Software Aging

Invited Plenary Talk

David Lorge Parnas

Communications Research Laboratory
Department of Electrical and Computer Engineering
McMaster University, Hamilton, Ontario, Canada L8S 4K1

ABSTRACT

Programs, like people, get old. We can't prevent aging, but we can understand its causes, take steps to limits its effects, temporarily reverse some of the damage it has caused, and prepare for the day when the software is no longer viable. A sign that the Software Engineering profession has matured will be that we lose our preoccupation with the first release and focus on the long term health of our products. Researchers and practitioners must change their perception of the problems of software development. Only then will Software Engineering deserve to be called Engineering.

1 What nonsense!

I can easily imagine the reaction of some computer scientists to the title of this paper.

> "Software is a mathematical product; mathematics doesn't decay with time. If a theorem was correct 200 years ago, it will be correct tomorrow. If a program is correct today, it will be correct 100 years from now. If it is wrong 100 years from now, it must have been wrong when it was written. It makes no sense to talk about software aging."

Like many such statements, the imagined quote is true but not really relevant. Software products do exhibit a phenomenon that closely resembles human aging. Old software has begun to cripple its once-proud owners; many products are now viewed as a burdensome legacy from the past. A steadily increasing amount of effort is going into the support of these older products. Like human aging, software aging is inevitable, but like human aging, there are things that we can do to slow down the process and, sometimes, even reverse its effects.

Software aging is not a new phenomenon, but it is gaining in significance because of the growing economic importance of software and the fact that increasingly, software is a major part of the "capital" of many high-tech firms. Many old software products have become essential cogs in the machinery of our society. The aging of these products is impeding the further development of the systems that include them.

The authors and owners of new software products often look at aging software with disdain. They believe that, if the product had been designed using today's techniques, it wouldn't be causing problems. Such remarks remind me of a young jogger scoffing at an 86 year old man (who, unknown to the jogger, was a champion swimmer into his 50's) and saying that he should have had more exercise in his youth. Just as we will all (if we are lucky) get old, software aging can, and will occur in all successful products. We must recognise that it will happen to our products and prepare for it. When old age arrives, we must be prepared to deal with it.

The purpose of this paper is to explain how an abstract, mathematical product can age and then to review some of the approaches to dealing with it.

2 The causes of software aging

There are two, quite distinct, types of software aging. The first is caused by the failure of the product's owners to modify it to meet changing needs; the second is the result of the changes that are made. This "one-two punch" can lead to rapid decline in the value of a software product.

0270-5257/94 $3.00 © 1994 IEEE

2.1 Lack of movement

Over the last three decade, our expectations about software products has changed greatly. I can recall the days when a programmer would "patch" a program stored on paper tape by using glue and paper. We were all willing to submit large decks of cards and to wait hours or days for the job to compile and run. When interactive programming first came in, we were willing to use cryptic command languages. Today, everyone takes on-line access, "instant" response, and menu-driven interfaces for granted. We expect communications capabilities, mass on-line storage, etc. The first software product that I built (in 1960) would do its job perfectly today (if I could find a Bendix computer), but nobody would use it. That software has aged even though nobody has touched it. Although users in the early 60's were enthusiastic about the product, today's users expect more. My old software could, at best, be the kernel of a more convenient system on today's market. Unless software is frequently updated, it's user's will become dissatisfied and they will change to a new product as soon as the benefits outweigh the costs of retraining and converting. They will refer to that software as old and outdated.

2.2 Ignorant surgery

Although it is essential to upgrade software to prevent aging, changing software can cause a different form of aging. The designer of a piece of software usually had a simple concept in mind when writing the program. If the program is large, understanding that concept allows one to find those sections of the program that must be altered when an update or correction is needed. Understanding that concept also implies understanding the interfaces used within the system and between the system and its environment.

Changes are made by people who do not understand the original design concept almost always cause the structure of the program to degrade. Under those circumstances, changes will be inconsistent with the original concept; in fact, they will invalidate the original concept. Sometimes the damage is small, but often it is quite severe. After those changes, one must know both the original design rules, and the newly introduced exceptions to the rules, to understand the product. After many such changes, the original designers no longer understand the product. Those who made the changes, never did. In other words, *nobody* understands the modified product. Software that has been repeatedly modified (maintained) in this way becomes very expensive to update. Changes take longer and are more likely to introduce new "bugs". Change induced aging is often exacerbated by the fact that the maintainers feel that they do not have time to update the documentation. The documentation becomes increasingly inaccurate thereby making future changes even more difficult.

3 Kidney failure

A problem that is often confused with, but is distinct from, software aging, is the system slow down caused by failure to release allocated memory. Files may grow and require pruning. Sometimes a memory allocation routine may not release all the space that has been allocated. Slowly, swap and file space are diminished and performance degrades. This problem is often a congenital design failure and can strike at any age; but it may also be the result of ignorant surgery or exacerbated by changing usage patterns. Nonetheless, it is more easily cured than the "aging" that is the subject of this paper. A dialysis type process may intervene and clean up the file system and memory, improved routines may cause the cleanup to occur rapidly and the software may be considered completely "cured".

4 The costs of software aging

The symptoms of software aging mirror those of human aging: (1) owners of aging software find it increasingly hard to keep up with the market and lose customers to newer products, (2) aging software often degrades in its space/time performance as a result of a gradually deteriorating structure, (3) aging software often becomes "buggy" because of errors introduced when changes are made. Each of these results in real costs to the owner.

4.1 Inability to keep up

As software ages, it grows bigger. This "weight gain" is a result of the fact that the easiest way to add a feature, is to add new code. Modifying existing code to handle the new situations is often difficult because that code is neither well-understood nor well-documented. As the size of a program increases, sometimes by one or two orders of magnitude over a period of several years, changes become more difficult in a variety of ways. First, there is more code to change; a change that might have been made in one or two parts of the original program, now requires altering many sections of the code. Second, it is more difficult to find the routines that must be changed. As a result, the owners are unable to add new features quickly enough. Customers may switch to a younger

product to get those features. The company experiences a notable drop in revenue; when they bring out a new version, it is of interest to a dwindling customer base. If they do attempt to keep up with the market, by increasing their work force, the increased costs of the changes, and the delays, lead to further loss of customers.

4.2 Reduced performance

As the size of the program grows, it places more demands on the computer memory, and there are more delays as code must be swapped in from mass storage. The program responds more slowly; customers must upgrade their computers to get acceptable response. Performance also decreases because of poor design. The software is no longer well understood and changes may adversely affect performance. A younger product, whose original design reflected the need for recently introduced features will run faster or use less memory.

4.3 Decreasing reliability

As the software is maintained, errors are introduced. Even in the early years of the industry, observers were able to document situations in which each error corrected introduced (on average) more than one error. Each time an attempt was made to decrease the failure rate of the systems, it got worse. Often the only choice was to abandon the product or at least to stop repairing bugs. I have been told of older software products in which the list of known, but not yet repaired, bugs, exceeded 2000 entries.

5 Reducing the costs of software aging

Inexperienced programmers can often be recognised by the elation that they show the first time that they get correct results from a program. "I'm done; it works!" is the shout of a new programmer who has just had a successful first demonstration[1]. The experienced programmer realises that this is just the beginning. They know that any serious product requires extensive testing, review and revision after the first successful run. The work that is invested by responsible, professional, organisations after the first successful run and before the first release is usually much greater than that required to get the first successful run. However, even experienced programmers focus on that first release. Our experience with software aging tells us that we should be looking far beyond the first release to the time when the product is old.

Too many papers at software engineering conferences focus on the problems of getting to the first release. Too many papers focus on the management issues, (e.g. configuration management and control). Dealing with software aging requires more than "patient management"; it requires solid engineering. It is the purpose of the remainder of this paper to consider what actions we might take to reduce the costs associated with Software Aging.

6 Preventive medicine

Since software aging is such a serious problem, the first question we must ask is what we can do to delay the decay and limit its effects.

6.1 Design for success

The first step in controlling software aging is applying the old slogan, "design for change". Since the early 70's we have known how to design software for change. The principle to be applied is known by various names, e.g. "information hiding", "abstraction", "separation of concerns", "data hiding", or most recently, "object orientation". To apply this principle one begins by trying to characterise the changes that are likely to occur over the "lifetime" of the product. Since, we cannot predict the actual changes, the predictions will be about classes of changes, e.g. revised expression representation, replacing of the terminal with a new type, changes in the user-interface, change to a new windowing system, etc. Since it is impossible to make everything equally easy to change, it is important to estimate the probabilities of each type of change. Then, one organises the software so that the items that are most likely to change are "confined" to a small amount of code, so that if those things do change, only a small amount of code would be affected. In spite of the simplicity of this principle, and in spite of its broad acceptance, I do not see much software that is well designed from this point of view. It is worthwhile to examine some of the reasons for the industry's failure to apply this principle.

- Many textbooks on software mention this technique[2], but they cover it in a superficial way. They say that one should hide, or abstract from "implementation details", but they do not discuss, or illustrate, the process of estimating the probability of change for various classes of changes. The principle is simple; applying it properly requires a lot of

[1] Students get this "rush" with the first error-free compilation.

[2] It is so well-accepted, that textbooks often fail to point out the places where the idea first appeared.

thought about the application and the environment. The textbooks do not make that clear.

- Many programmers are impatient with such considerations; they are so eager to get the first version working, or to meet some imminent deadline, that they do not take the time to design for change. Management is so concerned with the next deadline (and so eager to get to a higher position) that future maintenance costs don't have top priority.
- Designs that result from a careful application of information hiding are quite different from the "natural" designs that are the result of most programmer's intuitive work. The programmer's intuition is to think about steps in the data processing, not likely changes. Even when told to associate each module with a "secret", something that is likely to change that should be encapsulated, they use "secrets" of the form, "how to ....", and make each module perform some step in the processing, often violating the information hiding principle in the process.
- Designers tend to mimic other designs that they have seen. They don't see many good applications of information hiding. One example of information hiding design is [9]
- Programmers tend to confuse design principles with languages. For example, they believe that one cannot apply "object-oriented" ideas without an "object oriented" language. Even worse, they think that one has applied the techniques, if one has used such a language.
- Many people who are doing software development, do not have an education appropriate to the job. Topics that are "old hat" to those who attend this conference are unknown, or vague jargon, to many who are writing software. Each industry has its own software conferences and many programmers in each industry work as if their problems were unique.
- Software Engineering researchers continue to preach to converted, to write papers for each other, and to ignore what is happening where the majority software is written. They assume that "design for change" is an old problem, not one that requires further work. They are wrong!

Thus, although the principle of information hiding was first enunciated in the early 70's, (and illustrated even earlier), it is rare to find a software product that was properly designed from this point of view. The code is often clever, efficient, and correct; it performs rather amazing functions, but rarely is it designed to be easily changed. The problem is not that nobody knows how to do it, but that most programmers don't do it. I suspect that some programmers think that their program will be so good that it won't have to be changed. This is foolish. The only programs that don't get changed are those that are so bad that nobody wants to use them. Designing for change is designing for success.

6.2 Keeping records - documentation

Even when the code is designed so that changes can be carried out efficiently, the design principles and design decisions are often not recorded in a form that is useful to future maintainers. Documentation is the aspect of software engineering most neglected by both academic researchers and practitioners. It is common to hear a programmer saying that the code is it's own documentation; even highly respected language researchers take this position, arguing that if you use *their* latest language, the structure will be explicit and obvious.

When documentation is written, it is usually poorly organised, incomplete and imprecise. Often the coverage is random; a programmer or manager decides that a particular idea is clever and writes a memo about it while other topics, equally important, are ignored. In other situations, where documentation is a contractual requirement, a technical writer, who does not understand the system, is hired to write the documentation. The resulting documentation is ignored by the maintenance programmers because it is not accurate. Some projects keep two sets of books; there is the official documentation, written as required for the contract, and the real documentation, written informally when specific issues arise.

Documentation that seems clear and adequate to its authors is often about as clear as mud to the programmer who must maintain the code 6 months or 6 years later. Even when the information is present, the maintenance programmer doesn't know where to look for it. It is almost as common to find that the same topic is covered twice, but that the statements in the documentation are inconsistent with each other and the code.

Documentation is not an "attractive" research topic. Last year, I suggested to the leader of an Esprit project who was looking for a topic for a conference, that he focus on documentation. His answer was that it would not be interesting. I objected, saying that there were many interesting aspects to this topic. His response was that the problem was not that the discussion wouldn't be interesting, the topic wouldn't *sound* interesting and would not attract an audience.

