SOFTWARE ENGINEERING NOTES

An Informal Newsletter of the SPECIAL INTEREST GROUP ON SOFTWARE ENGINEERING

Volume 10, Number 2 Apr 1985

*** SPECIAL INTEREST GROUP ON SOFTWARE ENGINEERING ***

SOFTWARE ENGINEERING NOTES (SEN) is an informal publication of the ACM Special Interest Group on Software Engineering (SIGSOFT), concerned with the design and development of high-quality software. Relevant topics include methodologies for design and implementation, programming techniques, tools, practical experience, software economics, portability of programs, program validation, quality assurance, etc. SIGSOFT seeks to address research and development issues in these areas and to provide a common ground for both, through sponsorship of conferences, symposia and workshops, organization of sessions at major conferences, and the publication of SEN.

Material of a few pages may be submitted to the editor either on-line (ARPANET [where appropriate]: NEUMANN at SRI-CSL) or camera-ready, *originals*, single-spaced. Please avoid low-resolution dot-matrix printers, and unnecessary white space. Lengthy nonphototypeset material should be double-columned on oversized mats, two columns of about 4.5" by 12.5", to be reduced by ACM to 75%. Letters, short papers, abstracts of work in progress and available reports, book reviews, specialized bibliographies, notices, news reports, etc., are all welcome. All contributions are normally taken as personal rather than organizational statements. All technical contributions are considered as unrefereed working papers. Controversy is encouraged, but not personal attacks.

Members interested in participating in SIGSOFT activities may contact the Chairman or the Vice Chairman. Persons wishing to join SIGSOFT should contact ACM headquarters, 11 West 42nd St., New York, NY 10036, 212-869-7440. Annual dues are $8 for ACM members and associates, $6 for student members, $16 for others whose major professional allegiance is outside of information processing or computing. Institutional or library subscriptions to SEN are $14 per year. (Nonspecial) back issues are $2.00 for members and $4.00 for nonmembers, as available, plus handling; please order *only* from ACM Order Dept., POBox 64145, Baltimore MD 21264.

LETTER FROM THE CHAIRMAN

From the results of our questionnaire, it seems clear to me that SIGSOFT and ***SEN*** are doing quite well as they are currently organized and oriented -- politicians make far more sweeping inferences of mandate on much more ambiguous results than we received. Peter Neumann has distilled the figures and reports on them in his column. More importantly, he has culled out the many valuable suggestions so that he can continue to keep improving the value of ***SEN*** and so that the new officers, who will be elected this spring, can use them in establishing their *regime*.

The polling of the constituency and the exchanges with Don Reifer that preceded it have only broadened my confusion about what "software engineering" is and should be. So many people seem so certain about how better to achieve the full potential of software engineering, and the organizations that support it, that I feel much more crippled than the blind men who couldn't identify the elephant. I feel that if I could only have had these other people's experiences, then things would be so much clearer; but unless there really is a second life, then that's not the way that I am going to get the insights I seek. So, all I can do is continue to muddle and muse about my own experiences and my confusion. Since this will be my last editorial as Chairman, you'll have to put up with my latest fit of befuddlement.

When I introduced a data structures course into the University of Michigan's computer science curriculum in the early 70's, the residual demand meant that I had a very broad spectrum of students, most of whom had not been brainwashed about how to approach problem-solving as one of those odd people called a computer scientist. For the final exam, I asked them to design a data structure to support a railroad ticket vending machine, and the solutions fell into three very distinct categories. One type of solution, which I'll call the *sparse* solution, was characterized by parsimony, sometimes to the nth degree as in the one that used a **very** large array and a Gödel-numbering scheme for lookup. A second type, the *heavy* solution, was like a tinker-toy bridge -- the traditionally recognized parts were all there, but some of them were too massive, there was a little more redundancy than necessary for the real thing, and while one could identify it as a bridge, one could also tell that it wasn't. The final type of solution fell in between and we can call it the *right-on* type of solution (even though we weren't using Pascal!). For the record, the *heavy* solutions were most frequent by a disturbingly large margin, and the *right-ons* outnumbered the *sparse* ones -- but only by a slim margin. I think it was at this point that I began to worry about the effectiveness of my courses!

Without exception or contradiction, the *sparse* solutions came from students of (pure) mathematics, and students from the traditional engineering disciplines (electrical engineering, civil engineering, ...) produced *heavy* solutions. Students with a background in computer science, engineering-mathematics, or non-traditional engineering (biomedical engineering, engineering physics, ...) produced all of the *right-on* solutions and a small number of the *heavy* ones.

The intent of relating this experience is not to make negative, demeaning implications about mathematics or engineering. Rather, one intent is to emphasize the well-known phenomenon that computer science problems are frequently not amenable to problem-solving techniques developed for other, more-traditional, disciplines. The other, more important, intent is to point out that our heritage determines, to an over-riding degree, the way we approach problem-solving and the general nature of our solutions. Asked to consult on increasing the productivity of a chicken farm, an engineer will think in terms of harnesses and conveyor belts, an ergonomician will think in terms of comfortable perches, a computer hacker will think in terms of fertility drugs (to decrease execution time, in case you didn't catch that one), a psychologist will think in terms of rest periods and windows to let in cheering sunlight, and a mathematician will think in terms of spherical eggs. What would prove most effective is an amalgamation of these views and, since a team would be a disaster, the farmer should look for a consultant well-versed in **all** these concerns (yes, even in the issues of the theoretical advantages of different egg shapes).

With 20/20 hindsight, heritage-dominance in problem solving is obvious. That it applies to our metaphors is the point I want to stress. It seems to me that this is a fundamental difficulty in grappling with where to go with software engineering and software engineering support/enhancement activities. We all have our metaphors to aid our thinking; but they frequently turn out to be of little help, and they sometimes really hinder, in exactly that situation where we need them most, namely communicating with others.

That's a major problem, I think, with the concept of a *software factory* that is currently used in some circles to discuss software engineering environments. When using the metaphor, one runs the risk of quickly losing part of the audience. Some may turn off immediately at the implication of sweatshop labor. Others will need to be "reminded" that a factory, in a very general interpretation, has much more than just a (semi-automated) assembly line and includes support for marketing, product-line planning, quality assurance, maintenance, product-variation development, and product engineering. Sometimes, the "reminder" stresses the imagination to the breaking point -- are we really to believe that gas stations are part of Detroit automobile factories? What we need are one-minute metaphors rather than ones requiring long explanations.

We have been explaining the engineering metaphor for fifteen years now, so there must be something wrong. I personally like to think in terms of designing and constructing software so my candidate for an alternative would be ***software architecture***. Others, I suspect, could suggest other alternatives, but I don't wish to start a naming contest. Rather, I want to be able to point out that my alternative suffers some problems of miscuing an audience, but I have found them to be fewer and less severe -- except, that is, when I am talking to a group of engineers.

And therein lies the rub -- it's the audience that determines what the appropriate metaphors are and -- to the extent that we have to talk to many diverse audiences about what we do -- then we need many, diverse metaphors. The fact that we have anointed one as **the** metaphor by basing our name on it is the problem. As computer scientists we easily understand the difference between name and value, but that's a surprisingly difficult concept for others to be comfortable with to the extent needed to follow the semantic machinations we go through to explain what software *engineering* is all about.

The focus on a single metaphor has caused us problems within our community also. It brings along with it a lot of implications about the arenas appropriate for our projects and the criteria to be used to decide what projects or activities performed and evaluate the quality of their results. I know of three software engineering environment projects that are nearly identical but are taking place in different settings -- one is being planned by an industry consortium, one is being planned by a consortium of universities, and one is being planned by a group of Government organizations. The fact that all of them are experiencing problems of justification and planning indicates to me that the organizational structures inherited from the world of "real" engineering are simply not appropriate. My intuition is that this is true for software engineering projects of all types, from pure research to final transition into practice.

So, we find ourselves in a semantic morass that affects us as a community and causes us problems in interacting with other communities. It results from retaining a name that was intended to foster lively discussion at the conference that gave birth to the field rather than intended to indicate the details of solutions to problems. It continues because the name can be taken too literally. This leads some to espouse traditional engineering practice as something appropriate for software, in the extreme being the universal medicine for all of our ills. It also leads to a debilitating *them*-and-*us* partitioning of the community, along established institutional boundaries, that leads to unnecessary controversy and inhibits us from getting on with our work and improving our technology.

I have two milestones that I think will flag significant progress in bettering the situation. One is the disappearance of articles, letters, editorials(!), and the like, that speak in terms of established organizational structures or try to argue that *software engineering* should or should not be similar to the engineering of artifacts, the cooking of a meal, the construction of a building, the writing of a book, the painting of a picture, or whatever. The other is the appearance of an organization dedicated to software engineering that is not aligned according to established organizational boundaries but which receives enthusiastic and extensive support, both moral and financial, from all sectors.

The first milestone is relatively easy to accomplish in a very short period of time. If nothing else, we can just decide, as a community, to cut it out. A more positive move would be to establish a global policy that would require everything submitted for publication to have an early section entitled **My Metaphor** and then follow this section with something substantive. If nothing follows, then it wouldn't be published, even in a letters-to-the-editor section. More extreme, if it had something of substance but didn't explain the underlying metaphor, then it wouldn't be published either. When coupled with a rule about the proportionate size of the metaphor section, this policy would help to identify the one-minute metaphors appropriate to software engineering. It would also enhance understandability by not making the reader guess at the metaphor. Nor would the reader have to waste time worrying about the accuracy of the metaphor since everyone would know it was there merely to establish a context for understanding the real meat of the publication and, therefore, not really intended to be a point at issue.

