Successful Software Engineering Research

David Lorge Parnas, P.Eng.
Software Engineering Research Group
DEPARTMENT OF COMPUTING AND SOFTWARE
Faculty of Engineering
McMaster University, Hamilton, Ontario; Canada L8S 4K1

1 Introduction

Rumination about what makes research successful is a strong indication that a researcher will not continue to do successful research. Nonetheless, the invitation to publish a short article in SEN on the occasion of being honoured by receiving SIGSOFT's "Outstanding Research Award" has led me to reflect on what I have done. I have been active in research on software design for more than 35 years; perhaps this is the time to pause and look back. I also want to look forward; I have some concerns about the direction being taken by many researchers in the software community and would like to offer them my (possibly unwelcome) advice.

2 Research in other areas of engineering

I have the pleasure of working in a Faculty of Engineering and often walk past bulletin boards where the work of Mechanical, Chemical, and Civil Engineering colleagues is posted. As I look at these papers, I notice a pattern. The majority of those papers begin by describing a problem that is frequently encountered in connection with product design or production. They proceed to develop a model of the essential or fundamental parts of the problem, abstracting from facts that they consider irrelevant, and then proceed to analyse that model. Finally they show how the results of their analysis can be applied to solve, or improve the solution of, the original problem. Somewhere in the paper, there is a survey of alternative approaches, including those in the literature and those in use in other industrial environments.

Because of my deep interest in computer science and mathematics, I also get announcements and papers from those fields. The tradition of scholarship in these fields, (and others such as philosophy, history, literature), is quite different from that in engineering. Outside of engineering, scholars often begin with an analysis of the literature in their field, highlighting work that is related to their own. Often, they identify something that is either missing or, in their opinion, wrong in the earlier work. This allows them to explain their own approach to the subject and to then present their new results. Sometimes, but not always, there is a section that discusses the practical implications or applications of their work.

I can reformulate this observation to make my point more explicit. In engineering research, problems are found in current practice, abstraction is used to identify the fundamental issues, and analysis provides insight on those issues. After the abstract models are investigated, the engineering researcher provides some advice on how to solve the original design or production problems. Advice to developers is the goal of the research. In the other fields, many problems are found in the research literature and the goal seems to be to add to that literature. I know many exceptions to these observations, but they are "the exceptions that prove the rule"; they stand out because they are exceptions.

3 Is SIGSOFT consistent?

I am the second winner of the SIGSOFT award. An obvious approach is to ask what my work has in common with that of the 1997 winner of the prize, Dr. Barry Boehm. At first glance, the answer is "nothing". Barry Boehm addresses questions that strike me as too hard to answer and I don't see him attacking the issues that I personally view as most pressing. Is the fact that we have both won this prize merely a coincidence? If you look more deeply however, our work is similar and that the common properties are the secret of success.

Dr. Boehm and I have both followed the engineering pattern. Both of us had extensive contact with industrial practice and both set out to solve what we perceived to be the most serious problems encountered by practitioners. The fact that we picked different problems is not important for this discussion. It may be the result of the kind of industry or simply where we were within the company. The essential point is that our research was stimulated by what we saw in industrial practice.

4 Which paradigm is followed in Software Engineering research?

The reason that I have chosen to write on this difficult topic is that when I examine the literature in "Software Engineering", I see papers that follow the paradigm of fields other than engineering. Further, I observe referees judging papers by the standards of non-engineering fields. Finally, I observe that most software developers, ignore the bulk of our research. Whereas practising engineers find things of value in research publications, most software developers do not.

I have angered colleagues with this observation before, but it seems to me that much of the research published in our conferences and journals is ignored by software developers because it does not address issues that concern developers or offer solutions that they can use. When most software developers read, they don't look at research literature, but at "slick" magazines offering superficial descriptions of easy answers. Engineering in general, and software engineering in particular, is always difficult. Market pressures force us to try to do better than those who worked before us. Easy answers are usually not answers at all; easy answers are diversions. I do not see solid useful advice in the most popular software magazines. However, I cannot advise the readers of those magazines to turn to the research journals. The authors who write in those journals have some other audience in mind.

5 An ancient "case study"

My first piece of successful research was the result of an invitation to leave academia for a while and work with software developers at Philips Computer Industry in Apeldoorn, Netherlands. I went there thinking that I had the answer to

software development problems. Fortunately for me, the people who shared my vision, and had invited me to Apeldoorn, had been reassigned to another location. I found myself in a room full of people who were neither missionaries nor researchers; they were developing software. As I watched their work, and tried to understand their discussions, I realised that my vision (which I later discovered was shared by several other researchers) was completely wrong. I was assuming things to be easy that were in fact impossible. The developers' major problems were problems that I had never considered, problems that none of my professors or colleagues thought worthy of discussion.

It was sitting at a lunch table, listening to a frustrated discussion about changing interfaces, that led me to start questioning the way that people throughout the industry were dividing software into work assignments, which they called modules. In the middle of a sandwich, I predicted that there would be severe problems because one of my colleagues was describing an interface by drawing a picture of a control block on a paper napkin; I told them not to discuss the data structure because it was very likely to change. They told management that I was trying to obstruct progress. Two holes were punched in the napkin and it was inserted in a binder with other design documentation.

When it was time to integrate the software components that were discussed during that lunch, it turned out that (1) they were incompatible, and (2) they were very hard to change. Several important deadlines were missed because the picture on the napkin was no longer valid, but many people had based their program design on that interface description.

I had been among these intelligent, hard-working people long enough to understand their need for interface descriptions that were complete and precise. However, having the perspective that sometimes comes from being only a visitor, I saw something more. I realised that they should be using interfaces that were simpler, and less likely to change, than the data structures that were used to pass data from one program to another. I began to realise that only a different decomposition would allow stable interfaces. It was from this experience that the principle now known as "information hiding" evolved.

When I returned to my university position, I began informal (not controlled) experimentation to see if my idea could work in practice. Four widely cited papers [5,6,7,8] were the immediate result; the term "information hiding" which I coined to try to explain how my structure was derived, has since appeared in many software engineering textbooks.

6 Lessons learned

I see two important points in this anecdote. First, I would never have realised the nature of the problem, unless I had been working on that project, reviewing development documents, and sitting at that lunch table. Second, I chose <u>not</u> to respond to the immediate needs of the developers. They thought that their problem was simply that they did not know how to document the interfaces; their pictures described the format, but not the meaning of the data. Moreover, drawing those pictures took lots of time. They never questioned the need to draw and distribute the pictures; they never questioned organising the software so that the pictures had to be used to communicate between programmers. Everyone did it that way!

I have known researchers who, in similar situations, wrote programs that would draw pretty pictures of control blocks. Such a tool would have provided symptomatic relief, would have been welcomed by the developers; it would have been publishable research. However, the pictures did not describe the semantics of the control blocks, and they did not reduce either the impact of interface changes or make interface changes less likely. Further, if they wanted such a tool, the developers could have built it; there was nothing that required research training or the time to think enjoyed by most researchers.

I believe that the role of the successful engineering researcher is to understand developers' problems, but to use the luxury of not having to meet short-term deadlines, to look for the underlying causes and fundamental cures rather than immediate, symptomatic, relief. Developers, who must meet pressing market driven deadlines, do not have the time to look for long-term solutions. That is the researcher's job.

7 Some historical perspective

Younger researchers, who have heard about information hiding or abstraction since they were first introduced to programming, may not appreciate how novel the ideas were at the time. When I discussed the problem of software decomposition with my academic colleagues, they were not at all interested. One, whose work has had incredible impact over the years, told my department head that there was no substance in the problem of modularisation and suggested that I be fired. Another, a very senior person in Artificial Intelligence, claimed that the problem of software development was easy, would soon be solved, and was not worthy of academic research.

In his classic, still popular, and still important, book, "The Mythical Man Month"[1], Fred Brooks referred to my proposal as "a recipe for disaster".

When I first submitted [7] (with the title, "A New Criteria For Dividing Systems Into Modules"), it was rejected with a one line review that said, "Obviously Parnas doesn't know what he's talking about because nobody does it that way." I got the paper accepted without substantive change by pointing out that since my paper claimed that the method was new, it should not be rejected because nobody did it. Approximately ten years later, a textbook mentioned the same paper and said, "but Parnas only wrote down what all good programmers were doing anyway". If I believe both of these observers, I can conclude that the set of good programmers was empty.

In fact, neither observation was true. I found my idea by com-

[1] The 20th anniversary edition of the book, now recognises this remark is incorrect.

paring the few systems that did not have interface problems, with many that did. I rejected empty phrases, like "beautiful", "clean", and "elegant", which had been used to describe the better systems, and looked for a criteria that explained to engineers (who are widely believed to have no appreciation of beauty) what had to be done. In those days, the idea of hiding information was considered subversive. The company that I worked with thought that the solution to software problems lay in standardising documentation in order to make all design information accessible to everyone. My insight was obtained by a very unacademic type of research - reading a lot of code and reflecting on what was happening. Today, the idea seems so obvious that I am uncomfortable talking about it, but if I read more code, I see that it is still worth teaching. Even in programs that use the latest languages and are described as "object oriented", I find a failure to use abstractions or hide information.

8 Open problems for researchers

My 1972 papers left lots of questions open for other researchers, but very few people followed them up. Simple minded approaches to information hiding, and simple-minded implementations of information-hiding modules, lead to very inefficient programs and made the ideas, though logical, seem impractical. Research was needed on how to design interfaces, how to implement intermodule communication, etc. I had raised the problem of decomposition into modules, but there was, and is, a need for work on composing systems from separately developed "information hiding" modules. Implementation methods that work well for modules that do not abstract from information, do not usually work well with information hiding modules.

Research was also needed on how to apply the idea to systems with many independently changeable design decisions. My case study had only 5. Many methods work well when there are 5 components but not when there are 50 or 500. Although the idea of information hiding was quickly accepted by researchers, it was not being applied by the majority of software developers. If it had been widely applied, we would not have the "year 2000" problem today. In other words, all that researchers had to do was study why information hiding was not being used and they would have found lots of interesting and challenging problems worth investigating. In fact, most academic and industry researchers simply assumed that the issue was solved and returned to other issues (e.g. developing more new languages).

9 Who is studying inspection methods?

Anyone who takes a close look would realise that software inspection is a major problem in many development environments. The industry badly needs methods that will help inspectors to proceed systematically, carefully considering all cases in a way that provides confidence that nothing has been overlooked. There have been influential publications on inspection beginning with [2], followed up by [3] and, more recently, a book [4]. However, note that this work does not come from academic researchers but from practitioner/consultants. More important, these pragmatic publications focus on the management/organisational aspects of inspections, and take no advantage at all of the vast body of research literature on mathematical methods of verification. I first became aware of the difficulty in inspecting documents and code when attempting to apply design methods to an avionics system and we offered some useful advice in [10]. However, I became more aware of the importance of this problem, and the continuing dearth of research literature about it, when asked to work on inspecting a safety-critical system [13]. Driven by an immediate need, we developed an improved method (described in [12]), but there is still a tremendous need for improved methods and for tools and I see very little academic interest in this problem. I have seen a few other papers on the topic but noted little substance beyond that in those cited. Mathematically supported inspection should be far easier than automated verification, and of immediate value, but it has not attracted the attention of mathematically oriented researchers.

10 Who is taking a serious look at documentation?

My current area of study is another example of a sadly neglected topic. About a decade ago, a series of informal conversations with software developers led me to realise just how much time and money is lost because of the poor quality of software maintenance documentation. Ask your favourite software developer why a mistake was made and you are very likely to be told that the documentation was unclear, incomplete, inconsistent, or inaccurate. Programs are very precise and sensitive to minor changes. Complete documents must include a lot of detail and cover many different cases. Even program descriptions that describe what programs do, not how they do it, will be bulky and must be organised in such a way that (a) the information that you need is easy to find and (b) gaps and inaccuracies can be detected. Finding ways to write precise program documentation that is organised for information retrieval is a tremendously fertile field with many concrete problems for software engineering researchers who want their work to have impact on software developers. That field is not being ploughed by very many. In fact, the problem is not even accepted as "real research" by the people who are best qualified to solve it.

My engineering education has shown me how mathematics plays an essential role in the documentation of engineering products. My associates and I have carefully studied much of the "formal methods" literature and concluded that what those papers offer is not a solution to the software documentation problem. The examples that I have studied obviously represent a great deal of careful, often creative, thought. The specifications were certainly difficult to write, but they will be even harder to read. In most approaches, the reader is expected to derive the behaviour from a subtle set of axioms that may interact in surprising ways. In engineering mathematics, the documentation of behaviour is described by formulae, which the reader can evaluate simply by plugging in the values for the case that interests them. The "engineering approach" and the "formal methods" approach are equally mathematical, but they require a very different kind of rea-

soning when you use the mathematics to find out what a program will do.

Because we have used mathematical notation, and many researchers appear not to have thought about the practical aspects of program documentation, I have found reviewers judging our work as if it were work on denotational semantics. They are always disappointed because they find no new mathematics. In fact, we are pleased to have been able to apply the simplest of mathematical models. The fact that we have new things to say on how to represent and organise mathematical information is not important to many researchers because they have never watched a software developer try to answer specific questions about the code. More important, the broad field of software documentation includes many small and solvable research problems that are not being pursued by those who have the necessary mathematical background. In the meantime, the popular literature continues to suggest that software developers can be "engineers" without knowing or using mathematics. My colleagues and I have published on this topic [11, 14] but those papers represent early work and there are many research problems that must be solved before the methods become suitable for everyday use.

11 Conclusions

I am repeatedly amazed at how unaware many software engineering researchers seem to be of the differences between what is recorded in research literature and textbooks and what is actually happening. I am also amazed at how frequently people respond to academic papers without trying to understand the "real" problems or asking how current systems solve those problems.

I conclude with advice to Software Engineering researchers:

- Keep aware of what is actually happening by reading industrial programs.
- Try to apply your ideas to programs that were written for some other purpose, not to programs that you made up to illustrate your ideas.
- Don't attack the symptoms, but keep looking for the causes. The developers can, and will, attack the symptoms at least as well as we can.
- Keep asking why people aren't using our ideas and don't take "stupidity" or "ignorance" as an answer. You cannot eliminate stupidity and you can do little to correct ignorance, but if there is a weakness in existing research results, you have found a solid research problem.
- Be wary of fads. During my career I have seen many topics become very popular and then disappear. Who today is seriously interested in Algol-68, PL/I or Ada? However it hasn't been long since the research literature was filled with papers on those topics. Research topics are particularly likely to be fads in a field where each new paper is a response to a previous paper rather than to a fundamental problem. Always look for the fundamental problem and don't jump on bandwagons. Papers about yesterday's fads are forgotten.
- Be wary of vaguely defined buzzwords. A "buzzword" is a word that everyone knows but few people can define. "Buzzword" is a buzzword. Much of today's literature is a debate about the meaning of words that is a disguised as a debate about how to design software. For example, most of the debates that I see about the strengths and weakness of various Object Oriented (O-O) approaches boil down to differences of opinion about what O-O means. Pointing out buzzword problems is another service that researchers can provide [9].

The secret to successful research is picking the right problem. I have known many people who were better at solving problems than I am, but, I have been honoured by SIGSOFT's award because I found my research problems by working with developers.

References

[1] Brooks Frederick P. Jr., "*The Mythical Man-Month: Essays on Software Engineering*", Addison-Wesley Publishing Co. 1975, Frederick P. Brooks Jr., ISBN 0-201-00650-2.

[2] Fagin M.E., "Design and Code Inspections to Reduce Errors in Program Development", *IBM Systems Journal*, No. 3. 1976 pp. 184-211

[3] Fagin M.E., "Advances in Software Inspections", *IEEE Trans. Software Engineering*, July 1986 pp. 744-751

[4] Gilb, Tom "*Software Inspection*", Addison Wesley, 1993

[5] Parnas, D.L. "Information Distributions Aspects of Design Methodology", *Proceedings of IFIP Congress 1971*, pp. 26-30, 1972

[6] Parnas, D.L. "A Technique for Software Module Specification With Examples", *Communications of the ACM*, Vol. 15, No. 5, pp. 330-336, May 1972

[7] Parnas, D.L. "On the Criteria To Be Used in Decomposing Systems Into Modules" *Communications of the ACM*, Vol. 15, No. 12, pp. 1053-1058, December, 1972.

[8] Parnas, D.L., "Some Conclusions from an Experiment in Software Engineering Techniques", *Proceedings of the 1972 FJCC*, 41, Part I, pp. 325-330.

[9] Parnas, D.L., "On a 'Buzzword': Hierarchical Structure", *IFIP Congress* '74, North Holland Publishing Company, 1974, pp. 336-339.

[10] 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 in Journal of Systems and Software, December 1987.

[11] Parnas, D. L., Madey, J., Iglewski, M., "Precise Documentation of Well-Structured Programs", *IEEE Transactions on Software Engineering*, Vol. 20, No. 12, December 1994, pp. 948 - 976.

[12] Parnas, D. L. "Inspection of Safety Critical Software using Function Tables", *Proceedings of IFIP World Congress 1994*, Volume III, August 1994, pp. 270 - 277.

[13] Parnas, D. L., Asmis, G.J.K., Madey, J., "Assessment of Safety-Critical Software in Nuclear Power Plants", *Nuclear Safety*, vol. 32, no. 2, April-June 1991, pp. 189-198.

[14] Parnas, D.L., Madey, J., "Functional Documentation for Computer Systems Engineering" *Science of Computer Programming* (Elsevier) vol. 25, number 1, October 1995, pp. 41-61, a

Editor's Filler

Hey! Wasn't that good?

Didn't you catch yourself say "Yes! I knew that!"

I am glad David took the time to speak his mind, now if anyone has anything to add, send me a note and I will put it in a subsequent issue.

And now, another invited paper.

Read on and learn.

A History of Software Engineering at the National Science Foundation (A Personal View)

Bruce H. Barnes
Deputy Division Director (Retired)
Computer and Computation Research
National Science Foundation
<bhbarnes@erols.com>

The academic software engineering community can be justly proud of its accomplishment in the last twenty-five years. When I joint the National Science Foundation in 1974, Software Engineering as an academic discipline hardly existed. Structured programming and top-down design had permeated the curriculum, but that was the extent of software engineering in the curriculum. "Curriculum 1978" did not use the term Software Engineering, but emphasized good software development practices. In the objectives for the curriculum it stated that "Computer science majors should be able to write programs in a reasonable amount of time that work correctly, are well documented and arereadable." The report also notes that "The topics [in structured design] are of such importance that they should be considered a common thread throughout the entire curriculum." By 1986 most Computer Science programs had introduced senior project courses involving a significant portion of quality software engineering practices. The 1989 ACM report on "Computing as a Discipline" included Software Methodology and Engineering as one of its elements. The "Curriculum 1991" report of the ACM and the IEEE-Computer Society recommends a significant amount of software engineering for every computer science graduate and includes a sample curriculum with a Software Engineering emphasis. Currently there are Software Engineering programs, especially in Europe. There are even discussions concerning the accreditation of undergraduate programs in Software Engineering. Over the last 25 years the Computer Science curriculum has evolved from one based on the paradigms and philosophy of Mathematics and the Sciences, to one with more of an engineering emphasis. I believe that NSF's early recognition of the role of Software Engineering in the academic environment contributed to the evolution.

I was on leave from The Pennsylvania State University when I stared with the Nation Science Foundation as Program Director for Theoretical Computer Science. This program was part of the Computer Science and Engineering Section of the Division of Computer Research. The Division also had a section on Computer Applications in Research. The goal of this section was to develop and promote computational techniques for employing computers in scientific and engineering research. Software Quality Research was one of the programs in that section. It mainly supported the development of very high quality mathematical software. LINPACK was one of its major successes. It also supported some research into techniques for producing high quality software, i.e. software engineering. In 1976 we decided that computer applications were