For the past five or six years my own research, and that of many of my students and close colleagues, has focused on the problems of documentation. We have shown how, mathematical methods can be used to provide clear, concise, and systematic documentation of program design [3,4]. We have invented and illustrated new mathematical notation that is much more suited to use in documentation, but no less formal [5,6,7]. The reaction of academics and practitioners to this work has been insight-provoking. Both sides fail to recognise documentation as the subject of our work. Academics keep pointing out that we are neglecting "proof obligations"; industrial reviewers classify our work as "verification" which they (often correctly) consider too difficult and theoretical. Neither group can see documentation as an easier, and in some sense more important, topic, than verification. To them, documentation is that "blah blah" that you have to write. In fact, unless we can solve the documentation problem, the verification work will be a waste of time.

In talking to people developing commercial software we find that documentation is neglected because it won't speed up the next release. Again, programmers and managers are so driven by the most imminent deadline, that they cannot plan for the software's old age. If we recognise that software aging is inevitable and expensive, that the first or next release of the program is not the end of it's development, that the long-term costs are going to exceed the short term profit, we will start taking documentation more seriously.

When we start taking documentation more seriously, we will see that just as in other kinds of engineering documentation, software documentation must be based on mathematics. Each document will be a representation of one or more mathematical relations. The only practical way to record the information needed in proper documentation will be to use formally defined notation.

6.3 Second opinions - reviews

In engineering, as in medicine, the need for reviews by other professionals is never questioned. In the design of a building, a ship, or an aircraft, there is always a series of increasingly precise design documents and each is carefully reviewed by others. Although the topic of design reviews is widely discussed by software engineering lecturers, it is quite astounding too see how often commercial programs are produced without adequate review. There are many reasons for this:

- Many programmers have no professional training in software at all. Some are engineers from other fields, some are "fallen scientists" who learned programming incidentally while getting their education. Some were mathematicians, and some came from non-technical backgrounds. In many of those areas, the concept of preparing and holding a design review is nonexistent.
- Even among those that have Computer Science degrees have had an education that neglected such professional concerns as the need for design documentation and reviews. The emphasis is on the mathematics and science; professional discipline is not a topic for a "liberal" education.
- Most practitioners (and many researchers) do not know how to provide readable precise documentation of a *design*, as distinct from an implementation. No precise description, other than the detailed code, is available for review. Design reviews early in a project, when they would do the most good, are reduced to chat sessions because there are no detailed design documents to discuss.
- Much software is produced as a cottage industry, where there are no people who could serve as qualified reviewers and there is no funding to hire outside reviewers
- Software is often produced under time pressure that misleads the designers into thinking that they have no time for proper reviews
- Many programmers regard programming as an "art" and resent the idea that anyone could or should review the work that they have done. I have known programmers to quite working because they resented the fact that their work would be subject to review.

For any organisation that intends to maintain its software products over a period of years, reviews are essential and must be taken more seriously than is now usual. In particular, to ameliorate the problems of software aging, every design should be reviewed and approved by someone whose responsibilities are for the long-term future of the product. Reviews by people concerned with maintenance should be carried out when the design is first proposed and long before there is code. A discussion of how to review design documents can be found in [2].

6.4 Why software aging is inevitable

Even if we take all reasonable preventive measures, and do so religiously, aging is inevitable. Our ability to design for change depends on our ability to predict the future. We can do so only approximately

and imperfectly. Over a period of years, we will make changes that violate our original assumptions. Documentation, even if formal and precise, will never be perfect. Reviews, will bring out issues that the designers miss, but there are bound to be issues that the reviewers miss as well. Preventive measures are worthwhile but anyone who thinks that this will eliminate aging is living in a dream world.

7 Software geriatrics

Prevention is always the best medicine, but we still have to deal with old software. This section outlines several things that can be done to treat software aging that has already occurred.

7.1 Stopping the deterioration

If software has been maintained for some time without much concern for the issues raised here, a marked deterioration will be observed. The first step, should be to slow the progress of the deterioration. This is done by introducing, or recreating, structure whenever changes are made. The principles of design mentioned earlier, can be used to guide change and maintenance as well. If a design decision about the system is changed, the new data structure or algorithm can be hidden (encapsulated) in way that makes any future changes of that aspect of the system easier. Careful reviews must insure that each change is consistent with the intent of the original designers, that the original design concept is not violated by the new changes.

Stopping the deterioration is, like many other things in Software Engineering, much easier to say than to do. Many companies have allowed cancerous growth to go on unchecked in their software, for years. When times are good, growth is rapid and there is no obvious reason to be cautious. The result is that a single project may exist in many versions, each with subtly different structures and based on slightly different assumptions. When the period of rapid growth is over, every change must be made many times and the maintainers get confused by the profusion of almost alike versions. Someone has to do a serious study of all of those versions and record the differences. Then a team will have to agree on the proper structure and all versions will have to be forced into that mould. In a time when things are not going well, it is difficult to get enough staff to do the job properly.

New documents must be created and reviewed. The code must then be checked to make sure that it has been made consistent with these new documents. Such a process might take several years and during that time demands for changes and corrections will continue to come in. Nipping the growth in the bud is by far preferable. Retrenchment is always painful.

7.2 Retroactive documentation

A major step in slowing the aging of older software, and often rejuvenating it, is to upgrade the quality of the documentation. Often, documentation is neglected by the maintenance programmers because of their haste to correct problems reported by customers or to introduce features demanded by the market. When they do document their work, it is often by means of a memo that is not integrated into the previously existing documentation, but simply added to it. If the software is really valuable, the resulting unstructured documentation can, and should, be replaced by carefully structured documentation that has been reviewed to be complete and correct. Often, when such a project is suggested, programmers (who are rarely enthusiastic about any form of documentation) scoff at the suggestion as impractical. Their interests are short-term interests, and their work satisfaction comes from running programs. Nonetheless, there are situations where it is in the owner's best interest to insist that the product be documented in a form that can serve as a reference for future maintenance programmers.

A pleasant side-effect of documentation efforts is often, the improvement of the software. The formal documentation that we recommend requires a detailed and systematic examination of the code and often reveals bugs, duplicate or almost alike functions, and ways to improve performance. In a recent experiment, I asked an undergraduate student to produce documentation for a piece of software that was no longer functional. The author had left our country. Although the student was not asked to find bug, the systematic analysis necessary to create the formal documentation forced him to look at each routine carefully. He suggested some changes and the software is now functional - and well documented for future changes.

7.3 Retroactive incremental modularisation

Although all software experts now admit the importance of modularisation, and most large programs do have some units that are considered modules, a good understanding of the principles of modularisation is rarely reflected in the code. Modularisation requires more than simply identifying subroutines, or small groups of procedures and calling them modules. Each module must comprise all the programs

that "know" (are based on) a particular design decision that is likely to change. Recognising things that are likely to change requires experience, and successfully hiding or confining knowledge of a design decision to one module requires skills and understanding that are rare. Still programmers who understand information hiding and abstraction can usually find code segments that should be modules and collect them into units. A consultant, who views the software with fresh eyes, can often show how the job is done. Doing so, greatly eases the future maintenance of the code. Often of these improvements can be made at little cost as a side-effect of changes that have to be made anyway.

7.4 Amputation

Occasionally, a section of code has been modified so often, and so thoughtlessly, that it is not worth saving. Large sections can be discarded and replaced by artificial "stumps" which perform the function in some other way. Amputation is always a difficult and controversial decision. Those who have created the old code are not willing to admit that it is not worth keeping. Again, consultants are often helpful, if they can be fully informed. They don't have the emotional attachment to the code that the authors might have.

7.5 Major surgery - restructuring

When a large and important family of products gets out of control, a major effort to restructure it is appropriate. The first step must be to reduce the size of the program family. One must examine the various versions to determine why and how they differ. If one can introduce modules that hide those differences, agree on (and document) standard interfaces for those modules, and then make those changes in the various versions, one can collapse the versions into a single system that differs only in a few modules. Replacing the old versions with the restructured ones, allows future changes to the shared code to be shared by many versions. In many situations, the separate versions can be combined into one by introducing "switches" that are checked at run-time to determine which version of behaviour is wanted. This introduces a small amount of run-time inefficiency but greatly reduces the size of the maintenance staff. I have seen a few organisations that were able to offer what appeared to be a broad family of products by distributing a single piece of code and setting hidden switches to create systems that appear to be quite different. The maintenance costs of these organisations are much lower than they would be if they had separate versions. Unfortunately, some of their customers found the switches and were able to enjoy the benefits of features that they had not purchased. In spite of this, I suspect that the software manufacturer was ahead because of reduced maintenance costs.

8 Planning ahead

If we want to prevent, or at least slow down, software aging, we have to recognise it as a problem and plan for it. The earlier we plan for old age, the more we can do.

8.1 A new "Life Style"

It's time to stop acting as if, "getting it to run" was the only thing that matters. It is obviously important to get a product to the customer quickly, but we cannot continue to act as if there were no tomorrow. We must not let today's pressures result in a crippled product (and company) next year. We cannot do good work under stress, especially the constant stress of a 25 year crisis. The industry itself must take steps to slow down the rapid pace of development. This can be done by imposing standards on structure and documentation, making sure that products that are produced using "short cuts" do not carry the industry "seal of quality".

8.2 Planning for change

Designs have to be documented, and carefully reviewed, before coding begins. The programs have to be documented and reviewed. Changes have to be documented and reviewed. A thorough analysis of future changes must be a part of every product design and maintenance action. Organisations that are bigger than a few people should have a professional, or a department, devoted to reviewing designs for changeability. They should have the power to veto changes that will get things done quicker now but at a great cost later.

8.3 If it's not documented, it's not done

If a product is not documented as it is designed, using documentation as a design medium [1], we will save a little today, but pay far more in the future. It is far harder to re-create the design documentation than to create it as we go along. Documentation that has been created after the design is done, and the product is shipped, is usually not very accurate. Further, such documentation was not available when (and if) the design was reviewed before coding. As a result, even if the documentation is as good as it would have been, it has cost more and been worth less.

8.4 Retirement savings plans

In other areas of engineering, product obsolescence is recognised and included in design and marketing plans. The new car you buy today, is "old hat" to the engineers who are already working on future models. The car is guaranteed only for a (very) limited time and spare parts are also required to be available only for prescribed periods. When we buy a car we know that it will age and will eventually have to be replaced. If we are wise, we begin to plan for that replacement both financially and by reading about new developments. The manufacturers show similar foresight. It is only in the software industry where people work as if their product will "live" forever. Every designer and purchaser of software should be planning for the day when the product must be replaced. A part of this planning is financial planning, making sure that when the time comes to install or develop a new product, the funds and the people are there.

9 Barriers to progress

If we are going to ameliorate the problem of aging software, we are going to have to make some deep changes in our profession. There are four basic barriers to progress in Software Engineering. These are attitudes and assumptions that make it impossible for research to make a difference.

9.1 A 25 year crisis?

I first heard the term "software crisis" 25 years ago and have heard it used to describe a current problem every year since then. This is clearly nonsense. A crisis is a sudden, short-term serious emergency. The so-called "software crisis" is certainly serious, but it is neither sudden nor short-term. It cannot be treated as if it were a sudden emergency. It needs careful long-term therapy. "Quick and easy" solutions have never worked and will not work in the future. The phrase "software crisis" helps in dealing with certain funding agencies, but it prevents the deep analysis needed to cure a chronic illness. It leads to short-term thinking and software that ages quickly.

9.2 "Our industry is different."

Software is used in almost every industry, e.g. aircraft, military, automotive, nuclear power, and telecommunications Each of these industries developed as an intellectual community before they became dependent upon software. Each has its own professional organisations, trade organisations, technical societies and technical journals. As a result, we find that many of these industries are attacking their software problems without being aware of the efforts in other industries. Each industry has developed its own vocabulary and documents describing the way that software should be built. Some have developed their own specification notations and diagramming conventions. There is very little cross-communication. Nuclear Industry engineers discuss their software problems at nuclear industry meetings, while telecommunications engineers discuss very similar problems at entirely different meetings. To reach its intended audience, a paper on software engineering will have to be published in many different places. Nobody wants to do that (but promotion committees reward it).

This intellectual isolation is inappropriate and costly. It is inappropriate because the problems are very similar. Sometimes the cost structures that affect solutions are different, but the technical issues are very much the same. It is costly because the isolation often results in people re-inventing wheels, and even more often in their re-inventing very bumpy and structurally weak wheels. For example, the telecommunications industry and those interested in manufacturing systems, rarely communicate but their communication protocol problems have many similarities. One observes that the people working in the two industries often do not realise that they have the same problems and repeat each other's mistakes. Even the separation between safety-critical and non safety-critical software (which might seem to make sense) is unfortunate because ideas that work well in one situation are often applicable in the others.

We need to build a professional identity that extends to people in all industries. At the moment we reach some people in all industries but we don't seem to be reaching the typical person in those industries.

9.3 Where are the professionals?

The partitioning of people and industries with software problems is a symptom of a different problem. Although we have lots of people who are paid to write software, we don't have software engineers in the sense that we have aeronautical, electrical, or civil engineers. The latter groups are primarily people who have received a professional education in the field in which they work, belong to professional societies in that field, and are expected to keep up with that field. In contrast, we find that software in the nuclear field is written by nuclear engineers who have learned a programming language, software in the telecommunications field is written by communications

engineers and electrical engineers, software in the automated manufacturing field is written by mechanical engineers, etc. Programming engineers in those industries do not think of themselves as a profession in the sense that aeronautical or nuclear engineers do. Moreover, they have not received a formal education in the field in which they are now working. We find that engineers who write programs know far to little about computing science, but computer science graduates know far too little about engineering procedures and disciplines. I often hear, "anybody can write a program" and it's true, but programs written in an unprofessional way will age much more rapidly than programs written by engineers who have received an education in the mathematics and techniques that are important to program design.[8]

9.4 Talking to ourselves

Researchers have to start rethinking their audience. All too often, we are writing papers to impress our colleagues, other researchers. Even worse, if we try to write a paper for the practitioner, the referees complain if we include any basic definitions or problems. We end up writing papers that are read by our fellow researchers but not many others. We also spend too little time finding out what the practitioners know, think, and need. In Faculties of Engineering, professional practice is recognised as essential to good teaching and research. In many Science faculties, it is viewed simply as a way to make some extra money. This is one of many reason why I believe that Computer Science Departments would function better if they were always part of an Engineering Faculty.

10 Conclusions for our profession

(1) We cannot assume that the old stuff is known and didn't work. If it didn't work, we have to find out why. Often it is because it wasn't tried.

(2) We cannot assume that the old stuff will work. Sometimes widely held beliefs are wrong.

(3) We cannot ignore the splinter software engineering groups. Together they outnumber the people who will read our papers or come to our conferences.

(4) Model products are a must. If we cannot illustrate a principle with a real product, there may well be something wrong with the principle, Even if the principle is right, without real models, the technology won't transfer. Practitioners imitate what they see in other products. If we want our ideas to catch on, we have to put them into products. There is a legitimate, honourable and important place for researchers who don't invent new ideas but, instead, apply, demonstrate, and evaluate old ones.

(5) We need to make the phrase "software engineer" mean something. Until we have professional standards, reasonably standardised educational requirements, and a professional identity, we have no right to use the phrase, "Software Engineering".

11 References

[1] HESTER, S.D., PARNAS, D.L., UTTER, D.F., "Using Documentation as a Software Design Medium", *Bell System Technical Journal*, 60, 8, October 1981, pp. 1941-1977.

[2] PARNAS, D.L., WEISS, D.M., "Active Design Reviews: Principles and Practices", *Proceedings of the 8th International Conference on Software Engineering*, London, August 1985. Also published in *Journal of Systems and Software*, December 1987.

[3] VAN SCHOUWEN, A.J., PARNAS, D.L., MADEY, J., "Documentation of Requirements for Computer Systems", presented at *RE '93 IEEE International Symposium on Requirements Engineering*, San Diego, CA, 4 - 6 January, 1993.

[4] PARNAS, D.L., MADEY, J., "Functional Documentation for Computer Systems Engineering (Version 2)", CRL Report 237, CRL-TRIO McMaster University, September 1991, 14 pgs. (to be published in *Science of Computer Programming)*

[5] PARNAS, D.L., "Tabular Representation of Relations", CRL Report 260, CRL. McMaster University, October 1992, 17 pgs.

[6] JANICKI, R., "Towards a Formal Semantics of Tables", CRL Report 264. CRL McMaster University, September 1993, 18 pgs.

[7] ZUCKER, J.I., "Normal and Inverted Function Tables", CRL Report 265,CRL, McMaster University, December 1993, 16 pgs.

[8] PARNAS, D.L., "Education for Computing Professionals", *IEEE Computer*, vol. 23, no. 1, January 1990, pp. 17-22.

[9] PARNAS, D.L., CLEMENTS, P.C., WEISS, D.M., "The Modular Structure of Complex Systems", *IEEE Transactions on Software Engineering*, March 1985, Vol. SE-11 No. 3, pp. 259-266. Also published in *Proceedings of 7th International Conference on Software Engineering*, March 1984, pp. 408-417.