I used to be optimistic about seeing the second milestone accomplished during my lifetime, but I have become increasingly pessimistic. This isn't because of the early forties' phenomenon in which each year seems to shorten your lifetime much more than 365 days. Rather, it's because I have seen the few promising moves fail that have actually been tried in this direction -- and fail miserably. And it frequently seems that political or nontechnical concerns are the reason for failure, even in the face of extensive protestations that these sorts of concerns wouldn't get in the way.

I think the root problem is the (spoken or silent) requirement that the result and payoff be measurable in terms consistent with the evaluation criteria that come along with doing things in the context of established, engineering-oriented organizational structures. A little more altruism would help, as would a bit more of what I would call community responsibility. But the real need is for two things. First, we need a general recognition that what we call software engineering is neither characterizable nor understandable in traditional terms. Second, we need a wide-spread willingness to live with that situation for a while without trying to force a retrofit to the known world or letting our inherited biases get in the way.

William E. Riddle

Please Vote in the 1985 SIGSOFT Election.

Your vote is important. Ballots should be out soon.

If all goes well and the election is on time, the next issue should be able to announce a new slate of SIGSOFT officers.

LETTER FROM THE EDITOR

Questionnaire Results -- Methodology or Misology?

Enough results are in for us to see various patterns in your responses to the questionnaire. I summarize them -- and many of your suggestions -- beginning on Page 12, and hope you will look them over carefully. A few more general comments are also appropriate here.

- A few practitioners (as opposed to researchers) exhibit a hard-core *misology*, i.e., *a distrust or hatred of reason and reasoning*. When such a fear is deep-seated, I imagine it can be overcome only if reason someday becomes practical (e.g., cost-effective). However, clearly reason cannot be mandated!

  Diversity is vital to SIGSOFT. Our membership represents a wide variety of interests, and it is important to continue trying to recognize this. The industrial folks need to do more to unblock the log-jam of their characteristic nonparticipation accompanied by complaints that they are not represented. The researchers need to be more aware of the day-to-day worries of those producing what might be called Industrial-Strength Software. (But beware of Generic-Brand Industrial-Strength Software.)

  Many "software engineers" are unaware of both the Literature and the Illiterature. Worse yet, some of them keep creating more of the latter. We will try to help, e.g., by summarizing IEEE issues you may not get.

  Quite a few people said that even though it may fall on deaf ears, we should continue to solicit contributions from industry -- despite the characteristic lack of response. I sometimes feel we are flogging a straw herring in mid-flight, but we can't give up now.

Ode/Owed to Bill Riddle

Bill came in like a lamb, but goes out like a lion with his foregoing message. Thanks, Bill, for a very productive Chairmanship. [We hope that his successor is lion in waiting.]

Risks to the Public in Computer Systems

ACM Committee on Public Policy

Overcommitted as I am, I have taken on a new responsibility as Chairman of the ACM Committee on Computers and Public Policy. One of the major thrusts of this committee will be the establishment of a forum concerning risks to the public associated with the use of computer systems. That activity will attempt to provide a wide range of accurate information -- technical and nontechnical -- to a variety of audiences. The genesis of the effort is described in Adele Goldberg's President's Message in the February 1985 ***CACM*** (pp. 131-133). ***SEN*** will continue its interest in computer risks to the public, and begins herewith an explicit section on that subject. In addition, there are likely to be on-line message services encompassing accessible networks, providing a rapid outlet for bibliographic information, speaker bureaus, news, etc. Sponsorship of meetings, books, and articles in noncomputer publications as well as the usual computer literature is also contemplated. We hope to share information with many people. Your suggestions and contributions would be most welcome.

Not surprisingly, there were many relevant items in the last three months to report. Some of them will be familiar to Network and BBOARD junkies.

Another Robot Violates Asimov's First Law

The national Centers for Disease Control in Atlanta have reported "the first documented case of a robot-related fatality in the United States". Working with automated die-casting machinery last July, a Michigan man was pinned between the back end of a robot and a steel pole. He suffered a heart attack and died five days later. Although Asimov's Three Laws of Robotics serve well as an aid in writing science fiction (for a recent example, see The Robots of Dawn), they also provide a first approximation (albeit simplistic) to a set of requirements for designers of robotic systems. Thus, for example, the First Law would force greater attention to be paid to fail-safe mechanisms.

China Airlines' 747SP

A big story involved the China Airlines 747SP, one of whose engines failed on 19 Feb 1985. The autopilot continued to try maintaining the plane at 41,000 ft -- which was impossible anyway with only three engines at that altitude, but which was certainly hopeless after the plane began to careen downward at a very sharp angle. The other three engines then also failed (perhaps from a stall). The result was that the plane lost 32,000 feet, and was within seconds of crashing into the ocean before the pilots took control away from the autopilot and were finally able to restart the engines. Significant physical damage was done to the tail stabilizers, possibly from the flak of the landing-gear doors -- which were ripped off in the fall. The plane had been heading for Los Angeles, but landed safely in San Francisco.

It is not our intention to assess responsibility for this incident. However, we note that several different failures and design deficiencies may have been involved, illustrating the difficulties that exist in anticipating all possible *combinations* and *sequences* of unusual conditions. Perhaps the autopilot should not have been programmed to continue trying to do the impossible under such circumstances. Perhaps the pilots should somehow have been warned interactively what the autopilot was trying to do. Perhaps the pilots should have been quicker in overriding the autopilot, e.g., aided by better indicators of what was going on (shades of Three Mile Island!). Perhaps the landing gear cover should have been stronger. The point is that several contributing factors were involved, and that a chain reaction was set off. The moral for software engineers is that the entire operating environment is critical, not just a few programs, and that in emergencies, unanticipated events often seem to occur that stress the design in unanticipated ways.

The Rocket's Red Glare

As most of you read (e.g., the London Daily Express, 31 January 1985), an unarmed Soviet missile headed in the wrong direction on 28 December 1984 -- over Norway, crashing or downed in Finland, and reportedly en route to Hamburg -- allegedly resulting from the computer system being given the wrong flight path. I record this here for those of you who missed the story (and the ensuing flurry of highly contradictory follow-up stories) as just another routine instance [!] of one of your every-day harmless mishaps. [Nothing to veer but veer itself?]

Who Checks the Checker, Revisited

The San Francisco police computer system made the news for about a week running in February 1985 when the private files of the San Francisco Public Defender were found to have been easily accessible to police and prosecutors. As many as 1000 cases had been processed during that time, and could potentially have been compromised as a result. In the absence of adequate audit trails, no one is quite sure. In at least two cases, defense attorneys joined the public defenders to try to get murder charges dismissed -- although without success.

Another Leap-Year Bug

Every four years we get a few Leap-Day problems. Here is the latest, reported by Margaret Dibben, Money Editor, of **The Guardian**, Manchester, UK, on 14 Jan 1985. (It is strange that the story took almost a year to get reported.) Several people sent this one to me, the first being Brian Randell -- who netted it from Newcastle-upon-Tyne.

> Leap year may be for lovers, but 29 February 1984 caused a rift between the Midland and National Westminster banks. The cash machines at the two banks are usually happily compatible; each will obey instructions from the other's plastic card. But last Leap Year Day has come between them. Nat West's Servicetill remembered that 1984 was a Leap Year and reprogrammed to allow for one more day. Midland's AutoBank forgot. The consequence was that when January arrived AutoBank jilted the NatWest cash card, and Servicetill would not let go of the Midland cashcard. The computer got the blame. The tiff took some days to come out into the open, and is taking longer to patch up. Compatibility was lost because the information stored in the metallic stripe on the back of the plastic cards did not tally with that inside the cash dispenser machines. At this point, AutoBank was still pulling its punches. When Servicetill customers tried to get cash from AutoBank machines, AutoBank refused. It stayed tight-lipped, would not pay out the money, and spat out the plastic card. But for Servicetill, there were no holds barred. Servicetill refused to part with the money as well, but, with a streak of vindictiveness, also gobbled up the AutoBank cash card. For Midland Bank customers this is a cruel blow, because the cash card doubles as a cheque guarantee card. They started the year with no money, and with no way of using cheques either. The computer minders have now spoken to the two star-crossed lovers, and helped them patch up the quarrel. But there is one point which they have been unable to resolve. Midland Bank customers who used an AutoBank cash machine during the first week of January, before Midland reprogrammed, will still lose their card inside a Servicetill unless they first use it in an AutoBank to correct the message in the stripe.

I imagine that end of the century will produce some really astounding anomalies. We can expect lots of systems that will not have anticipated they would still be around in the year 2000. For example, they might store only the tens' and units' digits of the year, and might also be dependent on time-stamp ordering, collapsing because 99 > 00 rather than 99 < 00. (However, I hope that SIGSOFT will have found itself another editor by then to report on the excitement.)

More on Pacemakers

Our 1980 readers remember the saga of the microwave arthritis therapy that reprogrammed a man's pacemember and killed him (***SEN*** 5 1). I noticed recently that there about 100,000 new pacemakers being installed each year, and wondered what other risks might exist. (It is interesting to note that most patients dial up a monitoring service once a month or so, and have their pacemaker transmit a diagnostic sequence, although I would hope that the monitoring software is correct under all possible circumstances, and that there is no possible electronic communication in the opposite direction.)

Appropriately, Nancy Leveson (nancy@uci-icsd.ARPA) contributed the following.

> I just heard an interesting story. I was talking to somebody at the FDA and learned that there has been another case of a pacemaker accidentally reprogrammed -- in this instance it was accidentally reprogrammed by an anti-theft device in a store. Sorry, can't give you any more details because of potential lawsuits.
>
> There is an article on programmed pacemakers in a journal called PACE vol. 7 (Nov/Dec 1984, part 2).

I am again reminded of the first Sputnik, which just happened to transmit at the frequency of the early remote-control garage-door openers, opening and closing doors as it went overhead.

More From Nancy Leveson, on Living with High-Risk Technologies

There is also another report from Nancy.

> I just read a report on a new book that I think might interest you. I can only report on the report (in Technology Review, Jan. 1985) since I have not yet been able to get hold of the book itself. It is by Charles Perrow (a sociologist at Princeton or Yale or one of those schools) and is called "Normal Accidents: Living with High-Risk Technologies."
>
> According to the reviewer (Deborah Stone, a professor of political science at MIT), Perrow argues that hazards today are much higher than previously not because the technologies are more dangerous, but because the complex control systems they require are highly vulnerable to failure. Thus, he argues, we must accept the fact that catastrophic accidents are "normal" and we should be sceptical when advocates of these technologies, and the risk assessors they hire, reassure us that major accidents are extremely unlikely. He cites research evidence that undermines promises that "better management" and "more operator training" can eliminate catastrophic accidents.
>
> He goes on to say that catastrophic accidents may be normal and will probably be commonplace, but because complex technological systems are human constructions, people can also modify or eliminate them. "Ultimately, the issue is not risk, but power; the power to impose risks on the many for the benefit of the few." He criticizes the tendency of risk assessors to dismiss the public's opinions of the risks of various technologies in favor of "objective" analyses. Instead, he maintains that decisions regarding which risks society takes should be based on a democratic process.
>
> Perrow offers a model for making such decisions that tries both to limit the potential for accidents that could kill large numbers of people and to minimize the costs of abandoning high-risk technologies. He divides various technologies into three categories: (1) those with low catastrophic potential or high-cost alternatives that can be tolerated but should also be improved, (2) those with moderate catastrophic potential and moderate-cost alternatives that should be strictly regulated, and (3) those with high catastrophic potential, and with relatively low-cost alternatives, that should be abandoned and replaced. He puts nuclear power and nuclear weapons in the third group.

The Overselling of Expert Systems

And finally, still another message from Nancy. (Many thanks!)

> I suggest you look at "The Overselling of Expert Systems" by Gary Martins (he used to be a director of R&D at Rand where he was involved with AI research), *Datamation*, 1 November 1984 -- a very cogent, intelligent, well-thought-out critique and analysis of the current AI hype.

Bumpin' Grind -- Computer Turns Tables

The second performance in Baltimore of the new musical *Grind* starring Ben Vereen came to a grinding halt when the computer that operates the show's elaborate set failed 30 minutes into the first act. Director Harold Prince said this was the first time he had ever had to stop a performance of any of the 47 shows on which he has worked. He appeared stage front with the following extemporaneous speech:

> "The machine will not work. It's going to take some time -- maybe all night. It only bears out how I feel about this new mechanized world we live in. I don't think much of it."

The set, based on a 15,000 pound, three-story steel turntable, had worked properly during dress rehearsals and during a benefit performance at Baltimore's Lyric Opera House. [The automatic Grind slowed down to a drip and really turned the tables on Prince. Surprisingly there was no manual Grind on hand.]

Grindin' Bump -- Dropped Bits Drop Boulders

Joe Pistritto contributed this to Soft-Eng@MIT-XX.ARPA (see page 14).

> Well, there was this cement factory that a company [who shall remain nameless] for which I used to work built an 8080-based distributed control system. (At the time this was state-of-the-art in process control.) The plant crushed boulders into sand before mixing with other things to make cement. The conveyors to the rock crusher (and the crusher itself) were controlled by the 8080s. A batch of defective MOSTEK RAM chips used in the processor had a habit of dropping bits (no parity or ECC), causing at one point the 2nd of a series of 3 conveyors to switch off. This caused a large pile of boulders (about 6-8 feet in diameter) to pile up on top of the conveyor (about 80 feet up), eventually falling off and crushing several cars on the parking lot, and damaging a building. We noticed the problem when we couldn't explain the dull thuds we were hearing in the control room and looked out the window... You had to be there... Joe Pistritto <jcp@BRL-TGR.ARPA>

Bottled-Up Communications

The following item came from the January 16, 1985 edition of the Washington Post (page A6), and was reported by Howard Israel.

Computers Use the Phone

Calls to Coke Bottler Baffle Switchboard

> (AP) FAYETTEVILLE, N.C., Jan. 15--A municipal building was supposed to be closed and empty at night and on weekends, but computer records showed hundreds of telephone calls, some within seconds of each other, being placed from two phone extensions. It wasn't a burglar. It was two computerized Coke machines trying to phone home.
>
> City officials couldn't figure out how anyone was getting into the Sanitation and Fleet Maintenance building after hours to place the hundreds of local calls to the same number, as shown on a switchboard computer printout. Police found that the number being dialed was the local Coca-Cola Bottling Co. That led to the two Coke dispensers.
>
> Bob Johnson, who runs the service department at Coca-Cola Bottling, said that the machines were outfitted with a computer system that automatically called another computer each day, reporting how many bottles had been sold. This allowed the distributor to know when the machines needed refilling without making unnecessary stops. If the callers heard a busy signal they would disconnect and call again--and again. Johnson said the system was disconnected last week, after it was discovered that the machines had a "manufacturer's flaw."

Howard added a few comments on what made this article interesting to him:

1. When the telephone calls were discovered, a person was suspected, not a machine.
2. The "telephone computer" helped solve the mystery (computers "catching computers").
3. Records were kept on non-completed calls.

Plucked Off the Newswires

n034 1032 08 Jan 85, BC-WORDS (Newhouse 003)
Take Our Word for It column

(Note to editors: Take Our Word for It is prepared by the editors of Merriam-Webster Inc., publishers of Webster's Ninth New Collegiate Dictionary, based on questions received by Merriam-Webster.)

Dear Editor: A business associate told me that the word "bug," meaning flaw, came from an incident when a moth flew into an early computer and wreaked havoc. I find this story hard to believe. Could this be true? - T.L., Pennsylvania.

Dear T.L.: Your suspicions are correct; the computer theory of the origin of "bug" should be mothballed. Surprisingly, "bug" has been used to mean "an unexpected defect, fault, flaw or imperfection" since 1889. The earliest example in print refers to Thomas Edison's work on the phonograph, but there is no reason to believe the great inventor coined this usage. We'll probably never know just who did, or if there was an inspirational real-life bug plugging up someone's works.

Launch on Warning

The following item begins with a message on Human-Nets.

Lawsuit: Computers Cannot Declare War

By Mary Madison, Peninsula Times Tribune (Palo Alto CA), 9 March 1985

A computer manager at Stanford University has filed a lawsuit against Secretary of Defense Caspar Weinberger charging that a purported plan allowing a computer to launch a US attack on the Soviet Union is unconstitutional. Clifford Johnson, a British citizen, is challenging the concept of a Launch on Warning Capability (LOWC) system, which his complaint says can usurp the constitutional authority of the president and Congress to declare war by starting an attack of its own.

The suit argues that neither the president nor Congress may delegate their authority without limits under the constitution and certainly not to "error-prone" machines. Johnson said he learned about LOWC from stories in the New York Times and from communicating with government officials in other countries. "I'm a regular person who kept distantly abreast of the news on nuclear deployment until they put a missile in London, my hometown," he said. "Then I did some research."

The suit was filed by Johnson in the 9th US Circuit Court of Appeals in San Francisco, according to the Associated Press. Johnson is supported by a Palo Alto group called Computer Professionals for Social Responsibility. Johnson is manager of performance evaluation and capacity planning for Stanford's [ITS] computer.

His lawsuit is an appeal from a decision in US District Court where a judge ruled earlier that the judiciary should not intervene in foreign policy. However the district court judge said that the case might be reviewed in a court with "greater wisdom". The government is to reply March 20 to the brief. Federal authorities have refused to confirm or deny that LOWC exists, but Johnson claims such a system is part of the Pershing missile deployment in Europe and the proposed "Star Wars" plan.

If he wins his case against Weinberger, the secretary of defense would be required to ensure that no American land-based missiles could be launched prior to human confirmation.

"This is an attempt to undo the very first screw in the armaments race," Johnson said, "the very fact that this screw might be undone through the law would be highly significant." Johnson's complaint describes a scenario in which a North American Defense Command computer could detect something suspicious in the Soviet Union and interpret the data as indicating an imminent nuclear attack on the United States.

With just [6] minutes to decide, the computer could order America's land-based nuclear arsenal to retaliate, Johnson said. Humans, following a mandatory checklist, would then press buttons and launch missiles.

"My lawsuit is intended to stop [the government] from plugging LOWC into nuclear missiles."

Johnson, who is acting as his own lawyer in the case, filed a 50-page brief in the case. His support group, the Computer Professionals for Social Responsibility, was founded by former employees of Xerox's Palo Alto Research Center who participated in military computer research. Johnson and his group believe the Pentagon is placing too much emphasis and reliance on computers and that the technology isn't trustworthy.

If "flaky chips" launch missiles, "you've surrendered political power," he said.

When queried by netmail, Cliff added the following (although he is clearly not in a position to discuss the case in detail). Please excuse the redundancy -- it seems more appropriate for me not to edit.

Man vs. Machine -- Can Computers Legally Launch World War III?

Launch On Warning Is Unconstitutional!

On February 20, 1985, the appeal brief was filed (Johnson v. Weinberger, CA 84-2495, United States Court of Appeal for the Ninth Circuit, San Francisco) in a lawsuit asking that nuclear launch on warning capability (LOWC) be declared unconstitutional. LOWC (pronounced to rhyme with mousy) is defined as any set of procedures whereby retaliatory launching of nuclear missiles may occur in response to electronic warning of attacking missiles and prior to the conclusively confirmed commencement of hostilities with any State presumed responsible for the supposed attack. Because launch on warning response times are now under three minutes (primarily due to the land-based Euromissiles) LOWC must be fully automatic, and gives rise to a substantial probability of accidental nuclear war due to computer related error.

"It is not U.S. policy to launch on warning" said Pentagon spokesman Fred Hoffman. "I have no comment on whether or not there is such a capability." (S.F. Examiner, March 9, 1985, p.4.)

The Strategic Defense Initiative ("Star Wars") will include an automatic retaliatory launch reflex, and the temptation to plug West Germany's Pershing II deployments into it seems irresistable. The Strategic Computing Initiative, which has received full congressional funding ($600 million over a six year period) plans to develop the artificial intelligence capability necessary for LOWC.

The Constitution unequivocably mandates "The Congress shall have power ... to Declare War", and "The President shall be ... Commander in Chief of the Armed Forces." Unconstitutionality is argued on mainly the grounds that the ultimate decision to launch the first nuclear missile must be taken by the political branches, and not "delegated" to error-prone machines. In a 17-page response, Weinberger claimed the prerogative to implement automatic launch on warning. Violation of the United Nations Charter is also alleged.

The full Executive Committee of Computer Professionals for Social Responsibility has drafted a statement endorsing the complaint:

"Statement on the Possibility of Accidental Nuclear War Due to Computer Related Error

> In all but the simplest computer programs, hidden design flaws can persist, sometimes for years, even though the system appears to work perfectly. Such flaws are revealed only when the system meets a particular set of unforeseen circumstances, at which point the system may suddenly behave erratically. There exist no known methods for eliminating this uncertainty in complex computer systems. To the extent that significant decision making is handled by computers, such design flaws will contribute to inappropriate actions. In particular, a completely automated procedure for deciding to launch missiles that does not allow time for meaningful human deliberation and intervention poses the risk of an accidental launch."

After a District Court hearing, Judge Spenser Williams ruled: "it appears, and this court finds that Plaintiff raises a nonjustifiable issue under the political question doctrine, and that Defendant's motion to dismiss, as a matter of law, must be granted." The phrase "as a matter of law" acknowledges that sufficient evidence was presented in the lower court to overcome summary dismissal on the facts; indeed, Spenser Williams encouraged an appeal, concluding the hearing with the words: "what I am going to do is dismiss it, but give you the opportunity to take it before the Circuit Court of Appeals with higher authority and perhaps greater wisdom." In a lawsuit of such novelty and scope, this is as much as could have been hoped for at this stage of legal process.

ACM buffs may be interested in the one legal precedent I found prohibiting "preprogrammed" decision making where discretion is required -- a 1936 attorney general opinion (38 Op.Att.Gen. 458) requested by FDR:

> "... the authorities indicate that the president cannot delegate a discretionary duty, relieving himself of all responsibility, so that the duty *when performed* will not be his act ..."

I-Thought-You-Told-Me-It-Couldn't-Happen-Here Department?

After the 3 December 1984 chemical leak of methyl isocyanate that killed at least 2000 people at a Union Carbide plant in Bhopal, India, I recall various spokespeople for Unicarb (as opposed to Bicarb?) saying it couldn't possibly happen at the sister plant in South Charleston, West Virginia. On 7 March 1985 the West Virginia plant had a leak of 5700 pounds of a mixture that included about 100 pounds of poisonous mesityl oxide, a bigger leak than had ever occured previously at that plant (although there were no deaths this time). I mention this as another reminder symptomatic of the very common head-in-the-sand approach, sometimes also known as "nothing can go wrong [click] go wrong [click] go wrong ... ".

NEXT ISSUE. Please SENd your contributions for the next issue to me before 21 June 1985. In general, let me know before the equinox/solstice deadline if you might have something ready a little late; I can sometimes still fit it in.

Report to SIGSOFT Membership
FISCAL YEAR 1984 ACTUAL INCOME AND EXPENSES

		1stQTR	2ndQTR	3rdQTR	4thQTR	TOTAL
INCOME						
Member Dues		14912	15010	14841	15426	60189
Publications Sales		68	344	323	3660	4395
Conference Net Income		-699	0	31485	0	30786
Air Options Income		261	302	318	417	1298
TOTAL INCOME		14542	15656	46967	19503	96668
EXPENSES						
Travel and Subsistence		600	-102	83	-83	498
Temporary Help		81	0	0	0	81
Telephone		21	170	196	65	452
Office Mailing and Handling		19	95	331	97	542
Stationery and Supplies		0	0	66	68	134
Minor Printing		0	0	481	0	481
Copying and Duplicating		4	20	0	0	24
Production Printing		19298	6277	0	24900	50475
Wrapping and Handling		700	2189	0	454	3343
Shipping and Freight		2535	154	47	329	3065
Publication Postage		5093	6311	2081	896	14381
Data Processing		639	704	627	719	2689
Promotion and Advertising		0	0	0	0	0
Chapter Support		0	0	0	0	0
Meetings, Special Functions		0	171	0	0	171
Contingency Fund		0	0	8	0	8
Headquarters Allocation		6226	7145	5823	6086	25280
TOTAL EXPENSES		35216	23134	9743	33531	101624
NET		-20674	-7478	37224	-14028	-4956
FUND BALANCE	68354	47680	40202	77426	63398	63398

Pamela Zave

Pamela Zave, Secretary-Treasurer

Letter from Haim Kilov on Computer Literacy

I am incapable of refraining from remarking on *Software Engineering Notes*, volume 9, number 4, page 3, and on volume 9, number 5, pp. 5-6. However, I'd like to look at another aspect of the 'good programming' problem. Yes, you are completely right in that quite a few programmers will have to find out the hard way both the 'old problems' and their solutions. But I earlier thought that the so called 'new generation of programmers' will be much better than the old one because it will be adequately taught. Dijkstra once remarked that the replacement of generations will solve many problems. However, it seems he was over-optimistic -- and here mainly the personal computers are to blame. Now, the aspect I am speaking about is the one of adequate computing science education in general and good first computing science courses in particular. Personal computer manufacturers seem to instill in their users the belief that programming is easy, whereas a good first computing science course must instill the correct -- and opposite! -- point of view, i.e., that programming is a difficult and challenging activity, and that proper mathematical and computing science maturity is necessary for successful work in the area.

The old generation of programmers may look at such ideas with distrust, though quite a few representatives of this old generation are thinking the right way. At least, the scepticism may be understood -- all of us were badly taught and had to discover the correct approach to programming through our own experience plus through reading good books by Dijkstra, Hoare, Wirth, Gries, and others. But it seems a bit strange that the new generation repeats the old errors -- they could and should be taught adequately! When we were taught, many ideas had not emerged yet and some only made their first steps, but now computing science shows much greater maturity, and therefore absence of good textbooks and lack of good teachers seems to be inexcusable. So, where are good *first* computing science textbooks? There are very good second (and n-th) ones, e.g., *The Science of Programming* by Gries, but they usually presuppose some prerequisites. Therefore, the student in the best case will have to retrain. I am aware of two recent first computing science textbooks (both written by authors from Australia, and published by Prentice-Hall): *Computer Science: A Modern Introduction* by Goldschlager and Lister, and *How to Solve It by Computer* by Dromey -- which can be considered as very good ones. Add to them a couple of others I am unaware of (I'd like to know their authors and titles, anyway), and compare them with the enormous amount of various other computing science textbooks... The situation reminds me of the Preface to an Algol60 (very good) textbook written by Brudno in Russian in the 60s where the author, after acknowledging two or three good books, remarked that the sheer number of bad ones is too large to be mentioned. Incidentally, are the two books I am referring to known widely enough? After all, not only junior computing science students may (and need) be interested!

There is a point of view that programming literacy now is the same as reading and writing skills in the earlier times. Maybe. But this point of view might encourage the over-simplistic approach to programming in accordance to which some bright high-school students are better programmers than professional ones. Yes -- better than the so-called 'average professional programmers' they most certainly may be. But programming is too difficult, as Dijkstra had remarked, for the 'average professional programmer'. Something more is certainly needed. The problem of computer literacy is more complex.

Naturally, the 'engineering' part of software engineering often means a 'here and now' approach - but it does not mean that even then the abstraction (and thinking!) can be thrown away. On the contrary! In any case, a really professional programmer must have the desire (and time) to study programming and understand that this study requires some effort. Moreover, quiet deliberations are absolutely necessary before a programmer rushes into coding (or even specification). Here, I'd like to note a Pascal textbook where the approach is 'not to hurry' -- *Introduction to Pascal* by Welsh and Elder (Prentice-Hall) -- and therefore I liked the book.

However, there are symptoms of changes for the better. The already-mentioned good first computing science courses, some very good n-th ones, the appearance of adequate languages for beginners (e.g., one developed by CWI, formerly Mathematische Centrum, in Amsterdam) all reflect the greater maturity of computing science due to which, e.g., retraining of recent graduates, if needed, is implemented not merely due to osmosis among more competent senior colleagues, but in a more organized manner. I'd like to note here another warning by Dijkstra that teaching methodology enhances the capabilities of the already capable (the use of a good language does the same, to a lesser extent).

Two more observations from different ends of the spectrum: First, many mathematics and computing science teachers think about more proper mathematics courses for future computing scientists and make some practical steps (see e.g., *The Future of College Mathematics*, edited by Ralston and Young, Springer Verlag) -- the main idea is teaching of applicable mathematics motivated by computing science needs. And the second one is a recent positive review of Hehner's *The Logic of Programming* in a British trade paper (Computer Weekly). I quote from the review by Chris Naylor: "This is not a light bit of reading. But, like Dijkstra, Hoare, and others, it is unlikely to be safely ignored" ... Let's not only hope for the better, but if possible let's do our best in order to improve the situation.

K. Marx Str. 75
Riga 11
USSR

Haim Kilov

The Association for Computing Machinery
Special Interest Group on Programming Languages (SIGPLAN)
is pleased to announce the republication of

The *Conference Record of the* 1980 LISP Conference
held in Palo Alto, August, 1980;

The *Conference Record of the*
1982 ACM Symposium on LISP and Functional Programming
held in Pittsburgh, August, 1982.

This reprinting has been undertaken by SIGPLAN as a service to its members and professional colleagues. In the former case, it is the first time that this record has been available through ACM. (See the annotated report by G.L. Steele, Jr. [*SIGPLAN Notices* 17, 3 (March, 1982), 22-36], and alert your librarian.) This meeting was recently repeated in Austin (1984), and will be repeated again in 1986.

Copies may be ordered prepaid from:

ACM Order Department
P.O. Box 64145
Baltimore, MD 21264

	ACM Order No.	Members	All others
1980 LISP Conference	552800	$15.00	$21.00
1982 LISP & Functional Programming	552820	$18.00	$26.00
[1984 LISP & Functional Programming	552840	$20.00	$27.00]

product or is currently developing a product please send me information in the form of a informal paper for this subsection. (Please note that SIG policy does not permit the printing of commercial advertisements.) Most readers would be interested in a description of the tool or system, its capabilities, underlying philosophy, and a comparison with other existing tools or systems (research prototypes, etc.).

The first contribution to this new subsection is from Ray Houghton. Ray's "Annotated Bibliography of Recent Papers on Software Engineering Environments" should serve as an excellent starting point for anyone interested in software engineering environments. Most of the key papers, journals, and books are cited. A useful "categorization of references" is provided if you are searching for information on a particular topic area (overviews, implementation issues, etc.). Please read this bibliography and make productive use of its contents.

Annotated Bibliography of Recent Papers on Software Engineering Environments

Raymond C. Houghton, Jr.

[Alfo81] M. W. Alford and C. G. Davis. Experience with the Software Development System. In Software Engineering Environments, H. Hunke, Editor, North-Holland, 1981.

Paper presents a methodology and supporting environment for developing very large, complex, real-time systems. The environment emphasizes the discovery of errors early in the development process. The methodology consists of four major tasks: (1) Data Processing Systems Engineering (DPSE) - translate systems objectives into a consistent, complete set of subsystem functional and performance requirements (uses techniques based on verification graphs, petri nets, finite state machines, and graph models of decomposition for expressing requirements), (2) Software Requirements Engineering Methodology (SREM) - express functional and performance requirements as a graph model in Requirements Statement Language (RSL) and analyze with the Requirements Engineering Validation System (REVS), (3) process design engineering - translate requirements into a process design language, verify design, and evolve the design into code, (4) verification and validation - perform at all stages.

[Barn82] P. Barnard, N. Hammond, A. MacLean, and J. Morton. Learning and Remembering Interactive Commands. In Proceedings of Human Factors in Computer Systems, Washington, D. C. Chapter, ACM, Gaithersburg, MD, 1982.

Paper presents an experiment to determine how the choice of command names influences interactive performance of users by measuring access to an on-line help system. The authors propose that help assistance is a relatively passive cognitive strategy for learning and that it leads to less efficient operation retention if commands have general names.

[Baye81] M. Bayer, et. al. Software Development in the CDL2 Laboratory. In Software Engineering

This work was supported by the Institute for Computer Sciences and Technology, of the National Bureau of Standards under contract 415632. It is in the public domain and therefore not subject to copyright.

Environments, H. Hunke, Editor, North-Holland, 1981.

Paper presents a programming environment that supports the CDL2 programming language. The components of the system include a command interpreter, a dedicated editor/with formatting and cross-reference, a program database (underlies all tools), a file system (underlies the program database), a local and global analyzer, an intermodule interface checker, a local and global optimizer, a segment builder, an abstract code-generator, and a concrete code generator. The paper includes an in-depth discussion of the system architecture.

[Boeh84] Boehm, Barry W. et. al. A Software Development Environment for Improving Productivity. In Computer, Vol. 17, No. 6, June 1984.

Paper presents the background and status of TRW's Software Productivity System (SPS). The background includes discussion of a 1980 software productivity study and corporate motivating factors (increased demand of software, limited supply of software engineers, rising software expectations, reduced hardware costs). Based on an internal assessment, an external assessment, and a quantitative assessment, the productivity study recommends that TRW initiate a long range effort to develop a software development environment. In the short term, the study recommends the development of a prototype environment. The architecture and components of the prototype that was developed (called SPS-1) include: the work environment (improved office conditions), the hardware (a network of VAX's, LSI-11/23's, terminals, and communication equipment), a master project database (composed of a hierarchical file system, a source code control system, and a relational database), general utilities (menu, screen editor, forms package, date/time, report writer), office automation and project support (tool catalog, mail system, text editor/formatter, calendar, forms management, interoffice correspondence package), and software development tools (requirements traceability tool, SREM [Alfo81], program design langauge, Fortran-77 analyzer). Experience with the use of the prototype shows a definite improvement in productivity and also that immediate access to a good set of tools has the highest payoff.

[Bran81] Martha A. Branstad, W. Richards Adrion, and John C. Cherniavsky. A View of Software Development Support Systems. In Proceedings of NEC, Chicago, IL, October, 1981.

Paper presents views and examples of support environments and proposes four classes of support levels. The views include the toolbox approach, the VHLL (very high level language) approach, and the software development support system approach (an integrated system with an underlying database). Examples include PWB/Unix, Interlisp, Howden's environments [Howd82], and the Ada Programming Support Environment (APSE). The proposed four classes of support levels are: D1 (what most operating systems provide), D2 (the capabilities of D1 plus integration with an underlying database), D3 (D2 plus support for the entire life cycle), and D4 (D3 plus features that are current research topics). Extensions of each level are also proposed for critical applications.

[Bran81a] Martha A. Branstad and W. Richards Adrion, Editors. NBS Programming Environment Workshop Report. National Bureau of Standards, NBS SP 500-78, June 1981.

Report presents the results of the Programming Environment Workshop, Rancho Santa Fe, CA, April

1980. The goals of the workshop were to assess the current technology, indicate needed standards for tools, develop guidance for near term environment development, and determine future research directions. The workshop attendees were divided into four groups: (1) contemporary software development environments, (2) software environment research - the next five years (3) advanced development support systems, and (4) high level language programming environments. The results of each of the four groups is reported by their respective leaders, W. Howden [Howd82], L. Osterweil [Oste81], T. Standish, and M. Zelkowitz.

[Buxt80] J. Buxton. Requirements for Ada Programming Support Environments: STONEMAN. U.S. Department of Defense, Washington, DC, February 1980.

Report lays out the requirements for Ada environments. The general requirements include support for the entire lifecycle, integration of tools, and exploitation of modern hardware. The more specific requirements include: the Ada Programming Support Environment (APSE)/Minimal APSE (MAPSE)/Kernal APSE (KAPSE) model, database support, and tool support at the different APSE/MAPSE/KAPSE levels. The APSE/MAPSE/KAPSE model is a four level model where level 0 is the host level, level 1 (KAPSE) is a standard interface to level 0 that supports database interactions, communications, and run-time, level 2 (MAPSE) is a minimal tool set (editor, translator, linker, debugger, configuration manager), and level 3 (APSE) is a set of tools for full support (life cycle, documentation, and management support).

[Buxt80a] J. Buxton. An Informal Bibliography on Programming Support Environments. In ACM Sigplan Notices, Vol. 15, No. 12, December 1980.

Bibliography with brief notes and commentary on 40 papers that deal primarily with architectures of programming support environments. Included with the bibliography are short summaries of the following proposed and existing programming support environments: Cheatham's PDS Model, Cooprider's Thesis, CADES, C-MESA, Softech's MSEF, Stenning's Foundation System Model, and Tichy's Model.

[Camp84] Roy H. Campbell and Peter A. Kirslis. The SAGA Project: A System for Software Development. In Proceedings of the ACM Sigsoft/Sigplan Software Engineering Symposium on Practical Software Development Environments, Pittsburgh, PA, April 1984.

Authors discuss the SAGA Project and its current status. The project proposes to develop a lifecycle support environment for small to medium size projects. At the heart of the system is a proposed language-oriented editor generator that can synthesize a language-oriented editor for each life cycle language (i.e., requirements language, design language, implementation language, etc.). The current system is Unix based and includes a prototype for a language-oriented editor (implementation language), prototype version control components, and a production documentation tool.

[Cowe83] Wayne R. Cowell, and Leon J. Osterweil. The Toolpack/IST Programming Environment. In Proceedings of SoftFair, (IEEE Order No. 83CH1919-0), July 1983.

The paper discusses a portable, Fortran-oriented programming environment. The architecture of the environment includes a command interpreter, tool fragments (commands may invoke several tool fragments), and a virtual file system (files created by tools can be recreated by tools). The tools in the environment include: data flow analyzer, program instrumenter and debugger, documentation generation aid, program formatter, syntax-controlled editor, macro processor, text formatter, program structurer, Ratfor processor, portability checker, program transformer, and various file-handling utilities.

[Cox83] Brad J. Cox. The Message/Object Programming Model. In Proceedings of SoftFair, (IEEE Order No. 83CH1919-0), July 1983.

A tutorial on object-oriented programming, the paper discusses the operator/operand model and the message/object model. It then presents a case study developed on Smalltalk-80.

[Deli84] Norman M. Delisle, David E. Menicosy, and Mayer D. Schwartz. Viewing a Programming Environment as a Single Tool. In Proceedings of the ACM Sigsoft/Sigplan Software Engineering Symposium on Practical Software Development Environments, Pittsburgh, PA, April 1984.

Paper presents an interactive programming environment called Magpie. The user of Magpie deals with two states, the status of the source code and the status of execution. The environment features overlapping windows, a mouse pointing device, pop-up menus, a browsing capability, language-directed editing, incremental compiling, and debugging capabilities.

[Fair80] Richard E. Fairley. Ada Debugging and Testing Support Environments. In Proceedings of the ACM-Sigplan Symposium on the Ada Programming Language, December 1980 (see SIGPLAN Notices, Vol. 15, No. 11, November 1980).

A review of the requirements specified in [Buxt80] and a discussion of the issues associated with them. Analysis considerations, source level support and debugging, KAPSE considerations, and data abstractions are covered.

[Feil81] Peter H. Feiler and Raul Medina-Mora. An Incremental Programming Environment. In Proceedings of the 5th International Conference on Software Engineering, (IEEE Order No. 81CH1627-9), March 1981.

Paper reviews the support required by programming environments and the traditional method for providing this support (i.e., editing, translation, linking, loading, and debugging). It then presents the Incremental Programming Environment which features syntax-directed editing, common representation, incremental program translation, and language oriented debugging. This is followed by a discussion of design and implementation issues for such an environment.

[FIPS99] Guideline: A Framework for the Evaluation and Comparison of Software Development Tools, National Bureau of Standards FIPS PUB 99, March 1983.

Guideline presents a framework (a taxonomy) for identifying, discussing, classifying, evaluating, and comparing software development tools and environments. The taxonomy includes almost 100 features that are presented in a hierarchy. At the top level, features are divided into input/output categories or function categories. The functions include transformation, static analysis, dynamic analysis, and management. The appendix includes a set of event sequences for the acquisition of tools.

[Fisc84] C. N. Fischer, et. al. The Poe Language-Based Editor Project. In Proceedings of the ACM Sigsoft/Sigplan Software Engineering Symposium on Practical Software Development Environments, Pittsburgh, PA, April 1984.

Paper presents an overview of POE, a Pascal programming environment, and presents some of the technical issues associated with the development of the environment.

[Gutz81] Steve Gutz, Anthony I. Wasserman, and Michael J. Spier. Personal Development Systems for the Professional Programmer. In Computer, Vol. 14, No. 4, April 1981.

This paper reviews the problems with existing development environments, proposes a programmer's personal machine, and examines the advantages and disadvantages of such a machine. The proposed programmer's personal machine consists of: (1) an intelligent terminal with 1 Meg local storage, CPU and address space equivalent to a 32-bit mini, graphics capability, (2) hard disk (40 Megs) and floppy disk, (3) networking capability (with other PPBS's), (4) audio input/output, (5) pointing device (mouse, tablet, or light pen), and (6) capability to add more memory and other devices (e.g. a quality printer). The potential advantages of such a machine include constant response time, a comprehensive set of tools, less dependence on a single machine, integration of software development and office automation. The potential disadvantages include tool expense, training expense, communications, device dependence.

[Guya84] Jacques Guyard and Jean-Pierre Jacquot. MAIDAY: An Environment for Guided Programming with a Definitional Language. In Proceedings of the 7th International Conference on Software Engineering, (IEEE Order No. 84CH2011-5), March 1984.

Paper discusses an environment under development that is oriented around an object-oriented language and an algorithm design methodology. The environment enforces the methodology through a set of control functions. The user views a development session through a set of windows that present the development level, messages, the object being defined, objects remaining to be defined, the stored algorithm, and the current command.

[Hall80] Dennis E. Hall, Deborah K. Scherrer, and Joseph S. Sventek. A Virtual Operating System. In Communications of the ACM, Vol. 23, No. 9, September 1980.

Paper presents a Unix-like environment that can be implemented on top of a vendor-supplied operating system to make in-effect a virtual operating system. The environment consists of four layers: (1) the vendor-supplied operating system (the innermost layer), (2) the virtual machine (consisting of primitives such as opening and closing files, reading and writing to files), (3) utilities (a set of tools written in portable Fortran including Kernighan and Plauger's Software Tools), and (4) an integrated command interface.

[Haus81] Hans-Ludwig Hausen and Monika Mullerburg. Conspectus of Software Engineering Environments. In Proceedings of the 5th International Conference on Software Engineering, (IEEE Order No. 81CH1627-9), March 1981.

A paper which discusses the issues associated with the coverage of software engineering environments. The issues include support for full life cycle development, quality assurance, product control, management, specific applications, specific methodologies, and representation schemes. Also discussed are issues related to the integration of tools and motivations for developing environments. The appendix defines the criteria that must be met for a system to be considered an environment. These include: a software engineering orientation, the use of at least one recognized scientific concept, applicability to more than one life cycle phase, and some level of tool integration. The authors present short summaries of environments that they feel meet these criteria. They include AIDES, APSE, ARGUS, ASSET, CADES, CDL2-Lab, COSY, DREAM, Gandalf, Gypsy, HDM, ISES, PWB/Unix, SDEM/SDSS, REVS, and SEF.

[Houg82] Raymond C. Houghton, Jr. A Taxonomy of Tool Features for the Ada Programming Support Environment (APSE). National Bureau of Standards, NBSIR 82-2625, December 1982.

A review of the APSE [Buxt80], the ALS [Wolf81], and the AIE (the Navy's Ada Integrated Environment) based on [FIPS99]. The taxonomy includes a comparison of features in the areas of management, static analysis, dynamic analysis, transformation, and input/output. A set of underlying tool primitives is defined that support these features.

[Houg84] Raymond C. Houghton, Jr. Online Help Systems: A Conspectus. In Communications of the ACM, Vol. 27, No. 2, February 1984.

Paper discusses online assistance that is provided by various types of environments. It includes a discussion of the types of assistance and the issues associated with the development of online help systems.

[Howd82] William E. Howden. Contemporary Software Development Environments. In Communications of the ACM, Vol. 25, No. 5, May 1982.

Paper proposes four levels of tool support that could be provided by software engineering environments. For each level, the type of project, the estimated cost, and the support provided is detailed. For example, environment I has an estimated cost of $35K and is for medium-sized projects, while environment IV has an estimated cost of $3M and is for large scale projects. Requirements, design, coding, verification, and management tools and techniques are presented for each environment level.

[Huff81] Karen E. Huff. A Database Model for Effective Configuration Management in the Programming Environment. In Proceedings of the 5th International Conference on Software Engineering, (IEEE Order No. 81CH1627-9), March 1981.

Paper addresses configuration mangement issues (i.e., configuration identification and configuration control) in a software engineering environment and presents a model for effectively handling them.

[Hunk81] H. Hunke, Editor, Software Engineering Environments, North-Holland, 1981.

Book contains the proceedings of a symposium held at Lahnstein, Federal Republic of Germany, June 1980. Some of the papers include [Snow81], [Baye81], [Ridd81], [Alfo81], and [Mats81]. Papers related to some in the book are [Tayl84], [Stuc83], [Buxt80], and [Kern81]. Other papers discuss issues and tools related to software engineering environments including functional aspects of environments, computer aided design, support for concurrent and distributed systems, human factors, description languages, productivity, formal verification, performance, system decomposition, and version control. The book concludes with a bibliography by Hausen, Mullerburg, and Riddle that contains more than 350 citations from 1968 to 1980.

[Kern81] Brian W. Kernighan and John R. Mashey. The Unix Programming Environment. In Computer, Vol. 14, No. 4, April 1981.

A paper which extols the benefits of the Unix programming environment. It reviews the underlying interface, i.e., the hierarchical file system and the seven primitive functions: open, create, read, write, seek, close, and unlink. It reviews the overlying interface (the user interface), i.e., available tools, input/output redirection, and various operators available to the user. It then discusses how a user can avoid programming by building a shell procedure of simpler tools available on the system. Finally, the attributes of the system are discussed, e.g., support for medium size projects, support primarily for the latter stages of software development, loose integration of tools and facilities, general support for all applications.

[Kuo83] Jeremy Kuo, Jay Ramanathan, Dilip Soni, and Markku Suni. An Adaptable Software Environment to Support Methodologies. In Proceedings of SoftFair, (IEEE Order No. 83CH1919-0), July 1983.

Paper describes an environment that can be tuned to support different software development methodologies. The control mechanism is based on the gathering of project data through a forms-based interface. The forms are defined at the start of development.

[Lebl84] David B. Leblang and Robert P. Chase, Jr. Computer-Aided Software Engineering in a Distributed Workstation Environment. In Proceedings of the ACM Sigsoft/Sigplan Software Engineering Symposium on Practical Software Development Environments, Pittsburgh, PA, April 1984.

Paper discusses an Apollo-based distributed software engineering environment. Because instances of the system can be running at various nodes in the environment, the system consists primarily of managers that keep track of what is going on. The managers include: a history manager (source code control), a configuration manager (version control), a task manager (tracks interrelationships among development products), monitor manager (watches user-defined dependencies), and an advice manager (tracks general project information).

[Love83] Tom Love. Experiences with Smalltalk-80 for Application Development. In Proceedings of SoftFair, (IEEE Order No. 83CH1919-0), July 1983.

Paper extols the benefits of the single-user, single-language environment called Smalltalk-80. An example is presented of a graphics program that was developed using the object-oriented development methodology that is supported by the system. The "mode-less" user interface and the performance benefits of the interface structure and mouse are also discussed.

[Mage84] Kenneth Magel. Principles for Software Environments. In ACM Software Engineering Notes, Vol. 9, No. 1, January 1984.

Paper which lists and discusses a set of environment principles that include the following: generality (full life cycle support), adaptability (portability), user orientation (designed for a specific community), tailorability (adaptable to many types of interface devices), extensibility (new tools can be added), consistency (consistent use from part of the system to another), unification (user can anticipate how unfamiliar tools will operate), abstraction (hide as many details as possible), aggregation (bigger tools from smaller ones), incremental preparation, efficiency, predictability, subsetable, ability to group resources, and recoverability.

[Mats81] Y. Matsumoto, et. al. SWB System: A Software Factory. In Software Engineering Environments, H. Hunke, Editor. North-Holland, 1981.

Paper discusses a large scale software factory that consists of three physical buildings, 2000 employees, a methodology, a software environment, and management and quality control. The software products are for critical applications (nuclear power stations) and there is an emphasis on using reusable code. The software environment consists of a number of tools and techniques that emphasize the latter part of the life cycle (language and library processors, editors, debuggers, etc.). The plans for the environment include the addition of tools for the front end (SADT, design languages, etc.).

[Metz83] J. J. Metzger and A. Dniestrowski. PLATINE: A Software Engineering Environment. In Proceedings of SoftFair, (IEEE Order No. 83CH1919-0), July 1983.

Paper describes an environment that consists of a methodology (the PLATINE methodology) and a set of tools (the PLATINE tools). The environment supports the entire life cycle, is adaptable to product size, supports different types of users, and supports host/target development. The methodology consists of defining a software structure hierarchy (software structuration) which produces typed abstract objects which are then associated with elements (source, listing, binary, map, nomenclature, or status). The methodology also includes the production of software (merging of the elements), project management, and evolution. The user interface consists of a command language and a set of full screen panels. The tools include LSTR (specification of real time embedded systems, derived from PSL/PSA), SDL (system design representation), Metacomp (YACC like), EPCS (a project management tool based on DEC's project control system), a formatter (DEC's runoff), a screen editor (DEC's EDT), documentor (editor from source code), mail (DEC's), crossrf (data dictionary cross reference), complex (a complexity measure), a configuration controller, and a comparator.

[Tomo84] Tomoharu Mohri et. al. PDAS: An Assistant for Detailed Design and Implementation of Programs. In Proceedings of the 7th International Conference on Software Engineering, (IEEE Order No. 84CH2011-5), March 1984.

Paper presents an environment that uses a forms-oriented approach to standardize document format and to prevent inconsistencies between documents and programs. There are 10 types of forms for design which are based on a forms-oriented language. The system structure consists of a forms-oriented editing subsystem, a document generation subsystem, a program construction subsystem (generation is based on module algorithm descriptions), a design database, and a component database (interchangable program components). An interesting aspect of the environment is automatic Japanese to English translation from algorithm descriptions.

[Oste81] Leon Osterweil. Software Environment Research: Directions for the Next Five Years. In Computer, Vol. 14, No. 4, April 1981.

Paper discusses research issues associated with software engineering environments, in particular, the breadth of scope and applicability, user friendliness, reusability of internal components, tight integration of tool capabilities, and use of a central database. A five-year research plan is presented which includes studies of existing support systems, tool fragment studies, data base studies, construction, and test beds for configuring environments.

[Oste82] Leon J. Osterweil. Toolpack - An Experimental Software Development Environment Research Project. In Proceedings of the 6th International Conference on Software Engineering, (IEEE Order No. 82CH1795-4), September 1982.

An implementation of [Oste81]. Paper presents an environment under development that concentrates on tight integration of tool capabilities (use of tool fragments) and an underlying central database (virtual file system). See also [Cowe83].

[Pren81] Dan Prentice. An Analysis of Software Development Environments. In ACM Sigsoft Software Engineering Notes, Vol. 6, No. 5, October 1981.

A paper which emphasizes the problems. The issues discussed include lack of hardware support, high cost, user resistance to change, and poor user interfaces.

[Ridd81] W. E. Riddle. An Assessment of Dream. In Software Engineering Environments, H. Hunke, Editor, North-Holland, 1981.

Paper reviews the DREAM system. DREAM is oriented to the development of concurrent systems using DREAM Design Notation (DDN), a language that can be used to model a total system including hardware, software, and wetware (people, etc.). The model reflects the externally observable characteristics of a system and is an adequate basis for preparing implementation plans. The DREAM system tools include a data base core that stores DDN fragments, bookkeeping tools (entry and retrieval), and decision-making tools for paraphrasing (a re-structured presentation), extraction (simulation), and consistency checking. The paper concludes with lessons learned and problems for the future.

[Ridd83] William E. Riddle. The Evolutionary Approach to Building the Joseph Software Development Environment. In Proceedings of SoftFair, (IEEE Order No. 83CH1919-0), July 1983.

Paper reviews an effort to build an environment that was cut short due to the closing of Cray Labs. The Joseph environment was 30% completed. The paper includes a user scenario of the proposed environment which includes cabilities to view database information, extract database information, perform notation-directed editing, analyze changes, and deposit information into the database. The paper then discusses the work that was accomplished which includes a layered environment with Unix at the core, a set of integrated tools in the next layer (the crypt), and a requirements definition tool and a design definition tool in the outer layer (pharaoh and oasis). Pharaoh and oasis include viewing, notation-directed editing, and version control capabilities of requirements and design specifications. They use a notation that consists of key-words and description fragments.

[Rube83] Burt L. Rubenstein and Richard A. Carpenter. The Index Development Environment Workbench. In Proceedings of SoftFair, (IEEE Order No. 83CH1919-0), July 1983.

Paper presents a methodology and an associated environment for building application systems (informations systems for business applications). The methodology divides an application system into a dialogue manager, a database processer, an output processor, an action processer, an extended data dictionary, and a control monitor. The environment includes facilities for data dictionary specification, structured graphics, screen definition, output processing (for developing mock-ups), defining control between components, and generic data entry. An example of the use of the system is presented.

[Sava82] Ricky E. Savage, James K. Habinek, and Thomas W. Barnhart. The Design, Simulation, and Evaluation of a Menu Driven User Interface. In Proceedings of Human Factors in Computer Systems, Washington, D. C. Chapter, ACM, Gaithersburg, MD, 1982.

Paper discusses experiments relating to the user interface of an environment. An analysis of human errors led to the design of a system that provided an extensive hierarchy of menus for the inexperienced user and a variety of shortcuts to system functions for the experienced user.

[Shne80] Shneiderman, B. Software Psychology: Human Factors in Computer and Information Systems. Winthrop, Cambridge, Mass., 1980.

Book discusses a broad range of issues related to human factors in computers. Of particular interest to the developers of software engineering environments are the chapters on interactive interface issues and designing interactive systems. These chapters cover the user interface (control, response time, text editing, menu selection, error handling), the goals for interactive systems (simplicity, power, user satisfaction, reasonable cost), and the design process (from a human factors standpoint).

[Snow81] R. A. Snowdon. CADES and Software System Development. In Software Engineering Environments, H. Hunke, Editor, North-Holland, 1981.

A review of one of the early large scale software engineering environments. The paper presents a history of CADES dating back to the early 1970's. CADES was established to provide a mechanism by which information relating to the structural model of a system could be made available to system designers early in the development process. Underlying CADES is a hierarchical database called PEARL. The establishment of CADES led to the development of a structural modeling methodology: isolate functions, data, and constraints, produce data tree, produce function (holon) tree, consider next level of detail, code in Systems Description Language (SDL), and compile. Although CADES was developed to support the earlier phases of development, the author claims that CADES solutions are always sought for development or production problems and there is an increasing trend towards support for implementation.

[Solo84] Elliot Soloway. A Cognitively-Based Methodology for Designing Languages/Environments/Methodologies. In Proceedings of the ACM Sigsoft/Sigplan Software Engineering Symposium on Practical Software Development Environments, Pittsburgh, PA, April 1984.

A paper that discusses issues relating to use of an environment. In particular, the author claims that environments should be developed based on a methodology that derives design implications based on tested hypotheses of why software developers work the way they do.

[Sten81] Vic Stenning, et. al. The Ada Environment: A Perspective. In Computer, Vol. 14, No. 6, June 1981.

A paper that reviews the objectives (i.e., life-cycle support, open-ended environment, support for Ada, configuration control, project team support, portability, and host characteristics) and the architecture (i.e., the KAPSE/MAPSE/APSE approach, the database, KAPSE functions, the user interface, intertool interfaces, and tools) of the Ada Programming Support Environment (APSE).

[Steu84] H. G. Steubing. A Software Engineering Environment (SEE) for Weapon System Software. In IEEE Transactions on Software Engineering, Vol. SE-10, No. 4, July 1984.

Paper presents a large scale environment called FASP that is hosted on multiple, large scale commercial computers. FASP primarily supports the latter stages of software development, but the extension to the requirements and design phase is discussed. The author attributes a two-fold increase in lines per month to FASP and an increase in software quality due to the tools, standards, and testing associated with the environment. The environment includes an underlying database made up of libraries: source library, object library, test library, interface library, production data library, and documentation library. The system is command driven where the commands consist of lower level system commands (command procedures). Testing is supported by the ATA (Automated Testing Analyzer) and there is support for multilanguages and multitarget computers.

[Stuc83] Stucki, Leon G. What about CAD/CAM for Software? The ARGUS concept. In Proceedings of Soft-Fair, (IEEE Order No. 83CH1919-0), July 1983.

Paper proposes that software can be developed using a CAD/CAM approach with the aid of a software engineering environment. An overview of such an environment (ARGUS) is presented. ARGUS is a micro-based environment that was built on top of Unix. Argus is menu driven with a single key stroke approach. Six toolboxes are available at the top level; they are the management toolbox (scheduling tools, action item tracking tool, electronic spread sheet, and phone list update and retrieval system), the designer's toolbox (kernel CAD/CAM capabilities with a graphics/forms based approach), the programmer's toolbox (language-based, project-specific code template capability provided by a customizable editor and language specific syntax generation macros), the verifier's toolbox (analysis tools), and general/utility tools (general editing and communication tools). A noted CAD/CAM feature of ARGUS is the automatic projection of data to documentation and code templates from the underlying database.

[Tayl84] Richard N. Taylor and Thomas A. Standish. Steps to an Advanced Ada Programming Environment. In Proceedings of the 7th International Conference on Software Engineering, (IEEE Order No. 84CH2011-5), March 1984.

Paper presents a research environment for exploring concepts and issues related to software engineering environments in general and the Ada programming language in particular. The environment called Arcturus currently includes interactive Ada (a Pascal superset), template-assisted editing, performance measurement (histograms or color), mixed compilation and interpretation, and an Ada program design language. Some concepts and issues being explored include complexity (does it scale up?), AVOS (Ada Virtual Operating System, i.e. an Ada command language), user interface issues (an Ada shell), mixing interpretation and compilation, layered architecture (i.e., device level, user interface level, tool level, foundation level), and analysis, testing, and debugging of tasking programs.

[Teit81] Warren Teitelman and Larry Masinter. The Interlisp Programming Environment. In Computer, Vol. 14, No. 4, April 1981.

Paper presents a look at the Interlisp environment. Interlisp is an environment for users of Lisp (a non-procedural list processing language). The environment is very much language dependent and is intended for use by Lisp experts. Some representative facilities in Interlisp include: file package, masterscope (helps analyze the scope of a change), DWIM (do-what-I-mean spelling corrector), iterative expressions, and the programmer's assistant.

[Teit81a] T. Teitelbaum and T. Reps. The Cornell Program Synthesizer: A Syntax-Directed Programming Environment. In Communications of the ACM, Vol. 24, No. 9, September 1981.

Paper discusses an interactive programming environment with facilities to create, edit, execute, and debug programs written in a subset of PL/I. Editing is syntax-directed with underlying tree structures, predefined templates, and phrases to fill the templates. Execution of programs can be gear-shifted forward or backward with various controls on speed.

[Teit84] W. Teitelman. A Tour Through Cedar. In Proceedings of the 7th International Conference on Software Engineering, (IEEE Order No. 84CH2011-5), March 1984.

Paper presents the facilities of the programming environment called Cedar. The Cedar environment emphasizes the use of parallel operation, multiple windows on a screen, and user interaction with a mouse pointing device. The environment supports the use of an "industrial strength" pascal-like programming language. The tour makes stops at the display (bitmapped and object-oriented with the use of icons), viewer window package (supports multiple levels of windows which are tiled on the screen), whiteboards (work windows), tioga editor and document preparation system (supports tree structured documents, editing with the mouse, syntax-directed templates), user executive (programming interface), interpreter (for debugging), automatic storage management (garbage collector), rope (string) interface, bug tracker, electronic mail, support for parallelism, and icon editor (pixel oriented, graph editor).

[Wass81] Wasserman, Anthony I. Tutorial: Software Development Environments, (IEEE Order No. EH0187-5), 1981.

Tutorial is a reference collection of 39 papers including most of the landmark papers on software engineering environments.

[Wass83] Wasserman, Anthony I. The Unified Support Environment: Tool Support for the User Software Engineering Methodology. In Proceedings of SoftFair, (IEEE Order No. 83CH1919-0), July 1983.

Paper presents an overview of the User Software Engineering methodology and the tools in the environment that support the methodology. The methodology involves users in the early stages of development and addresses user interactions with information systems. The tools in the USE environment include: the Troll relational database (underlies and is used by other tools), RAPID (rapid prototyping tool oriented to the development of information systems), PLAIN (a procedural language oriented to the development of information systems), Focus (screen-oriented editor and browser, and IDE (a software management and control tool).

[Wert82] Harald Wertz. The Design of an Integrated, Interactive, and Incremental Programming Environment. In Proceedings of the 6th International Conference on Software Engineering, (IEEE Order No. 82CH1795-

4), September 1982.

A paper that presents the details of a proposed environment that integrates editing, executing, and annotating programs.

[Wirt81] N. Wirth. Lilith: A Personal Computer for the Software Engineer. In Proceeding of the 5th International Conference on Software Engineering, (IEEE Order No. 81CH1627-9), March 1981.

Paper discusses the development, features, and architecture of the Lilith programming environment for Modula-2. The system provides a high bandwidth between the user and the system partly through the use of a mouse pointing device and the hardware structure. The display is suitable for text, diagrams, or graphics.

[Wolf81] Martin I. Wolfe, et. al. The Ada Language System. In computer, Vol. 14, No. 6, June 1981.

Paper discusses the Ada Language System which is currently under development at SofTech. The system will provide capabilities at the MAPSE level [Buxt80]. Issues relating to the development of an Ada compiler are also discussed.

APPENDIX A

Categorization of References

Overview of software engineering environments:

[Bran81]	[Bran81a]
[Buxt80a]	[FIPS99]
[Haus81]	[Howd82]
[Hunk81]	[Oste81]
[Wass81]	

Issues in building software engineering environments:

[Barn82]	[Gutz81]
[Houg84]	[Huff81]
[Mage84]	[Pren81]
[Sava82]	[Shne80]
[Solo84]	

General software engineering environments:

[Boeh84]	[Hall80]
[Kern81]	[Kuo83]
[Lebl84]	[Mats81]
[Metz83]	[Tomo84]
[Ridd83]	[Steu84]
[Stuc83]	

Systems development environments:

[Alfo81]	[Ridd81]
[Rube83]	[Snow81]
[Wass83]	

Programming environments:

[Baye81]	[Buxt80]
[Camp84]	[Cowe83]
[Cox83]	[Deli84]
[Fair80]	[Feil81]
[Fisc84]	[Guya84]

[Houg82]	[Love83]
[Oste82]	[Sten81]
[Tayl84]	[Teit81]
[Teit81a]	[Teit84]
[Wert82]	[Wirt81]
[Wolf81]	

APPENDIX B

General References on Software Engineering Environments

1. Branstad, Martha A. and W. Richards Adrion, Editors. NBS Programming Environment Workshop Report. National Bureau of Standards, NBS SP 500-78, June 1981.

2. Hunke, H., Editor, Software Engineering Environments, North-Holland, 1981.

3. Proceedings of the 5th International Conference on Software Engineering, (IEEE Order No. 81CH1627-9), March 1981.

4. Proceedings of the 6th International Conference on Software Engineering, (IEEE Order No. 82CH1795-4), September 1982.

5. Proceedings of the 7th International Conference on Software Engineering, (IEEE Order No. 84CH2011-5), March 1984.

6. Proceedings of the ACM Sigsoft/Sigplan Software Engineering Symposium on Practical Software Development Environments, Pittsburgh, PA, April 1984.

7. Proceedings of SoftFair, (IEEE Order No. 83CH1919-0), July 1983.

8. Wasserman, Anthony I., Guest Editor, Special Issue on Programming Environments, Computer, Vol. 14, No. 4, April 1981.

9. Wasserman, Anthony I., Tutorial: Software Development Environments, (IEEE Order No. EH0187-5), 1981.