A Future for Software Engineering?

Leon J. Osterweil
Laboratory for Advanced Software Engineering Research
University of Massachusetts
Amherst, MA 01003 USA
ljo@cs.umass.edu

Abstract

This paper suggests the need for a software engineering research community conversation about the future that the community would like to have. The paper observes that the research directions the community has taken in the past, dating at least back to the formative NATO Conferences in the late 1960's, have been driven largely by desire to meet the needs of practice. The paper suggests that the community should discuss whether it is now appropriate to balance this problem-solving-oriented research with a stronger complement of curiosity-driven research. This paper does not advocate what that balance should be. Neither does it advocate what curiosity driven research topics should be pursued (although illustrative examples are offered). It does does advocate the need for a community conversation about these questions.

0. Preamble

"…we recognize that a practical problem of considerable difficulty and importance has arisen: The successful design, production and maintenance of useful software systems. The importance is obvious and the more so since we see only greater growth in demands and requirements in the future. The consequences of poor performance, poor design, instability and mismatching of promise and performance are not going to be limited to the computing fraternity, or even their nearest neighbors, but will affect considerable sections of our society whose ability to forgive is inversely proportional to their ignorance of the difficulties we face. The source of difficulty is distributed through the whole problem, easy to identify, and yet its cure is hard to pinpoint so that systematic improvement can be gotten.

Our problem has arisen from a change of scale which we do not yet know how to reduce to alphabetic proportions. Furthermore we must assume that additional magnification of goal will take place without necessarily being preceded by the emergence of a satisfactory theory or an organized production of tools that will permit work and costs to fall on growth curves which lie significantly below those which now exist. For example, we can see coming the need for systems which permit cooperation, e.g., between engineering and management information. Not only must we know how to build special purpose systems, but how to combine them into larger ones." A.J. Perlis, 1968 [17].

1. Introduction

What is the future of Software Engineering? To project the future, it seems necessary to understand the past—and the present as well. Knowing the past and the present will indicate just where we are and will also suggest the momentum that will be pushing us forward. But our community and our discipline will face additional forces that will try to move us and shape our future as well. Many of these forces are external, and will have a crucial effect on where we go and what we do. This paper argues, however, that we would be well advised not to allow our future directions to be determined solely by external forces. Decisions that we make for ourselves and about ourselves should also be prime forces that guide us forward.

Thus, the structure of this paper is as follows: we begin with a brief summary of the history of Software Engineering, leading up to an attempt to characterize our current position and situation. The paper then proceeds with a discussion about a variety of current trends, and their potential impacts upon the discipline of Software Engineering. The paper concludes with a plea for community introspection in the form of *curiosity-driven research*, suggesting the need for forums for setting goals and directions for the future, especially as illuminated by consideration of the past and present.

The future of Software Engineering should be in our own hands. But it may require some proaction and determination to seize it.

2. The past

Most observers would agree that Software Engineering began as a discipline in the late 1960's with the famous "NATO Conferences" [3, 12]. But, in fact, software was being developed and engineered for at least 15 years prior to the first of these conferences. The conferences, nevertheless, gave this activity a name, "Software Engineering", and an overriding goal, namely dealing with the "Software Crisis". The attendees at these meetings were senior figures in computing. In that they discovered a remarkable set of similarities in the problems that they were having trouble dealing with, they thus legitimatized Software Engineering as the study of the broad range of problems encountered in developing software.

The presentations and discussions at the 1968 NATO Conference ranged over a set of topics, some of which no longer seem relevant. For example, the debate about whether or not software should be priced separately from hardware [12, pp. 129-135] has long since been resolved. Likewise, there is now little discussion of whether or not system software can reasonably be coded in a higher level language. Both were topics of considerable debate in 1968.

But the 1968 NATO Conference also devoted considerable attention to many issues that are quite familiar to us today. Thus the issue of how to create processes that could be expected to be effective in producing high quality software on schedule and within budget [2, 6, 21] was highlighted. Dijkstra [4] advocated the use of hierarchy that anticipated approaches that gained popularity in subsequent decades. McIlroy [11], moreover, was already advocating the use of components as keys to effective software development.

Problems of scale were also introduced and addressed by speakers such as Perlis [17], and others [12, pp. 65-70]. Another major topic of conversation in 1968 was software quality. Llewelyn and Wickens [9] focused on the issue of testing in general. Pinkerton [18] addressed approaches to the problem of performance testing. This was complemented by considerable discussion throughout the meeting of the role of more mathematical approaches to reasoning about software in order to assure correctness and other qualities. Yet another topic of considerable discussion was the nature of design, its role in the overall software development process, and its relation to coding [12, pp. 35-65]. All of these issues remain before our community today, and continue to be the subjects of ongoing research and discussion.

The response of the early pioneers in Software Engineering was to grasp these problems eagerly and begin to seek ways to address them. Better programming languages were proposed and evaluated. Tools for supporting testing were implemented, and were complemented by more formal methods of analysis, all aimed at measuring and improving quality. There was recognition of the need for considerable effort prior to coding. The importance of requirements was emphasized, and the position of this activity in the larger software development lifecycle was established. The importance of effective management of development led to the establishment of software metrics, and attempts to use them to guide the development process.

It is important to note that many of these research efforts led to important improvements in the way in which software engineering has been practiced. In many cases, the impact of the research has been indirect, and in nearly all cases, the impact has taken a frustratingly long time to be seen and felt. But impact has been felt, and can be demonstrated, as well will elaborate upon shortly.

On the other hand, despire vigorous efforts over the past 40 years, many of the problems of 1968 remain essentially unsolved. Surely there has been gratifying progress in such areas as programming language improvement, software testing approaches, support for precoding phases of development, and software metrics. But in each area the solutions currently at hand are far from adequate for solving the problems that software developers face. The comments that Perlis made in 1968, quoted in the preamble to this paper, are, even today, a reasonable characterization of the situation that we face as software engineers. Thus, the stubborn persistence of problems first identified decades ago has certainly led to frustration. But it has increasingly also led to a growing suspicion that the immediate problems might well be manifestations of issues that are far deeper and more profound.

It seems particularly important to observe that frustrations arising from continued grappling with stubborn problems has often led to understandings of deeper issues that turn out to be needed in order to fashion successful approaches to the problems. This has certainly been the experience in other disciplines. Thus, for example, calendar-making, and predicting the positions of planets in the sky, became increasingly difficult and frustrating over the centuries, until the Copernican revolution identified the need to replace the Earth with Sun as the center of the solar system. This conceptual shift enabled the establishment of rigorous laws of planetary motion that, in turn, led to substantial simplifications of problems such as predicting planetary positions and establishing reliable calendars.

Similarly, electricity was a source of amusement and amazement through the early years of the 19th century. But it could not be used as the source of reliable energy for lighting and for the operation of machinery until its fundamental nature was understood, and its behavior characterized by scientists such as

Maxwell. In medicine too, progress in the treatment of diseases was ad hoc and unsatisfactory until the nature of microorganisms was understood. These understandings paved the way for the effective application of antibiotics in treating broad classes of infections. More recent understandings of the nature of viruses is starting to lead to antiviral drugs and therapies increasingly available today.

In a similar way, we note that software engineers over the past few decades have also increasingly viewed the persistence of stubborn problems as indicators of the need for fundamental research that probes the nature of deeper problems. Thus, for example, software evolution was recognized as a central problem at least as long ago as during the 1968 NATO conference, where the need to migrate software in response to changes in need was identified as a major software engineering problem. A variety of tools and systems for addressing this problem were built and evaluated. But the stubborn resistance of this problem to a variety of solution attempts eventually led early researchers such as Parnas [16] to look for deeper issues and ultimately suggest approaches such as implementation-hiding, and the enunciation of the deep concept of modularity. Exploration of the nature of modules, and their use in improving the effectiveness of software development, was also addressed by others (eg. [7. 8]), and has led over the years to a spectrum of improvements to practice.

In t1his example, we see a nice illustration of the duality, and mutual reinforcement, of problem-solving and theoretical conceptualization. A problem in software evolution was eventually seen as a symptom of the need for a more disciplined approach to the development of software, centered around a recognition of the need for implementation-hiding modules. A period of conceptual research followed, during which there was considerable attention paid to understanding the nature of a module, to defining modules rigorously, and to supporting the implementation of modules and their use in evolution. The basic concept, and the basis it provided for the development of more effective tools, led to more effective approaches to the evolution of software, and to other improvements as well.

It is important to note that in this example, as in a number of other examples that could be adduced, the problem originated in the domain of practice. The research community understood the importance of the problem, found it to be interesting, and made it a part of the community research agenda. Understanding of the underlying problem was reached as a consequence of fundamental conceptual research, which in turn led to improvements in practice. The history of Software Engineering contains a number of other such illustrations of this close duality. Indeed, it seems particularly interesting to note that the editors of the 1969 NATO conference had already noted that the tone of that second conference emphasized the growing problem of a "software gap" between theory and practice. The proceedings feature a long comment from Strachey [23] that focuses on the existence of strong divisions between those who actually build large software systems, and those who would try to help with research ideas and approaches. Strachey and others at the conference emphasized the need for greater mutual understanding between the communities of software development practice and software research (theory). Certainly this theme has resonated through the decades that have ensued, and to our present situation today.

It seems clear that the agenda for basic software engineering research has in the past been driven strongly by study of the problems arising in practice, and progress in practice often benefits greatly from the results of research. This continues to be the case today. None of this should be at all surprising, as it is very much analogous to experience in such other areas as medicine, physics, and chemistry. In some important senses, the recognition of how to exploit interactions between research and practice stands as a key pillar of Software Engineering today.

It seems important to note that case studies of the ways in which software development practice and software engineering research have complemented each other, to the benefit of both, are being carried out as the focal point of the Impact Project [14]. The Impact Project's studies note that the impact of research upon practice tends to be indirect, and that research results rarely move quickly and directly from the research lab to industrial practice. Both researchers and practitioners need to continue to work to improve this situation. But it is also important to recognize that research findings have not uncommonly been the sources of change and innovation in industry, affecting agendas for improvements in practice just as surely as study of industrial practice has influenced research agendas.

Software Configuration Management (SCM) provides one very specific example. In [5] the development of SCM is traced from its early beginnings. SCM grew out of a need to support the management of complex software development projects that increasingly required the collaboration of many people, and the assembly of large numbers of software objects, often requiring the application of sequences of tools and systems. Clear conceptualizations of the nature of this problem came from careful study of the underlying problem. These led to early SCM systems. Their inadequacies were observed from their application in practice, leading to further cycles of study, understanding, and improvement. Other Impact Project studies are leading to similar conclusions, namely that research and

practice are mutually supportive, and that continued interactions between the two areas of endeavor seem to be most effective in leading to continued improvements in practice, as well as continued understandings of deeper issues.

3. The present

The foregoing helps to elucidate the dual nature of Software Engineering today. It encompasses two complementary, mutually supportive, types of activities, namely the development of tools and technologies to directly address the practical problems of the day, and the search for deeper understandings that can provide the basis for more effective tools and technologies. As indicated above, the synergy between these two types of activity has been effective in the past. Indeed, it has been necessary. There has been a clear need to develop a technology of software development, just as there had been a need for a technology of dealing with electricity, chemicals, machines, and airplanes. In these latter cases, this led to the development of Electrical Engineering, Chemical Engineering, Mechanical Engineering, and Aerospace Engineering. Each of these engineering disciplines, however, was based upon a base of science, provided largely by Physics and/or Chemistry. In the case of Software Engineering, however, this has not been possible, as none of the existing sciences seems to provide a satisfactory basis upon which to rest an effective technology for engineering software. Clearly Mathematics, especially Finite Mathematics has much to offer, but so do Sociology, Management, Psychology, and Epistemology. Thus, we have had to develop our own basis in science, drawing importantly upon a variety of existing scientific disciplines, and synthesizing as seems useful. As a consequence, we have seen the development within Software Engineering of both an activity that leans towards technology development, and an activity that leans towards scientific inquiry. The result has led to some notable successes.

The Software Engineering community should take great pride in the fact that Software Development is now one of the world's preeminent economic forces. The total value of software development products and services must certainly be measured at least in the many hundreds of Billions of (US) dollars annually. Software development is viewed as an industry that can bring wealth to nations, corporations, and individuals worldwide. Moreover, software now drives applications in virtually all areas of human endeavor. The traditional application of software to problems in such traditional areas as business and communications, for example, are now complemented by pivotal use of software in medicine, transportation, and even the arts and entertainment. Certainly the success of the software development community in meeting challenges raised by these many diverse areas must be viewed as a triumph of historic dimensions.

While these successes must be credited most directly to the practitioner community, it would be a mistake to ignore the contributions of software engineering research. The enormous volumes of code required to meet these challenges, for example, cannot be managed without such technologies as Configuration Management, and Impact Project studies, as noted above, demonstrate the indispensable contributions of research in this area. Superior software languages, management approaches, testing tools, and modeling methods all play similarly important roles. And in each of these areas software engineering research continues to provide pivotal insights, prototypes, and analyses that help move practice forward. Documentation of the contributions of research in these areas can also be found in additional Impact Project studies [7].

It is important to emphasize, in addition, that software engineering research continues to obtain a symmetric benefit from its contact with the community of software development practice. The development community brings to the research community a wealth of problems whose consideration continues to lead to important new areas of research. Indeed the flow of new problems from the community of practice has continually served to energize and rejuvenate the software engineering research community. The energy can be felt through a large and diverse number of workshops, symposia, and conferences now held more or less continuously all around the globe. There is also a large and growing archive of research papers published in a collection of magazines and journals that also continues to grow in size and scope. Despite periodic boom-and-bust cycles, employment opportunities for software engineers continue to grow, and opportunities for generous compensation for successful product innovations continue to deliver outsized wealth to a lucky few.

In summary, it seems that the important indicators of health and success for our community are all already very strong and yet steadily improving. It seems important, however, to consider where these indicators lead, and whether or not our future should be to simply extrapolate them forward.

4. One view of the future

The current picture, thus, of the software engineering community seems to be one of robust health, with research and practice interacting over a broadening range to the benefit of both, and to society. In the past we have developed notions such as modularity and encapsulation to enable us to generate ever larger numbers of machine instructions from

every line of source code written. We are now able to produce systems consisting of hundreds of millions of instructions. We have devised such notions as Software Configuration Management, Software Product Lines, Software Test Automation, and Software Development Environments, and have produced tools and systems to support them. This has enabled the global software development industry to field huge and complex systems, to maintain them in the field, and to evolve them in the development lab.

Pressures and incentives to continue in this way are very strong, and appear to be growing. The size and complexity of the software systems demanded by society continue to grow (exactly as noted by Perlis in 1968). Larger and more complex networks, more life critical applications, increased concern for privacy and security, all create new challenges, requiring research into new areas, and serving to further energize the research community. Our success in meeting technical challenges in the past emboldens us to take on these new challenges as well. Thus, software engineering research is increasingly examining such challenges as studying security and privacy in embedded systems, and ways to determine the range of possible behaviors of systems built out of distributed components, even when source code for some of the components is inaccessible.

Other new domains of investigation are now on the horizon and moving closer. We note, for example, a sharp increase in interest in software used in the medical, mechatronic [22] and automotive domains [20]. Some of the challenges faced in these domains seem relatively familiar, and amenable to approaches that our community has already developed. Other challenges will require ingenuity, and the marshalling of technologies and scientific insights from other domains (eg. realtime systems, database technology, human-computer interfaces, etc.).

In addition, it seems clear that software engineering will have much to gain from a more intimate set of interactions with researchers in the traditional sciences, especially the Life Sciences. It is increasingly clear that DNA sequences define the objects and processes that make living organisms work the way that they do. Life Scientists are increasingly trying to understand the large-scale behavior of devices (ie. living organisms) by looking at encodings of their low-level workings. In this way, they are in need of skills and technologies that Software Engineering has been struggling with for decades. Our technological expertise can help Life Science research. Conversely, the nature of the devices defined by DNA sequences far exceeds in complexity the computing systems that we have built and studied in the past. We have much to learn from joining Life Scientists in trying to understand (and modify) the systems that they are in contact with. Indeed, it is increasingly apparent that most of the traditional sciences are making increased use of computation and computational modeling as tools for pursuing their research. The problems that they are encountering seem to have the potential to stimulate growth in software engineering research, much in the ways that the problems encountered by business and industry have been providing our community with research inspiration in the past.

Thus, it seems clear that both traditional and new communities of practice will continue to push us to interact with them. Modes and mechanisms of interaction with these communities have been established, and will certainly be adapted and evolved to foster increasingly productive interactions in the future.

It seems important to consider, however, whether the directions of the past are suitable for the future. In particular, the development of scientific inquiry in Software Engineering in the past seems to have been driven primarily by consideration of practical problems arising from communities outside of our own. We have obtained clear benefits from this. But does it seem reasonable and appropriate for our community to look primarily outside of the community for sources of research inspiration?

5. A different view of the future

It may be time for the Software Engineering community to step back, examine the trajectories it has been following over the past decades, and think about whether these trajectories might be modified or augmented. We have noted that mature scientific disciplines such as Physics, Biology, and Chemistry all seem to have sprung from an increasingly elaborate and effective practice of addressing problems arising from the difficulties of the real world. While these various practices became increasingly effective in dealing with these problems, the growing successes increasingly highlighted and circumscribed areas in which success was less predictable and reliable. Thus, the practice of healing diseases arising from bacteria was dramatically improved with the discovery of antibiotics. But, viral diseases were completely resistant to this approach. This striking lack of success led to investigation of the nature of viruses and the opening of huge new vistas of scientific inquiry and discovery. In similar ways, early successes in metallurgy circumscribed some areas of failure, leading to the opening of new areas of chemistry. Is it time for Software Engineering, while taking great pride in what we have done so successfully in the past, to now step back and circumscribe areas that have continued to resist our best efforts, and to see if probing their nature can open large and important new areas of scientific inquiry?

Our past success in addressing problems arising

from the domain of practice can be labeled as problem-driven research. Perhaps it is time to supplement this type of research with a complementary type of research that we should call curiosity-driven research. While the goal of problem-driven research may be to return to the real world solutions to problems encountered there, and to answer questions arising there, the goal of curiosity-driven research is to identify questions of a deeper nature. Typically such questions arise from the minds of the researchers after long and serious grappling with the problems of the real world. Thus, for example, biologists eventually concluded that, rather than struggling to understand why antibiotics don't help with a long list of diseases, it might be more appropriate to instead try to understand what such diseases might have in common in order to devise a broader approach to all of them. The difference here is that the initiative for the establishment of a research direction such as this comes directly from the researchers in the community, and only indirectly from the practitioners in the domain.

Thus, there are two important rationales for curiosity-driven research. One is that this type of research has the potential to address a variety of problems across a broad range, rather than separate individual problems. The other, however, is that in pursuing curiosity driven research, the research community seizes direct control over the directions that it will be taking—it takes control of its own agenda. Mahoney [10] has observed that this is the very definition of a mature scientific discipline.

We can bring these rationales together with the single observation that mature scientific disciplines seem to all center around their study of a core of deep and enduring questions that have defied adequate resolution for very long periods of time (perhaps centuries), but whose continued dogged pursuit has led to innumerable findings of interest and importance. Indeed, it seems that it is the identification of these questions, and the elusive nature of their resolution that characterizes the highly respected mature scientific disciplines such as Physics, Astronomy, and Biology. Thus, for example, Physics continues to try to understand the nature of matter and energy, and their relationship to each other. Astronomy seeks to understand the origin of the universe. Biology seeks to understand what life is, and how living organisms work. Nobody expects quick and easy answers to these questions, but study of the many ramifications and manifestations of such questions has led to the growth of these disciplines, and the respect that they have earned. It is not insignificant to observe that these questions did not come from practitioners seeking help with their immediate problems. Rather they came from the minds of researchers looking for unifying understandings that could help in dealing with ranges of problems.

Perhaps it is time for Software Engineering to embrace the importance and timeliness of curiosity-driven research, as a complement to the problem-driven research that has driven us in the past, and must continue to be a driver in the future. Perhaps it is time for software engineering to seek the deep and enduring questions that can serve to define us as a discipline, while also leading to the fundamental understandings that can support more effective solutions to the problems that arise in practice.

Perhaps it is time for us to devote less of our energy to seeking answers to the questions raised by others, and more of our time and energy to seeking our own curiosity-driven questions.

6. Some questions

It would remiss if this paper did not at least suggest some examples of curiosity-driven questions that might serve the community of software engineering research. While it is not the purpose of this short paper to insist upon the suitability of these exact questions, and thereby attempt to define the direction for software engineering research, it seems appropriate to at least try to provide a few examples that might be food for thought. In that spirit, the following are suggested:

6.1. Question: What is design (the noun)? And how should it be performed (the verb)?

As noted above, these questions were indeed raised at the NATO conferences. Indeed at that time Naur suggested that "software designers are in a similar position to architects and civil engineers, particularly those concerned with the design of large heterogeneous construction, such as towns and industrial plants. It therefore seems natural that we should turn to these subjects for ideas about how to attack the design problem." Naur suggested that much might be learned from earlier scholarly investigations of design, such as the book by Alexander [1]. While much has indeed been learned by observing how other disciplines have pursued design and understood it, there is also a line of thought that suggests that software engineers might have some uniquely keen and useful insights of their own to offer. While civil engineers, for example, deal ultimately with tangibles, software engineers, because of the nature of their medium, are forced to deal exclusively with abstract non-tangibles. As design seems to be a non-tangible, perhaps software engineers might have some uniquely sharp and useful insights to offer about the nature of design, both the noun and the verb. Whether or not this may be true, it seems clear that those who have studied the development of software should be able to make an important contribution to understanding the nature of design, and

should take the pursuit of this issue as a key item on the community's research agenda.

It is important to emphasize that indispensable inspiration, intuition, and insight are to be drawn from the problems encountered in practice, and that invaluable contributions to practice will be made as a consequence of success in achieving deep insights. But investigation of the nature of design must not be driven entirely by the exigencies of practice, important though they may be. Curiosity about the nature of design in general, even though it may not have an immediate connection to practice, must be not simply tolerated, but encouraged and nourished. In that regard, it seems important to note that this volume itself contains a paper that explores the nature of software design, and its relation to the larger issue of design in general [24]. Papers dealing with curiosity-driven research of this kind must not be restricted only to the pages of special volumes such as this, but should become a staple of mainstream software engineering research venues, such as the proceedings of meetings such as the Foundations of Software Engineering and International Conference on Software Engineering as well.

6.2. Question: What is a model?

It is interesting to note that much of the work on such topics as software design, software architecture, and software requirements ultimately winds up dealing with models. While modeling seems to be an essential vehicle for the exploration of many software engineering topics, it seems to be a mistake to think of modeling solely as a manifestation of other activities. Perhaps it is also important to contemplate the question of what a model itself is, and what the activity of modeling is all about. As with the above question, it may be that software engineers, because of their long and deep involvement with non-tangibles, might have some special insights that might be less readily apparent to those who deal with models in the context of their work with tangibles such as cars, chairs, or buildings. Indeed, Plato, in his Allegory of the Cave [19], suggested that what is less directly observed and observable might, in an important sense, be more real, and what is being directly observed might, in that same sense, ultimately turn out to be less real. Software engineers often have a sense that the abstract models of the systems that they seek to build are often, in some (currently) largely intuitive sense, more valuable and useful than the (often flawed and inadequate) code that is most directly observable. As with Plato, the sense here is that the abstract, unimplemented model might actually be more real and important, while the directly observed code may be a more imperfect, and less satisfying, shadow of "reality". Perhaps Software Engineers should study Plato as part of their education?

It seems clear that the concept of a model is closely related to notions of abstraction, and thus that Software Engineers, who deal effectively with the creation and exploitation of abstractions, might have some especially keen and useful insights into the nature of models and modeling.

6.3. Question: How can we quantitatively measure software and assess its quality?

The problem of determining software quality was also enunciated at the 1968 NATO Conference, and it has been the subject of research continuously in the decades since then. Our continued inability to make much progress in coming up with measures that are scientifically sound and reliably effective in dealing with a broad spectrum of problems has continued to frustrate the community. It is interesting to note, for example, that attempts to measure even the most rudimentary software dimension, the size of a body of software code, have been frustrating. Ought we to measure the number of statements, "lines", characters, function points, generated machine code, etc.? Indeed an entire handbook [15] has been written to suggest the variety of different ways in which this seemingly easy dimension of software is to be measured.

Attempts to quantify quality have been even more frustrating. Indeed, it is not uncommon to note that quality is multidimensional, encompassing such attributes as correctness, efficiency, robustness, flexibility, etc. Thus it has often been suggested that it makes more sense to try to measure "qualities" rather than trying to measure the more monolithic "quality". But attempts to define measures of any of the dimensions of quality have proven to be equally frustrating.

As noted above, such continued frustrations should be taken as a clear indication that there is probably an underlying problem that is quite deep, and whose direct investigation might well lead to gratifying understandings and broad and important insights. The insights and understandings are likely to have important ramifications for improvements to practice. But here too it seems that this research might ultimately be more successful if it is curiosity driven, rather than driven by the exigencies of practice. The underlying problem here seems to be that the entity to be measured and assessed has no tangible manifestations at all. It seems far easier to measure and assess entities that have manifestations that are amenable to detection by the five traditional human senses. How are we to deal with the desire to measure something that is not detectable in these ways? Those whose work ultimately results in tangibles have tended to fall back upon the more familiar and comfortable measurement of these tangibles. Thus, for example, Mechanical Engineers seem to be far more successful

in measuring the mechanical devices that result from their work, than the designs they produce. Software Engineers have no such tangibles to measure, as the results of our work are non-tangible. We can indeed, and have in the past, attempted to assess the software that we build in terms of its effects upon the tangible world. But perhaps it is time for us to complement this indirect approach to measuring and evaluating software, and try a direct attack upon the harder problem of measuring the intangible directly. Indeed other types of entities such as laws, processes, designs, and recipes raise this same problem. The attempts of practitioners in these areas to quantitatively measure and assess their non-tangibles continue to lead to similar frustrations. Perhaps Software Engineering has something unique to offer. Perhaps it is time for us to address this problem directly.

6.4. Question: What is Software?

It has been puzzling to note that this question has somehow never been made the subject of serious inquiry by our research community. At ICSE 2002, this question was asked of more than 30 randomly selected conference attendees, and it was discovered that none had ever thought about it. Indeed, at the closing panel of that conference one of the most respected researchers in the software engineering community decried it as a question of dubious value. Yet, it seems odd that our discipline should be named as the engineering pursuit of something called "software", and yet no definition of "software" seems to be accepted by the community, and no research seems to have been addressed towards understanding the very nature of this entity/concept/??.

It is likely that there would be little objection to the suggestion that "software" contains elements of design, is somehow representable by models, and that it should be measured, and should be amenable to the quantification of its quality. Thus, the investigation of the preceding questions could potentially contribute to an understanding of what software really is. But there should also be room for a direct inquiry into the nature of software itself, in addition to inquiries into various of its parts and manifestations.

Perhaps one approach is to consider the natures of various of its manifestations, and the nature of their similarities and differences. Previous research has suggested that, for example, processes are software [13]. Indeed decades of research seems to confirm that computer software and process software have much in common. If this is the case, then what defines the type of the entity that we refer to as software, of which these two are subtypes? What characteristics can we infer from each by studying the other? What approaches to the development, verification of qualities, and evolution of one can help with the other, and what does all of this say about the supertype of both? Are there indeed other subtypes of software, and what might they tell us about all of this? These questions seem to be deep, enduring, curiosity-driven, and not likely to be of immediate interest to the community of practice. Exploration of these questions, however, might well lead to fundamental understandings that might indeed be of enormous value to the community of practice.

The pursuit of these questions could define an important current of software engineering research for decades, and could do much to place control of our community research agenda in our own hands.

7. The future of software engineering is in our hands

At the beginning of this paper it was suggested that our future is in our hands. The meaning of that statement should now be more clear. This paper has tried to make a case for the importance of curiosity-driven research, as a complement to the problem-driven research that has been predominant in the past. But it is the software engineering community that must decide upon the importance of this type of research, and must support it in tangible ways, such as accepting its legitimacy in its featured publication venues. Such acceptance is definitely not assured, however. It is clear, for example, that there will be an increasing flow of problems that originate in practice, and there is no doubt that there will be a steadily increasing need for our community to grapple with them. Correspondingly, there is a steadily increasing number of venues for the presentation of such papers, and the publication of archival research results originating from the domain of practice. If these publication venues choose to insist upon the publication only of results of problem-driven research, and if these venues decide to deny publication to curiosity-driven research, then the direction of our community will have been decided. Fortunately, these venues are all controlled by our own research community, however, and thus these decisions are not mandated upon us, but can be chosen by us. A pessimistic assessment of the current situation is that curiosity-driven research papers are rarely accepted by these publication venues, and that there seems to be a decided preference for papers driven by the needs of practice. There is nothing wrong with this, if the decision to favor such papers, essentially to the exclusion of papers of the other kind, is the result of careful consideration of what is best for the health of the community. On the other hand, there seems to have not been any debate about this, leading to the perpetuation of a research direction and trajectories that were begun decades ago.

Perhaps it is now time for a reconsideration of these

trajectories. It would be a great shame if the future of research in software engineering were decided by default and inertia, rather than by consideration of what we as a community think is best for ourselves. A debate on that subject is needed, and now seems to be an appropriate time for such a debate. The solicitation and appearance of this paper for this venue is a very encouraging sign. One can only hope that it signals the beginning of consideration of some fundamental questions that could form the core of an enduring scientific discipline of software engineering.

8. References

[1] C. Alexander, *Notes on the Synthesis of Form*, Harvard University Press, Cambridge, MA, 1964.

[2] R.W. Bemer, "Checklist for planning software system production", in P. Naur and B. Randell, eds., *Software Engineering, Report on a conference sponsored by the NATO SCIENCE COMMITTEE*, Garmisch, Germany, 7-11 October 1968. Scientific Affairs Division NATO, Brussels, Belgium, pp. 165-181. Also available at http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF.

[3] J.N. Buxton and B. Randell, eds., *Software Engineering Techniques, Report on a conference sponsored by the NATO SCIENCE COMMITTEE*, Rome, Italy, 27-31 October 1969. Scientific Affairs Division NATO, Brussels, Belgium, also available at http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1969.PDF.

[4] E.W. Dijkstra, "Complexity controlled by hierarchical ordering of function and variability", in P. Naur and B. Randell, eds., *Software Engineering, Report on a conference sponsored by the NATO SCIENCE COMMITTEE*, Garmisch, Germany, 7-11 October 1968. Scientific Affairs Division NATO, Brussels, Belgium, pp. 181-186. Also available at http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF.

[5] J. Estublier, D. Leblang, A. van der Hoek, R. