Our Task Force does not undertake to prove that it is urgent; we do tell how to attack it if one wants to.

3 STARS – Software Technology for Adaptable, Reliable Systems

The STARS program objective is to achieve by 1995 a dramatic improvement in our ability to build reliable, cost-effective defense software by applying known new technology. STARS seeks improvement in methods, techniques, tools, personnel practices, and business practices.

The Task Force examined the STARS program on several occasions during the past two years. The program is floundering. Little has been achieved during the last several years. OSD management has recognized the problems, and some remedial steps are under way. It is too early to tell if these steps will work.

Findings

STARS as originally formulated is a very good idea.

Members of the Task Force do not expect to see dramatic near-term research discoveries. However, many incremental improvements in software engineering have been made over the last decade, and will continue to be made. These advances could improve our war-fighting capability if they were practiced in DoD programs now. STARS can accelerate their application.

OSD has not provided the vital leadership needed; until recently STARS has lacked a director with strong technical and management ability.

The program had no permanent program manager for over a year. Consequently, it lacked leadership, guidance, and vitality. There has been no single vision of program objectives, no coordination of spending, no monitor to ensure that components of the program were complementary, and no assurance that the program acted in response to the software problems of the Services. Strong top-down leadership, both technical and administrative, is required. A new, permanent director has recently been appointed.

The program plan has been fuzzy.

The Task Force had difficulty in identifying specific goals or plans to achieve them. The program plan does not even recognize the existence of some major software trends, such as the personal computer revolution. STARS should enable solutions, not develop them from first principles. It should identify possible technical approaches and tools, harness the marketplace capability to produce solutions, and ensure early application to real mission-critical system developments.

The STARS Program as it stands today has become focussed at a particular, well-defined part of the military software problem—custom systems, new or converted, middle-sized to large, whether embedded or command and control. STARS addresses follow-through software engineering support for ADA software.

This focusing is entirely commendable and beneficial; indeed, it was badly needed. A corollary is that the problems STARS does not address have also become clear. They include:

- personal computers, workstations, and little software systems built on them
- acquisition of off-the-shelf commercial software
- supercomputer calculations (still and perhaps forever in FORTRAN)
- old systems not worth converting

To enumerate these is not to criticize the STARS program. It never really could address all the problem; it only claimed to do so as long as it was fuzzy and unfocussed.

Balance between program elements has been missing.

Devising a single software engineering environment dominates the attention of the program. In contrast, emphasis on multiple possible environments, even some that are off-the-shelf, would serve the objectives better. Most of what the STARS program proposes to deliver is scheduled very late in its lifetime. Early operational milestones would better speed transfer of the technology to the DoD and civilian practitioners.

The program is organized as uncoordinated activities; many are executed by part-time volunteers.

An independent committee explores each activity area; little communication relates the committee actions. For example, there is insufficient integration of the activities of the business practices area with each of the technical areas.

STARS needs better coordination with the Services, the Software Engineering Institute, AJPO, DARPA's Strategic Computing Program and the Strategic Defense Initiative.

All these programs have interlocking interests and development programs. Links have not been carefully established for the input of Service needs into STARS planning and the output of STARS back to the Services.

It is difficult to get the needed funds allocated for software engineering; if STARS is terminated a major opportunity will be lost.

An effective STARS Program is indeed needed to accelerate the application of the best ideas from the laboratory to weapon systems development. If an effective STARS Program does not materialize, software risks will remain high.

Salvage of STARS may not be possible, but it should be attempted. Drastic action is required.

Recommendations

Recommendation 1: Move STARS and rebuild it.

Create a Joint Program Office to oversee the STARS program, AJPO, and the Software Engineering Institute. This Office should be headed by a flag-rank military officer in order to demonstrate DoD cammitment to provide firm oversight, resources, and control over DoD software technology efforts. This Joint Program Office should report to the Deputy Undersecretary of Defense for Research and Advanced Technology. Locate it at the Electronic Systems Division in Bedford, Massachusetts, with the Air Force as executive agent.

This management organization has been an effective technique for mustering Service cooperation on joint efforts in the past. Examples include WIS, Joint Tactical Fusion, JTIDS, and the Joint Surveillance and Target

Acquisition Radar System. OSD retains oversight authority, and the Joint Program Office organization will ensure that benefits accrue across all the Services.

Recommendation 2: Task the STARS Office, the Ada Joint Program Office, the Software Engineering Institute, the SDI software methodology program element, and the DARPA Strategic Computing Program to produce a one-time joint plan to demonstrate a coordinated DoD Software Technology Program.

This plan must ensure ongoing technical exchange among the five programs.

Recommendation 3: Task the new STARS Director to define a new set of program goals together with an Implementation plan; emphasis should be on visible, early milestones that have demonstrable results.

This plan should emphasize widespread adoption of the best that exists today. It should provide incremental products. It should complement what the commercial sector is doing and focus on DoD-unique requiremrents. It should be realistic.

Recommendation 4: Direct STARS to choose several real programs early in development and augment their funding to ensure the use of existing modern practices and tools.

4 ADA

DoD defined the Ada language (see MIL-STD-1815A) to be its common, machine independent programming language for DoD-wide use in mission-critical computer applications. This intent was established in 1981 by draft versions of DoD Instruction 5000.31. A subsequent June 1983 memorandum from Dr. Richard DeLauer, Undersecretary of Defense (R&E), mandated the use of Ada on all new DoD mission-critical computer procurements entering concept definition after 1 January 1984 or entering full-scale development after 1 July 1984. Mr. Don Hicks, Undersecretary of Defense (R&E), reaffirmed thie mandate to Ada in December, 1985, as did Secretary Weinberger in November, 1986.

The Task Force discussed Ada, its compilers, and its application in military programs with the three Services, inter-Service programs such as WIS, and the acting Director of the Ada Joint Projects Office. The Task Force also considered fourth-generation languages and their implication for the Ada effort.

Findings

Improved Software Engineering Techniques

Software engineering methods and techniques have dramatically advanced over the last decade, yet these techniques are not generally practiced in DoD.

Ada is not merely a programming language; it is a vehicle for new software practices and methods for specification, program structuring, development and maintenance. Without enforced usage of such a vehicle, the radical improvements in software engineering will not move rapidly into use. Standardization on a language is the best way to introduce the new practices rapidly.

Ada, The Standard Language

It is a major technology step forward for the DoD to insist that all software be built in a high-level language. It is a major management step forward to standardize on a single high-level language.

It is not simple to do so; Fortran and Cobol will each survive in some military applications. The driving reasons to standardize new development in one high-level language remain valid. Specifically, the quality of the resulting software will be higher. Enhancement of function, adaptation to environmental changes, and fixing of errors will be less buggy and cheaper.

Even where exceptions to the use of Ada are granted, all software can and should be designed using Ada as a design language.

Ada was designed by the DoD to be that standard language; It is the best candidate for standardization available today; it promises to remain so for the foreseeable future.

Ada's constructs support modern software technology and discipline. Ada supports the evolution and maintenance of reusable software, portable software, and real-time software. The language definition is precise enough. Other candidate languages have many more deficiencies than Ada with respect to the DoD's needs.

Ada is admittedly complex. This complexity has contributed significantly to the slow maturation of the language and of its compilers and tools.

Enough Ada compilers now exist to demonstrate they can be built. Because of language complexity, current compilers execute slowly in comparison to a good Fortran compiler. However, the compilers are doing more checking, and pointing out errors to the programmer; this is cost-effective. Engineering refinement of compilers will yield acceptable, even good, Ada compiler speed in the near term. Moreover, modern partial compilation techniques today reduce the impact of raw compiler speeds.

Due to Ada's complexity, the code generated by current Ada compilers Is not yet highly optimized.

Again, engineering refinement will produce optimizing compilers in the near term. Whereas Ada application code can be quite slow if all dynamic checking is enabled, most checking can be turned off in the production version of application code. There is no technical obstacle to achieving optimized code for applications written in Ada.

The DeLauer mandate to use Ada was premature; it could not be followed In 1983 because of slow maturation of the language and its compilers.

Consequently it became toothless. The compilers have been developed to a point that the mandate can be implemented now; it should be.

Switching to Ada necessitates an up-front investment In order to reap longer term benefits.

One cost is education. Teaching Ada also implies teaching the new software engineering practices and disciplines. This must be done anyway. Forcing this learning is a major motive for adopting Ada quickly and extensively.

Computer time costs will be somewhat higher because of the slower compiling. These costs are transient and will go down as Ada programming environments are widely installed, as the software tools mature, and as hardware cost/performance continues to drop.

Although incurring the up-front costs is wise for DoD, individual program managers and contractors have no incentive to do so.

The costs of training, compiler and tool acquisition, and running the current immature compilers are present and readily measured, whereas the benefit is future and more difficult to measure. Adoption must therefore be mandated by high management.

Ada is being successfully used today in military programs, such as AFATDS.

At least sixty-four validated compilers exist, with more in the wing. Moreover, Ada is not just a DoD captive language. Civilian commitment to Ada is emerging. It is noteworthy that the majority of compilers for Ada are built with private, not DoD, funds. One cannot predict, however, that Ada will become the standard language for civilian data proceasing, as Cobol did. Too many different forces are at work.

Fourth-Generation Languages

Fourth-generation languages are application-specific program generators; because they are not general purpose, they are not in competiton with Ada.

The term fourth-generation is a misnomer. It has been used to characterize a wide variety of languages which are not descendants of the third-generation, general-purpose languages. The term encompasses application-specific languages such as database languages and electronic spread sheets, program generators, non-procedural languages, and even artificial intelligence languages such as Prolog. Each language is designed to be applied to problems in a limited domain. Therefore the fourth-generation languages do not compete with Ada.

If an application is well-matched to a fourth-generation language, the cost of realizing the application can be a hundredfold less expensive than programming it in any general purpose language, Including Ada.

Spreadsheets are routinely used to accomplish tasks in minutes that would require hours of work in a general purpose language. Similarly, an exploratory artificial intelligence (AI) task may be programmed in an AI language in days versus months in any general purpose language.

A weapons system development is not one task in a single problem domain; the Task Force is skeptical that any fourth-generation language is well-suited to such applications.

Note that some of the high-risk tasks in a weapons system may be advantageously prototyped in a fourth-generation language to experiment with algorithms or software structure before actual development commences. Similarly, some single-domain mission critical applications, such as some intelligence data processing, may be cost-effectively implemented in a fourth-generation language such as a database provides.

Some efforts to develop large software systems entirely in a fourth-generation language, such as the New Jersey Motor Vehicle Registration System, have been unsuccessful.

DoD Management of Ada

Only top DoD management can sustain a policy and program for incurring the costs and risk of early DoD use of Ada.

Contractors incurring the up-front costs must have assurance that investment in Ada tooling will pay off. Programs must plan for long-range cost and quality improvements.

The Ada Joint Program Office (AJPO) is the DoD's focal point for policy and coordination of Ada standardization, validation, and language control; it has done a commendable job in achieving its technical objectives.

The AJPO has maintained a stable language definition. It has defined a comprehensive validation suite of language conformity tests. Note that language conformity does not ensure that, a compiler is robust, acceptably efficient at compile time, or capable of generating correct or efficient code for real applications. Concentrated focus on language conformity has slowed compiler maturity along these other dimensions. The AJPO has also performed a communication function with its Ada Information Clearinghouse.

Definition of the Ada language and development of compilers has been successful; the next step is to implement DoD applications in Ada.

This step is mainly acquisition management and is discussed elsewhere. AJPO can best assist by providing truthful, complete, and candid information about compiler and application activity.

The next technical step is to develop Ada support tools beyond compilers and to integrate them with one another and with the underlying operating system.

The unclear boundary between the AJPO and the STARS programs' charters has led to some confusion of who should develop what support software technology.

Ada support tools

Ada has been overpromised.

The Ada language embodies much current software technology. But to build application code that is portable and reusable requires disciplined use of the new engineering practices and tools as well. Such support tools are not yet integrated with the compilers. Support tools are needed for such activities as:

- software documentation writing and formating;
- version and configuration control of both software and documentation;
- maintaining development history in a way that links requirements, design specification, code documentation, source code, compiled code, problem reports, code changes, tests, and test run results;
- debugging; and
- project schedule and effort management.

As a consequence, non-technical managers of programs are expecting results that no high-level language can by Itself deliver.

Environments that integrate such tools are not yet available for Ada. They are likewise available only piecemeal or not at all for Jovial, C, CMS, Fortran, etc.

Acquiring these environments is the next step.

The DoD is supporting efforts to develop such environments. Environment design is more difficult than language definition. Efficient environments depend integrally upon the host operating system, yet one wants tool portability and language independence. We expect that market forces will produce a variety of environments around Ada if the DoD maintains firm commitment to the language.

Recommendations

Recommendation 5: Commit DoD management to a serious and determined push to Ada.

Management waffling is more likely to cause a failure in Ada than are technical or acceptance problems.
Specifically, the DoD should

- Finalize and issue software language policy which reaffirms and details the policy set forth in the DeLauer and Hicks memoranda, and the Weinberger speech.
  - The DoD should establish Ada as the common, machine-independent, mission critical computer system programming language for DoD-wide use. The mandate should not be limited to embedded computers in weapon systems.
  - Each DoD component should develop and implement a plan for cutting over to full Ada usage. These plans should provide for support software, education, and training of military, technical, and management personnel.
- Stiffen practices for granting exceptions from Ada policy so that exceptions are difficult now and become increasingly difficult with time.
- Mandate that where implementation exceptions are granted, software should nevertheless be designed using Ada as the design language.

Recommendation 6: Move the Ada Joint Program Office Into the same organizationas STARS and the SEI.

The major objective for Ada has become one of implementation — using the Ada language for DoD systems — now that the AJPO has technical control of the language. Common management of these three programs will strengthen each and permit easy coordination of common goals and objectives.

Recommendation 7.: Keep the AJPO as the technical staff support agent for the DoD's executive agent.

Specifically, it should:

- Continue firm control of the Ada language definition, permitting only minimal, if any, change to the now-stable language for the next two years.

- Continue Ada compiler validation. In enlarging the validation suite, give priority to adding tests that are representative of real applications.
- Encourage the definition of Ada working vocabularies. Promulgate them.
- Change the AJPO communication function to reduce the overselling of Ada and to gather and report accurate, credible information on Ada tools and usage. It should:
  - provide information on the existence and performance of Ada compilers and environments.
  - document experiences of the application of Ada including both success stories and lessons learned.
  - disseminate significant Ada information via newsletters, on-line data bases, books, articles, workshops, conference presentations, and videotapes.
  - continue to act as a clearing house recording availability of existing and reusable Ada packages, or even entire tools, and objectively reporting on the experience with that software.
- Continue the effort to establish a performance test suite as a companion to the language conformity test suite.
- Locate software measurement techniques and tools. Publicize them and make these tools and techniques available to project managers using Ada.
- Initiate significant measuring and recording of lifecycle costs for several major Ada application programs.
- Continue to encourage the development of programmer support environments built on Ada, but be slow to standaidize environments. Let a winner emerge first. In particular, ALS, ALS/N, AIE and CAIS should not be standardized until and unless experience with prototypes shows implementations to be effective and efficient.

Recommendation 8: DoD policy should continue to forbid subsetting of the Ada language.

There must be only one definition of the language. Further, all compilers should correctly process the entire language; the validation process should continue. Without this, much of the benefits of standardization will be lost. However, organizations and educators should be encouraged to establish and publish useful working vocabularies to simplify the task of learning and using Ada.

Specific working vocabularies may change as the technology matures. We see three categories - acceptable vocabularies, vocabularies to be used with certain constraints, and vocabularies that might not be used until the compilers and runtime environments mature. Tasking exemplifies the second category. The use of tasking has to depend upon the performance capability of the specific compilers. In the third category, today the Use statement might be limited because an inordinate amount of recompile time is needed.

Recommendation 9: The DoD should increase investment in Ada practices education and training, for both technical and management people.

Each DoD component's implementation plan should include provisions for extensive, in-depth Ada education and training. Do not underestimate the education and training required for managers, analysts, and administrators, in addition to that for software engineers.

Recommendation 10: Allow fourth-generation languages to be used where the full life-cycle cost-effectiveness of using the language measures more than tenfold over using a general-purposelanguage.

Marginal increases should not dictate using such languages, especially for long-lived, production software. The estimated cost of a program element built in a fourth generation language must include full life-cycle cost, including development, integration of the program element with others built in Ada, as well as the increased maintenance costs of support software written in multiple languages.

5 Strategic Defense Initiative Software

Findings

The Strategic Defense Initiative (SDI) has a monumental software problem that must be solved to attain the goals of the initiative.

It is critical quite out of proportion to its cost, because hardware has high replication costs and software does not. Initial contractor proposals therefore largely ignored it.

The software problem has already received considerable public attention and notoriety.

No program to address the software problem is evident.

Recommendations

Recommendation 11: Focus a critical mass of software research effort on the software needs that are unique to the SDI objectives.

The SDI should use what STARS, the SET, DARPA and industry produces. Much of the software problem faced by the SDI is due to the magnitude of the required software and the complexity of controlling the interactions of so many components with very rapid communication and response. This is not a unique requirement. The SDI should determine what portion of their software problem is unique and concentrate its attention on solving the SDI-unique problem, not the general software technology problems.

Recommendation 12: Use evolutionary acquisition, including simulation and prototyping, as discussed elsewhere in this report, to reduce risk.

6 DoD and the Civilian Software Market

Findings

The civilian market for software is today substantially larger than the size of the DoD market, although the DoD continues to be the largest single customer for computer software [Jorstad, 1984; Boehm, 1986].

This new phenomenon requires a radical update in DoD thinking, policies, and procedures.

We find that in policy drafting and debate, the mass civilian market is generally ignored.

One important implication is that DoD cannot, as it did with Cobol, create a *de facto* standard and impose it on the civilian market. This is not to say that the civilian market will not adopt tools and methods from DoD where they are perceived as advances. We merely observe that it will not adopt them Just because DoD has.

A second implication is that DoD, although it will necessarily lead in some aspects of the technology such as interfaces to sensors and effectors, cannot expect to lead in most aspects of software technology development.

A corollary is that wherever DoD's software methodology diverges from what the civilian market is evolving by competitive selection, it will have to support its own idiosyncratic methodology all by itself, without the resource commitment of the larger economy.

Computer Security and Commercial-Off-The-Shelf-Software. Computer security requirements are frequently cited as a reason why commercial off-the-shelf software cainot be used. The National Computer Security Center in NSA has published criteria for assessing the effectiveness of security controls in ADP systems (DoD CSC-STD-001-83). The Center is also working on guidelines for matching the appropriate level of security controls to a problem. With support from the National Computer Security Center, industry is now developing operating systems, trusted computer programs, guards, and other software and hardware products with security controls built in to the hardware and software and is seeking certification for these products. This approach will provide more standard and cheaper ways of dealing with computer security than the current practices of custom-tailoring systems. It will also facilitate integration of computer security controls, with communications security systems through the use of low-cost encryption devices and standard interfaces for network control.

Enlisting Industry and Universities. Our Terms of Reference explicitly charged us with suggesting how DoD can enlist the efforts of universities and industry in its software technology advance. Our response must be that this charge itself assumes the bygone situation. It is now the case that DoD, in mapping its software thrust, must in part assess which way the technical thrust of the larger community is going, and diverge only when absolutely necessary.

On those aspects where DoD is truly innovating technically, it can enlist the larger community by the very excellence of the innovations — the competitive marketplace is responsive to innovation.

Recommendations

Recommendation 13: The Undersecretary of Defense (Acquisition) should adopt a four-category classification as the basis for acquisition policy

We see vast differences in the software systems that DoD buys and builds. We recommend that these differences should be explicitly recognized by an official classification into four major classes according to uniqueness and novelty. Acquisition guidance, policies, and procedures should be framed separately for each class. The classes are:

Standard:	Off-the-shelf, commercially available
Extended:	Extensions of current systems, both DoD and commercial
Embedded:	Functionally unique and embedded in larger systems
Advanced:	Advanced and exploratory systems.

Each Program Manager would classify his system, its subsystems, major components, major increments, and phases into one of these classes, with the burden of proof being to show why the element should be in a higher class instead of a lower one.

Table 5.1 provides attributes of the four categories. Table 5.2 illustrates the categories by classifying some current acquisitions.

It may also be wise to establish categories by size (lines of source code) for uniformity in description. A possibility might be:

Modest:	Under 2000 LOSC
Small:	2000-10,000 LOSC
Medium:	10-100 KLOSC
Large:	Over 100 KLOSC

Then a planned undertaking could be characterized for policy or procedure purposes as, for example, a "Medium Extended-Class software project."

Recommendation 14: The Undersecretary of Defense (Acquisition) should develop acquisition policy, procedures, and guidance for each category.

Recommendation 15: The Undersecretary of Defense (Acquisition) and the Assistant Secretary of Defense (Comptroller) should direct Program Managers to assume that system software requirements can be met with off-the-shelf subsystems and components until it is proved that they are unique.

The *cheapest* way to get software is to buy it in the commercial marketplace rather than to build it.

The *fastest* way to get software is to buy it in the commercial marketplace rather than to build it.

The *surest* way to get robust, maintained, supported software is to buy it in the commercial marketplace rather than to build it.

Even though commercial software is often delinquent in these latter respects, competitive pressures get the delinquencies fixed. Custom-built software has historically been notoriously bad in these respects, without the same pressures to fix it.

Hence the DoD-wide assumption should be that a commercial product will be usable if the function is similar to that required — perhaps with modifications or extensions, perhaps with extra documentation, perhaps with different support and maintenance arrangements. The first investigation when requirements begin to be formulated should be into the market, to see what is available already. Even if the best off-the-shelf product is not ultimately used, adopting it for pilot use will help radically in setting specifications for the custom product.

Recommendation 16: All the methodological efforts, especially STARS, should look to see how commercially available software tools can be selected and standardized for DoD needs.

Although end-user systems, especially embedded ones, will often need to be specialized, and perhaps custom-built, it will rarely be justified for DoD to custom-build the tools with which its software is built. The assumption should be that marketplace tools will be used.

7 DoD In a Sellers' Market for Software

Findings

Because of the explosion of the commercial software market, DoD now Is In a sellers' market for software-building.

Just as the DoD need for software is growing exponentially, and its software-skilled personnel grow more slowly, the same is true of the commercial software market. Programmers are in short supply, programming managers even scarcer, and software system designers very scarce. Hence the companies that have these skills, and have them organized into functioning, equipped teams, have many choices as to how best to market their services.

DoD is perceived as a poor customer, and the stable of DiD custom software vendors stays small even though the requirement grows radically.

Poor Return. Building custom software for DoD has a poor profit margin. In calculating proper profit levels for cost-plus-incentive contracts, DoD tends to use the same margin for software development as for hardware development, although the latter is customarily followed by a production cycle at acceptable total profit levels. Ten percent profit on sales is considered high in DoD, whereas it is grossly unacceptable in computer industry pricing on software.

Weak Incentives for Productivity. Even on fixed-price software contracts, there are only weak incentives to manage for higher productivity or to invest company money in capital tooling that will save labor cost. High productivity and high quality are not rewarded by DoD except at the time of contractor selection. If "excessive" profits result from high productivity on firm-fixed-price contracts, the profits are readjusted after audit. The standard for "excess" is the same as that for hardware development, despite the absence of the production cycle.

Heavy Regulation. DoD Directive 5000.29, "Management of Computer Resources in Major Defense Systems", DoD STD 2167, "Defense System Software", and the proposed new Federal Acquisition Regulation FAR 27.4, "Data Rights", and the proposed DoD FAR Supplement 27.4 have as their purpose to ensure fair, consistent, and open competition, and to get the most capability for each dollar spent. In practice, however, they inhibit the use of standard commercial practices in software acquisition and maintenance. They also encourage the building of new software rather than purchase of off-the-shelf items.

Unreasonable Rights-In-Data Requirements. DoD FAR Supplement 27.4, as proposed for public comment by 10 January 1986, set forth demands for DoD ownership of rights to programs, documentation, tools, and methods that were

- formulated only in terms of hardware, entirely ignoring the different circumstances of software,
- completely at variance with commercial practice for software,
- probably illegal under U.S. Copyright law, and
- destructive of most vendor incentive to hivest in better tools and methods.

We were convinced that the misfit between the proposed supplement and software is entirely unintentional, and so the Task Force made timely presentations to the Undersecretary of Defense (Acquisition) and to the Assistant Secretary for Acquisition and Logistics during the comment period. If the proposed supplement were to be adopted, however, it would be another powerful disincentive for vendors to bid on DoD custom software development.

Constant, Small, Stable of Vendors. Given the regulatory and financial structure of DoD contracting outlined above, a vendor can participate in substantial DoD software contracting only if it:

- has staying power to average over crests and troughs in contracting business,
- has a management superstructure to cope with the regulatory overhead, including a Program Control Management System, and specialized accounting to meet DoD STD 7000.2,
- has an infrastructure of technical specialists to deal with configuration management per regulation, documentation per regulation, acceptance testing, etc.
- therefore has a critical mass of skilled people so that it can carry several contracts at once, both to smooth crests and troughs and to pay for the administrative and technical superstructure necessary to cope with the regulatory overhead.

As a result, we estimate there to be only about two dozen houses that are regularly available to participate in the development of substantial mission-critical software systems, and this number grows slowly. On the contrary, several vendors are seriously considering leaving the DoD business entirely, and refusing to bid in the future.

The net result of this small stable of vendors in a time of exponential growth of work is sure to be higher bids and longer schedules. It is in DoD's interests, and the nation's interests, for DoD to make itself an attractive customer. We believe the net costs to the nation of weapon-system software will be lower if it does. Present practices are penny-wise and pound-foolish in many petty ways.

Recommendations

Recommendation 17: DoD should devise increased productivity incentives for custom-built software contracts, and make such incentivized contracts the standard practice.

A new contracting form, part-way between fixed-price and cost-plus-fee, should be devised. For instance, on a cost-type contract, a productivity figure is usually bid. Competition in vendor selection keeps the figure from being unreasonably low. So reimbursement might be structured to split any savings due to increased productivity evenly between the buyer and the seller.

Another new contracting form that we recommend DoD consider would be to guarantee a quantity buy of some software product and to request bids solely on a per-copy price. Here the vendor would bear all the non-recurring costs and recover gains on capital investment and productivity enhancement. Ada compilers and software development and maintenance environments are examples that could be purchased or licensed this way.

Recommendation 18: DoD should devise increased profit incentives on software quality.

One such incentive could be a sliding profit margin based on the quality of the delivered complete software product. This requires quality metrics, recommended below.

Another kind of incentive could be introduced by requesting contractors to bid a per-copy-per-year fixed license fee including maintenance. In this way, high quality resulting in low maintenance would provide financial rewards to the contractor and operational rewards to the users.

Recommendation 19: DoD should develop metrics and measuring techniques for software quality and completeness, and Incorporate these routinely in contracts.

There are today no metrics for source-code quality, object-code quality, documentation quality, etc. Part of the STARS methodological effort should be addressed to such metrics, for Ada programs in particular.

Meanwhile, there are techniques for judging the over-all quality of complex performances outside the computer field. There is wide agreement as to what quality is, and skilled practitioners make similar judgements when presented with products to judge. Even today software quality can be judged by panels of trained judges, just as such panels judge Olympic diving, skating, and acrobatic performances. DoD should immediately begin testing such panel methods for consistency and reliability, and, if they work, begin using such judgements for quality incentives.

DoD buys software for operational systems that will be used for a decade or more. The software must be maintainable and changeable. DoD is willing to pay the price for complete software products, but all too often it accepts less because of schedule slippages and operational needs.

Complete software products should be mandated in contracts to include:

- specifications the describe the actual software structure as built, as opposed to that originally specified,
- documentation showing the structure and organization of the software,
- source code that is properly structured, well modularized, and well commented, especially in procedure headers and variable declarations,
- cross-reference documentation that traces articles in the specifications to the corresponding source code, and vice-versa.

Recommendation 20: DoD should develop metrics to measure Implementation progress.

Such metrics would help ensure that costs and schedules are being met and that complete products will be delivered. They might include, for example, program size, software complexity metrics, personnel experience, testing progress, and incremental-release content. Development of such has in the past been part of the STARS plan; it should continue to be. Meanwhile, panel-judging techniques as discussed above can be applied to progress as well as to quality.

Recommendation 21: DoD should examine and revise regulations to approach modern commercial practice insofar as practicable and appropriate.

Recommendation 22: DoD should follow the concepts of the proposed FAR 27.4 for data rights for military software, rather than those of the proposed DoD Supplement 27.4, or it should adopt a new "Rights In Software" Clause as recommended by Samuelson, Deasy, and Martin in Appendix A 6.

The legal problem is highly technical. Two good solutions, the arguments for proposed FAR 27.4, the concerns about the clarity and applicability of the proposed DoD Supplement 27.4, and the arguments for a new clause are all skillfully set forth in the SEI Technical Report incorporated herein as Appendix 6A.

To those technical arguments we would add an economic one: the proposed DoD Supplement 27.4 is, intentionally or otherwise, grabby in spirit and effect. No fair-minded person would consider it to propose equitable treatment among vendors, or between DoD and vendors. Its lack of clarity and clumsiness of drafting will occasion litigation. The net effect will be to further shrink the vendor pool, at great cost to DoD and the taxpayer.

8 A New Life-Cycle Model for Custom DoD Software

Findings

The most common present method of formulating specifications — issuing a Request for Proposal, accepting bids, and then letting a contract for software delivery — is not in keeping with good modern practice and accounts for much of the mismatch between user needs and delivered function, cost, and schedule.

As discussed above under Current Trends, we now understand the importance of iterative development of requirements, the testing of requirements against real users' needs by rapid prototyping, and the construction of systems by incremental development, with early incremental releases subjected to operational use.

The Task Force finds that Directive 5000.29 and STD 2167, as interpreted, have made it difficult to apply these modern methods.

Although some parts ef the recent Draft DOD-STD-2167A appear to encourage modern methods, the draft as a whole continues to reinforce exactly the document-driven, specify-then-build approach that we believe causes so many of DoD's software problems.

Recommendations

Recommendation 23: The Undersecretary of Defense (Acquisition) should update DoD Directive 5000.29, "Management of Computer Resources In Major Defense Systems", so that it mandates the iterative setting of specifications, the rapid prototyping of specified systpems, and incremental development.

We propose that the iterative development, of specifications can be reconciled with the needs of fair and open competition by letting two level-of-efforts contracts for the specifying and prototyping of major software systeLms. After prototyping is complete and specifications formulated, a software production contract can be put out for bidding in the usual process.

An alternative for the specification process is to let a separate contract to a government contractor for specification, with the specifying contractor excluded from bidding on the build.

The iterative development of specifications is a small part of the total cost of a major software system, usually less than 10%, but it has profound effects on the procurement cost, life-cycle cost, schedule, and function of the product.

We believe the procurement cycle should be modified so that the requirements remain unfrozen and subject to alteration until the cost and performance effects of their provisions can be known from early product design. This means final requirements would not be frozen until perhaps one-third of the way through the procurement period, a substantial departure from present practice.

Recommendation 24: DoD STD 2167 should be further revised to remove any remaining dependence upon the assumptions of the "waterfall" model and to institutionalise rapid prototyping and incremental development.

Recommendation 25: Directive 5000.29 and STD 2168 should be revised or superseded by policy to mandate risk management techniques In software acquisition, as recommended In the 1983 USAF/SAB Study.

The Air Force Scientific Advisory Board in 1983 identified software risk factors and recommended risk management techniques [Munson, 1983]. While parts of the recommendations are Air-Force-specific, the ideas are applicable to all Services. An example of how these risk management techniques have been incorporated into program management is included in table 7.1 below. The risk-management approach provides an effective way for a project to determine when, where, and how much to use prototyping and similar risk-reduction techniques.

Table 7.1: Software Risk Management Plan

1. Identify the project's top 10 risk items.
2. Present a plan for resolving each risk item.
3. Update list of top risk items, plan, and results monthly.
4. Highlight risk-item status in monthly project reviews.
5. Initiate appropriate corrective actions.

Recommendation 26: Each Service should provide its software Product Development Division with the ability to do rapid prototyping In conjunction with users.

The DoD software system acquisition agents are the service product development divisions. Each of these divisions needs facilities and equipments to mock-up, simulate, and build critical prototypes of the new systems being acquired. Some of the system interfaces can be tested prior to delivery to the users. Such facilities can also serve as a place for the developer and user to meet and refine requirements and procedures. The placement of such facilities at the product development division level will allow their use by multiple Program Managers who all report to the same local commander. The product development divisions are organized along mission

categories, and programs in each division will tend to need equipment and software with similar power and capabilities. The facilities could also be used for Ada training.

Recommendation 27: Each Service should provide its software Using Commands with facilities to do comprehensive operational testing and life-cycle evaluation of extensions and changes.

The user commands are responsible for defining the original requirements. However, many of the systems that are being developed are doing jobs that have never been done before. These types of systems tend to be technology-driven and must be placed in the hands of the user as early as possible to develop new operational procedures. In the early phases of system acquisition, a facility at the user command is needed for testing, evaluation, training, and procedure development. Throughout the life of the system, the user commands need the facility for testing and evaluation of changes and upgrades to systems.

Recommendation 28: The Undersecretary of Defense (Acquisition) and the Assistant Secretary of Defense (Comptroller) should by directive spell out the role of Using Commands in the evolutionary and incremental development of software systems.

The relationships between Developing Commands and Using Commands for the different kinds of systems should be spelled out in policy statements. The role and responsibilities of the user commands can vary with the system acquisition procedures and the kind of system being acquired. In conventional acquisitions such as weapons, platforms, or sensor systems, a system is developed, tested, and turned over to the user. For command and intelligence systems, much of the development and testing can take place at the user command. The experience of the users with the first capability built can be used (or required) as feedback to the second increment of the system. User involvement is obviously heavier when evolutionacy acquisition procedures are used.

9 Module Reuse in DoD Custom Software

Findings

Software technology now enables the extensive reuse, even in mission-critical embedded systems, of software modules written for other systems.

The typical module size is on the order of one to two pages of source code, 25-50 source lines. Ada in particular allows modules to be used easily and safely, because the module interface is packaged and specified separately from the body, which the user does not ordinarily need to inspect or alter.

Module reuse requires new forms of contractor incentives, both to make modules available for others to use, and to use them themselves instead of building anew. Making a module reusable requires a modest extra effort in design and development. There has to be incentive and compensation for this effort.

Module reuse requires the establishment of clearinghouses or markets where modules can be exchanged.

Module exchange requires the establishment of standards of description of function and of degree of testing.

Recommendations

Recommendation 29: The Undersecretary of Defense (Acquisition) should develop economic incentives, to be incorporated into standard contracts, to allow contractors to profit from offering modules for reuse, even though built with DoD funds.

Recommendation 30: The Undersecretary of Defense (Acquisition) should develop economic incentives, to be incorporated Into all cost-plus standard contracts, to encourage contractors to buy modules and use them rather than building new ones.

Acquisition contracts should be structured so that contractors will be motivated to build and sell reusable software, and to buy it. Reusable software will be successful when contractors decide it is in their competitive self-interest to reuse software rather than to develop it each time. The proper incentives with respect to data rights, warranties, licenses, liabilities, and maintenance must be included in the RFPs and the contracts.

Recommendation 31: The Undersecretary of Defense (Acquisition) and Assistant Secretary of Defense (Comptroller) should direct Program Managers to identify in their programs those subsystems, components, and perhaps even modules, that may be expected to be acquired rather than built; and to reward such acquisition In the RFP's.

Recommendation 32: The Software Engineering Institute should establish a prototype module market, focussed originally on Ada modules and tools for Ada, with the objective of spinning it off when commercially viable.

A scheme for how such a marketplace might work, including some possible financial and licensing arrangements, is proposed in Appendix A7.

The White Sands Missile Range is operating today an Ada Software Repository, apparently using volunteer labor and spare computer capacity. We believe that a more regular and reliable service must be based upon licensing and license fees, and our proposal includes that. Rudimentary validation of compilability by the marketer may also be necessary.

Recommendation 33: The Software Engineering Institute, in consultation with the Ada Joint Program Office, should establish standards of description lor Ada modules to be offered through the Software Module Market.

10 Software-Skilled People

DoD's demand for software capability grows exponentially. It does so at a greater rate than the combined growth in the size and productivity of the pool of software personnel.

Previous DoD studies have identified personnel issues as critical elements of DoD's software problems. These studies have made excellent recommendations on personnel issues, but the recommendations have not been acted upon.

The Task Force recommends a new approach to the software personnel problem.

Findings

Previous Studies Have Made Good Recommendations

A number of previous studies of the DoD software problem have identified the scarcity of in-house DoD software personnel as a critical problem. They have developed similar sets of recommended actions for dealing with the problem, e.g.:

- Determine DoD's needs for the various software-related skills,
- Create and maintain a skills inventory for DoD personnel,
- Create and implement more attractive career paths for DoD software personnel,
- Establish educational programs to support these career paths.
- Analyze the factors influencing the development and retention of DoD personnel with the appropriate mix of software-related skills,

Previous Recommendations Have Not Been Acted Upon

If these actions were vigorously pursued, they would go a long way toward solving the problem.

However, for various reasons, DoD and the Services have not acted on these previous recommendations. It is therefore unlikely that any effective action would result from yet another restatement of these recommendations by this Task Force.

We believe the pool of DoD software personnel has remained about the same size for many years.

The national pool of software personnel Is growing rapidly.

The number of Bachelor's and Master's Degrees in Computer Science, Mathematics, and Statistics are shown in Table 9.1. Historically, the supply of computer science graduates has been augmented primarily from graduates in Mathematics. These sources of new personnel meet requirements estimated at 54,000 new graduates per year in 1983 [Hamblen, 1984] to sustain a pool of computer specialists whoee size was estimated at 299,000 in 1982 [Vetter, 1985] and whose annual growth rate is estimated at about 5% [NSF, 1984].

It appears that DoD is not competing effectively with the private sector in attracting and retaining software personnel.

DoD needs software talent primarily to support the acquisition process

We agree with previous studies that the software personnel shortage hurts DoD most in the area of software acquisition management.

MITRE and TRW experience have found software acquisition has been most effective when DoD had an acquisition management cadre whose size is roughly 10% (5-15%) of the size of the developer's staff; a cadre with a thorough understanding of software technology and acquisition management.

The people in such a cadre are not just watchers. They add considerable value to the software product by developing specifications, operational concept documents, and life cycle plans; managing competitive concept definition, preparing precise RFP packages, performing thorough source selections, including pre-award audits

Table 9.1: Degrees In Math/Statistics and Computer Science, 1970-82

	Bachelor's Degrees		Master's Degrees	
Year	Math/Statistics	Computer Science	Math/Statistics	Computer Science
1970	27,442	1,544	5,636	1,459
1971	24,801	1,624	5,121	1,588
1972	23,713	3,402	5,198	1,977
1973	23,067	4,305	5,028	2,113
1974	21,635	4,757	4,834	2,276
1975	18,181	5,039	4,327	2,299
1976	15,984	5,664	3,857	2,603
1977	14,196	6,407	3,695	2,798
1978	12,569	7,224	3,373	3,038
1979	12,115	8,769	3,578	3,055
1980	11,473	11,213	2,868	3,647
1981	11,078	15,121	2,567	4,218
1982	11,599	20,267	2,727	4,935

Source: Table 12 of [Vetter, 1985]

and independent cost estimates; exercising prototypes for realistic user feedback; improving specifications, plans and manuals; monitoring the effective performance of quality assurance, configuration management; and subcontracting and financial management, representing user interests on change control boards.

DoD does not have adequate career paths for software professionals

Some Services have no career paths; some Services alternate computer assignments with totally unrelated non-computer assignments, thereby diffusing the officer's experience. In many cases, software expertise is encoded as a subspecialty inflection rather than in a primary specialty code.

Software engineering methods and techniques are advancing dramatically. It is critical for software professionals to master these advanced methods and techniques and to keep learning new techniques as they are developed. This is as important for acquisition people as it is for production people. (Building architects have to know the technologies better than most contractor people.) In-house software skills do not match those of top contractor pools.

Current deployment of the software talent pool is ineffective

Currently, many software-qualified personnel are assigned to jobs that could effectively be assigned to contractors. Many DoD software acquisitions are either in-house development efforts staffed entirely with DoD personnel or contracted acquisitions with DoD staffing levels far below the needed 5-15%.

Recommendations

Recommendation 34: Do not believe DoD can solve its skilled personnel shortage; plan how best to live with it, and how to ameliorate it.

The software personnel shortage will not disappear by direct DoD action. All DoD plans should be based on the assumption that an acute skill shortage will persist. Organization structures should be tuned, assignment policies should be adjusted, and educational programs should be revised to produce the military and civilian cadre needed to acquire and maintain highly complex systems.

Further, DoD should facilitate supplementing the software acquisition management process with contractor support where the supply of in-house personnel is insufficient.

Recommendation 35: Use DoD people for acquisition instead of construction.

Instead of hoping that enough personnel can be hired and retained to satisfy the needs of the current strategy for using software personnel, change that strategy to train and assign available personnel to the highly-leveraged tasks, namely software acquisition management. DoD should sharply reduce in-house software construction, extension, and maintenance, limiting such to critical functions at operational bases, adaptation of existing software to local needs, and special security-sensitive work.

Recommendation 36: Establish mechanisms for tracking personnel skills and projecting personnel needs.

No meaningful studies have been found that catalog seasoned personnel, and no studies have been found that include uniformed personnel and government civilians.

Each Service needs to have, and all DoD needs access to, a database that covers its officers, its senior and technically skilled enlisted people, and its technically skilled civilian employees. This should show not only career history and assignments, but technical skills, experiences, and trainings by quite fine subject codes.

From such a database each Service should not only draw for particular assignments, but also project biennially the trends by skill, by seniority, by median age within skill, and by years of particular skill experience. Such trends can then be assessed against projected needs.

Recommendation 37: Structure some officer careers to build a cadre of technical managers with deep technical mastery and broad operational overview.

Where possible, operational assignments should be chosen to give intense systemusing experience in real operations, development/acquisition assignments should be on related software systems; and education assignments should focus on new technical and management approaches.

Recommendation 38: Enhance education for software personnel.

DoD should implement the education and training necessary for its software acquisition management personnel to master both software technology and acquisition management.

11 Appendices

A1. The Task Force

A2. Terms of Reference

A3. Meetings and Briefing

A4. Documents Studied

A5. Software — Why Is It Hard?

A6. Proposal for a New "Rights in Software" Clause

A7. Proposal for a Module Market

Appendix A1

DEFENSE SCIENCE BOARD SOFTWARE TASK FORCE

Members and Key Persons

Dr. Donald Hicks, USD(R&E), Sponsor 202-695-6639

Dr. James P. Wade Jr., Deputy USD(R&E), Acting ASD(A&L), Sponsor

Dr. Victor Basili, Member
Chairman, Department of Compute: Science
University of Maryland
College Park, Maryland 20742
301-454-2002

Dr. Barry Boehm, Member
TRW Defense Systems Group
One Space Park, R2-2086
Redondo Beach, CA 90278
213-535-2184

Ms. Elaine Bond, Member
Senior Vice President
Chase Manhattan
One New York Plaza, 21st Floor
New York, NY 10081
212-676-2982

Dr. Frederick P. Brooks, Jr., Task Force Chairman
Kenan Professor of Computer Science
University of North Carolina at Chapel Hill
New West Hall 035A
Chapel Hill, NC 27514
919-962-2148

Mr. Neil Eastman, Member
IBM Federal Systems Division
18100 Frederick Pike
Building 929, Room 1C12
Gaithersburg, MD 20879
301-240-2170

MGen. Don L. Evans, Member
USAF (ret)
President
Tartan Laboratories
461 Melwood Avenue
Pittsburgh, PA 15213
412-621-2210

Dr. Anita K. Jones, Member
Tartan Laboratories
461 Melwood Avenue
Pittsburgh, PA 15213
412-621-2210

Dr. Mary Shaw, Member
Software Engineering Institute
Carnegie-Mellong University
Pittsburgh, PA 15213
412-578-7731

Mr. Charles Zraket, Member
President
The MITRE Corporation
Burlington Road
Bedford, MA 01730
617-271-2356

Dr. Edward Lieblein, Government Representative, OSD
202-694-0208

LTC Bill Freestone, Government Representative, Army
202-694-7298

Mr. Marshall Potter, Government Representative, Navy
202-697-9346

LTC Dave Hamond, Government Representative, Air Force
202-697-3040

Major Susan Swift, USAF, Executive Secretary
202-695-7181

CDR Chris Current, DSB Secretariat Representative
202-695-4157

CDR Mike Kaczmarek, DSB Secretariat Representative
202-695-4157

Mr. Robert L. Patrick, Contractor Support
Willow Springs Road
Star Route 1, Box 269
Rosamond, CA 93560
805-256-4444

Mr. John K. Summers, Contractor Support
The MITRE Corporation W90
7525 Colshlre Drive
McLean, VA 22102
703-883-6146

Appendix A2 — Terms of Reference

2 NOV 1984

MEMORANDUM FOR CHAIRMAN, DEFENSE SCIENCE BOARD

SUBJECT: Defense Science Board Task Force on Software

You are requested to form a Task Force on Software. Software costs are projected to increase substantially in the next decade, and the cost of software development is becoming an increasing fraction of total development costs of many types of weapons systems. In addition, the testing of software to prove performance is becoming increasingly difficult and-time consuming, leading to delays in system deployment. The need for software productivity improvement is well recognized.

The task Force should address this broad problem and identify and assess the following:

A. Assess and unify the conclusions and recommendations of the various recent studies of military software problems.

B. Examine and discuss the theoretical and practical reasons that make software costs high including the design and analysis of tests.

C. The probable effectiveness of the proposed DoD STARS program at addressing military software problems, and the relative priority of the components of the STARS program (suggest alternative to STARS if necessary)

D. Ways of enlisting industry, Service laboratories, and university efforts in-programs aimed at software productivity.

E. The probable effectiveness of the STARS program and U.S. international competitiveness in software production.

F. How to use limited R&D funds to make the biggest improvement in the development of military software capabilities.

G. Implementation concepts for an incremental, evolutionary approach in case an all-out assault on the software problem cannot be funded.

Based on the above assessments, the Task Force should make specific recommendations for significant improvements in the way the DoD manages and develops software. I would appreciate a report in approximately six months.

Dr. James P. Wade, Jr., Assistant Secretary of Defense (D&S), is sponsoring this study. Dr. Frederick P. Brooks has agreed to serve as Chairman. Major Susan Swift, USAF, will serve as Executive Secretary. Commander Chris Current, USN, will be the DSB Secretariat representative. It is not anticipated that your inquiry will go into any "particular matters" within the meaning of Section 208 of Title 19, U.S. Code.

/s/ R. D. DeLauer

Enclosure
Proposed Membership

Appendix A3 — Condensed Briefings and Minutes

The complete minutes are on file in the DSB Office. The meetings arn abstracted below.

Date	Place	Subject	Briefer	Organization
11 March '85	Pentagon	Conflict of Interest	David Ream	DoD Counsel
		Army Study	LTC Sisti	USA
		AF Study	John Munson	
		STARS Program	Joseph Bats	OSD-STARS
22 April '85	Pentagon	Planning of Study		
28-29 May '85	MITRE, McLean	Navy Software Perspective	COMO Harry Quest	USN
		Tactical Flag Com. Ctr.	CMDR DeMarse	USN TFCC
		Discussion re Task Force	Dr. James P. Wade, Jr.	OUSDR&E
		Army Software Perspective	BG Alan Salisbury	USA Sys Command
		STARS Program	Joseph Bats	STARS
			Dr. Edward Lieblein	OSD
		Assessment of STARS Conf.	Dr. Barry Boehm	
		SEI/STARS Relationship	Dr. Mary Shaw	SEI
		1982 DoD Joint SW Study	Larry Druffel	Rational Technology
		AF Software Perspective	BG Dennis Brown	USAF
24 June '85	Pentagon	WIS Program	John Gilligan	Deputy SPO Director
			Gene Frank	GTE
			Don ONeil	IBM
		Executive Session	Don Latham	ASD(C3I)
			Dr. James Wade	Acting USDR&E
8 July '85	MITRE, McLean	Report Discussion	Dr. Danny Cohen	ISI
22-23 October '85	MITRE, McLean	Stategic Computing Initiative	Dr. Steve Squires	DARPA
		Strategic Defense Initiative	Maj. David Audley	SDI
		Compet. In Contracting Act	Wayne Wittig	OASD(A& L)
		STARS	Dr. Jack Kramer	OSD-STARS
26 November '85	Pentagon	DFARS 27.4	Ms. Pamela Samuelson	SEI
		SW Test & Eval. Project	Dr. Rich DeMilo	Georgia Tech
22 January '86	Pentagon	Report Discussion	Dr. Hicks, Task Force	USDR&E
4-5 March '86	MITRE, McLean	Discussion of DSB Briefing		
15 April '86	MITRE, Bedford	Report Discussion		
27 May '86	SEI	Report Discussion		

Appendix A4 — Documents Studied and References

Bailey, Elizabeth, *et al* An Assessment of the STARS Program, September-October 1985 (IDA Memorandum Report M-137)

Barbacci, M.R. *et al* "The Software Engineering Institute: Bridging Practice and Potential," *IEEE Software*, November, 1985

Boehm, Barry W. "Understanding and Controlling Software Costs," *Information Processing 86*, H.J. Kugler, ed., Amsterdam: Elsevier Science Publishers B.V. (North Holland)

Brocks, Frederick P. "No Silver Bullet," *Information Processing 86*, H.J. Kugler, ed., Amsterdam: Elsevier Science Publishers B.V. (North Holland). Reprinted in *Computer*, April, 1987

DeMillo, Richard A., *et al* Software Engineering Environments for Mission-Critical Applications - STARS Alternative Programmatic Approaches (IDA Paper P-1780) August 1984

DoD DoD Directive 5000.29 Management of Computer Resources in Major Defense Systems, 26 April 1976

Draft DoD Directive 5000.29, 15 January 1986

DoD-STD-2167 Defense System Software Development, 4 June 1985

Draft DoD-STD-2167A Defense System Software Development, 1 April 1987

DoD Directive 3405.1 Computer Programming Language Policy, 2 April 1987

DoD Directive 3405.2 Use of Ada in Weapon Systems 30 March 1987

Strategy for a DoD Software Initiative, 1 October 1982

Software Technology for Adaptable Reliable Systems (STARS) Program Strategy, 1 April 1983

STARS Implementation Approach, 15 March 1983

48 CFR Parts 214, 215, 227, and 252 Revised Defense Federal Acquisition Regulation Supplement Technical Data 10 September 1985

Druffel, Larry E., *et al* Report of the DoD Joint Service Task Force on Software Problems, 30 July 1982

Frewin, Gillian, *et al* "Quality Measurement and Modelling - State of the Art Report," REQUEST Consortium, Esprit Project ESP/800 Strasbourg, 8 July 1985

Hamblen, John W. *Computer Manpower — Supply and Demand by States*. Quad Data Corp., Tallahassee, 1984

Jones, Victor E., *et al* Final Report of the Software Acquisition and Development Working Group, July 1980

Jorstad, Norman D., *et al* Report of the Rights in Data Technical Working Group (RDTWG) 23 January 1984 (IDA Record Document D-52)

Lieblein, Edward "The Department of Defense Software Initiative — A Status Report," *Communications of the ACM, 29*, 8, August, 1986

Munson, John B., *et al* Report of the USAF Scientific Advisory Board Ad Hoc Committee on the High Cost and Risk of Mission-Critical Software December 1983

NSF "Projected Response of the Science, Engineering, and Technical Labor Market to Defense and Nondefence Needs: 1982-87." NSF Report NSF84-304. 1984

Parnas, David L. "Designing Software for Ease of Extension and Contraction", *IEEE Trans on SE, 5*, 2, March 1979, 128-138

Redwine, Samuel T., *et al* DoD Related Software Technology Requirements Practices, and Prospects for the Future (IDA Paper P-1788) June 1984

Samuelson, Pamela Toward a Reform of the Defense Department Software Acquisition Policy (Working Paper)

Selin, Ivan, *et al* Interim Report on Air Force Base Level Automation Environment, National Research Council, National Academy Press, June 1985

Taft, William H., IV Memorandum to the Joint Logistics Commanders, 12 August 1985

Vetter, Betty M. *The Technological Marketplace: Supply and Demand for Scientists and Engineers*. Scientific Manpower Commission, Washington, 1985

Vick, Charles R., *et al* "Methods for Improving Software Quality and Life Cycle Cost," AF Studies Board, National Academy Press, 1985

Weinberger, Casper W. "Remarks Delivered at the Ada Expo 1986"

Yaru, Nicholas, *et al* Army Science Board Study on Acquiring Army Software, 1983

Zracket, Charles A., *et al* "Initiatives to Improve the Development of USAF C^3I Software," MITRE, March 1984

Appendix A5 — Why Is Building Software Hard?

There are no radical breakthroughs now in view; moreover the very nature of software makes it unlikely that there will be any – no inventions that will do for software productivity, reliability, and simplicity what electronics, transistors, large-scale integration did for computer hardware.[3] We cannot expect ever to see two-fold gains every two years.

To see why this is so, and to determine what actions we must follow instead of hoping for breakthroughs, let us examine the difficulties of the software development process. We divide them into *essence*, the difficulties inherent in the nature of the software, and *non-inherent* problems, those difficulties which today attend its production but which are not inherent.

The non-inherent problems I discuss in the next section. First let us consider the essence.

The essence of a software entity is a construct of interlocking concepts: data sets, relationships aunong data items, algorithms, and invocations of functions. This essence is abstract, in that the conceptual construct is the same under many different representatiuus. It is nonetheless highly precise and richly detailed.

I believe the hard part of building software to be the specification, design, and testing of this conceptual construct itself, not the labor of representing it and testing the fidelity of the representation. We still make syntax errors, to be sure; but they are fuzz compared to the conceptual errors in most systems.

If this is true, building software will always be hard. There is inherently no magic.

Let us consider the inherent properties of this irreducible espence of modern software systems: complexity, conformity, changeability, and invisibility.

Complexity

Software entities are more complex for their size than perhaps any other human construct, because no two parts are alike (at least above the statement level). If they are, we make the two similar parts into one, a subroutine, open or closed. In this respect software systems differ profoundly from computers, buildings, or automobiles, where repeated elements abound.

Digital computers are themselves more complex than most things people build; they have very large numbers of states. This makes conceiving, describing, and testing them hard. Software systems have orders of magnitude more states than computers do.

Likewise, a scaling-up of a software entity is not merely a repetition of the same elements in larger size, it is necessarily an increase in the number of different elements. In most cases, the elements interact with each other in some non-linear fashion, and the complexity of the whole increases much more than linearly.

Many of the classical problems of developing software products derive from thia essential complexity and its non-linear increases with size. From the complexity comes the difficulty of communication among team members, which leads to product flaws, cost overruns, schedule delays. From the complexity comes the difficulty of enumerating much less visualizing, all the possible states of the program, and from that comes the unreliability. Computer programs do not break or wear out. The bugs one finds are either design flaws, implementation errors, or are the consequences of changed environments and interface. From the complexity of the functions comes the difficulty of invoking those functions, which makes programs hard to use. From complexity of structure comes the difficulty of extending programs to new functions without creating side effects. From complexity of structure come the unvisualized states that constitute security trapdoors.

[3]This appendix is an extract from "No Silver Bullet", an invited paper presented in Dublin by F. P. Brooks at the 1986 Congress of the International Federation of Information Processing. The full paper is in the Congress Proceedings.

Not only technical problems, but management problems as well come from the complexity. It makes overview hard, thus impeding conceptual integrity. It makes it hard to find and control all the loose ends. It creates the tremendous learning and understanding burden that makes personnel turnover a disaster.

Conformity

Complexity alone is nothing unique to the software discipline. Physics deals with terribly complex objects even at the "fundamental" particle level. The physicist labors on, however, in a firm faith that there are unifying principles to be found, whether in quarks or in unified field theories. Einstein repeatedly argued that there must eventually be simplified explanations of nature, because God is not capricious or arbitrary.

No such faith comforts the software engineer. Much of the complexity he must master is arbitrary complexity, forced without rhyme or reason by the many human institutions and systems to which his interfaces must conform. These differ from interface to interface, and from time to time, not because of necessity but only because they were designed by different people, rather than by God.

In many cases the software must conform because it has most recently come to the scene. In others in must conform because it is perceived as the most conformable. But in all cases, much complexity cnmes from conformation to other interfaces; this cannot be simplified out by any redesign of the software alone.

Changeability

The software entity is constantly subject to pressures for change. Of course, so are buildings, cars, computers. But manufactured things are infrequently changed after manufacture; they are superseded by later models, or essential changes are incorporated in later-serial-number copies of the same basic design. Call-backs of automobiles are really quite infrequent; field changes of computers somewhat less so. Both are much less frequent than modifications to fielded software.

Partly this is because the software in a system embodies its function, and the function is the part which most feels the pressures of change. Partly it is because software can be changed more easily – it is pure thought-stuff, infinitely malleable. Buildings do in fact get changed, but the high costs of change, understood by all, serve to dampen the whims of the changers.

All successful software gets changed. Two processes are at work. As a software product is found to be useful, people try it in new cases at the edge of, or beyond, the original domain. The pressures for extended function come chiefly from users who like the basic function amd invent new uses for it.

Successful software also survives beyond the normal life of the machine vehicle for which it is first written. If not new computers, then at least new disks, new displays, new printers come along; and the software must be conformed to its new vehicles of opportunity.

In short, the software product is embedded in a cultural matrix of applications, users, laws, and machine vehicles. These all change continually, and their changes inexorably force change upon the software product.

Invisibility

Software is invisible and unvisualizable. Geometric abstractions are powerful tools. The floor plan of a building helps both architect and client evaluate spaces, traffic flows, views. Contradictions become obvious, omissions can be caught. Scale drawings of mechanical parts and stick-figure models of molecules, although abstractions, serve the same purpose. A geometric reality is captured in a geometric abstraction.

The reality of software is not inherently embedded in space. Hence it has no ready geometric reuresentation in the way that land has maps, silicon chips have diagrams, computers have connectivity schematics. As soon as we attempt to diagram software structure, we find it to constitute not one, but several, general directed graphs, superimposed one upon another. The several graphs may represent the flow of control, the flow of data, patterns of dependency, time sequence, name-space relatioiships. These are usually not even planar, much less hierarchical. Indeed, one of the ways of establishing conceptual control over such structure is to enforce link cutting until one or more of the graphs becomes hierarchical [Parnas, 1979].

In spite of progress in restricting and simplifying the structures of software, they remain inherently unvisualizable, thus depriving the mind of some of its most powerful conceptual tools. This lack not only impedes the process of design within one mind, it severely hinders communication among minds.

Past Breakthroughs Solved Accidental Difficulties

If we examine the three steps in software technology that have been most fruitful in the past, we discover that each attacked a different major difficulty in building software, but they have been the non-inherent, not the essential, difficulties. We can also see the natural limits of each such approach.

High-Level Languages

Surely the most powerful stroke for software productivity, reliability, and simplicity has been the progressive use of high-level languages for programming. Most observers credit that development with at least a factor of five in productivity, and with concomitant gains in reliability, simplicity, and comprehensibility.

What does a high-level language accomplish? It frees a program from much of its incidental complexity. An abstract program consists of conceptual constructs: operations, data-types, sequences, and cornmmunication. The concrete machine program is concerned with bits, registers, conditions, branches, channels, disks, and such. To the extent that the high-level language embodies the constructs one wants in the abstract program and avoids all lower ones, it eliminates a whole level of complexity that was never inherent in the program at all.

The most a high-level language can do is to furnish all the constructs the programmer imagines in the abstract program. To be sure, the level of our sophistication in thinking about data structures, data types, and operations is steadily rising, but at an ever-decreasing rate. And language development approaches closer and closer to the sophistication of users.

Moreover, at some point the elaboration of a high-level language becomes a burden that increases, not reduces, the intellectual task of the user who rarely uses thc esoteric constructs.

Time-Sharing

Most observers credit time-sharing with a major improvement in the productivity of programmers and in the quality of their product, although not so large as that brought by high-level languages.

Time-sharing attacks a quite different difficulty. Time-sharing preserves immediacy, and hence enables one to maintain an overview of complexity. The slow turnaround of batch programming means that one inevitably forgets the details, if not the very thrust, of what he was thinking when he stopped programming and called for compilation and execution. This interruption of consciousness is costly in time, for one must refresh. The most serious effect may well be the decay of grasp of all that is going on in a complex system.

Slow turn-around, like machine-language complexities, is an unnecessary rather than an essential difficulty of the software process. The limits of the contribution of time-sharing derive directly. The principal effect is to

shorten system response time. As it goes to zero, at some point it passes the human threshold of noticeability, about 100 milliseconds. Beyond that no benefits are to be expected.

Unified Programming Environments

Unix and Interlisp, the first integrated programming environments to come into widespread use, are perceived to have improved productivity by integral factors. Why? They attack the incidental difficulties of using programs together, by providing integrated libraries, unified file formats, and pipes and filters. As a result, conceptual structures that in principle could always call, feed, and use one another can indeed easily do so in practice.

This breakthrough in turn stimulated the development of whole toolbenches, since each new tool could be applied to any programs using the standard formats. How much more gain can be expected from the exploding researches into better programming environments? One's instinctive reaction is that the big-payoff problems were the first attacked, and have been solved: hierarchical file systems, uniform file formats so as to have uniform program interfaces, and generalized tools. Language-specific smart editors are developments not yet widely used in practice, but the most they promise is freedom from syntactic errors and simple semantic errors.

Perhaps the biggest gain yet to be realized in the programming environment is the use of integrated database systems to keep track of the myriads of details that must be recalled accurately by the individual programmer and kept current in a group of collaborators on a single system.

Surely this work is worthwhile, and surely it will bear some fruit in both productivity and reliability. But by its very nature, the return from now on must be marginal.

Conclusion

All of the technological attacks of the software process are fundamentally limited by the productivity equation:

$$\textit{time of task} = \sum_i (\text{frequency})_i \times (\text{time})_i$$

If, as I believe, the conceptual components of the task are now taking most of the time, then no amount of activity on the task components that are merely the expression of the concepts can give large productivity gains.

We are left with the inherent task — getting the complex concepts right, and changing them correctly as the world keeps changing about them. This is a human activity, and a labor-intensive one.

Appendix A6 — Proposal for a new "Rights in Software" Clause

Porting note — Interested readers are referred to the original PDF of this report[4], or a direct search for Technical Report SEI-86-TR-2 of the Software Engineering Institute of Carnegie Mellon University, authored by Pamela Samuelson.

But I didn't feel like porting over the entire report.

[4] `https://apps.dtic.mil/dtic/tr/fulltext/u2/a188561.pdf`

Appendix A7 — A Proposal for an Ada Software Module Market

An Ada Software Module Market enterprise could perhaps operate and become viable on the following basis:

A. The ASOMM would be established for the purpose of supporting Ada through disseminating Ada modules. It would also accept Ada support tools written in other languages.

B. The ASOMM would set standards for module catalog description that specify precisely the portability properties (e.g., "compiles and operates in Unix 4.2 bsd environments, including these which we have tested: SUN, VAX11"). It might also set standards for other description attributes – accuracy, speed, function. It would set standards for the form of source code and documentation and of test cases and test drivers.

C. The owners of modules would set the prices for their offerings, but ASOMM would have a uniform set of terms and conditions, so that prospective users would have minimum paperwork.

D. ASOMM would handle all marketing, distribution, and licensing, charging a substantial (but perhaps sliding) commission on revenues.

E. ASOMM would not itself undertake module development, enhancement, documentation, repair, support, validation, certification, or warranty. It would be a mail-order software dealer.

F. All modules would include full copyrighted source, except perhaps for security submodules, which might be object-only.

G. The standard terms and conditions would include at least four kinds of licenses, at different prices:

   - One copy, limited-period trial use, price refundable except for rental fee.
   - Per machine/copy, unlimited use
   - Site linceses *[sic]*, at least for personal computers and workstations
   - Per incorporation as a component in a larger product, with free sub-licensing rights. This would enable an incorporator to do a one-time transaction and not undertake a long-run paperwork burden.

H. The standard terms and conditions would include different levels of support for licensed users (but not sub-licensees):

   - Fully supported by the owner, perhaps for an annual fee.
   - Supported by the owner on a fixed-fee-per-fix basis.
   - Support negotiable with the owner.
   - Unsupported. Caveat emptor.

I. ASOMM would maintain lists of licensed users for recall and update purposes.

J. For a modest surcharge, a user could get, along with the module, a list of the other licensed users willing to be listed.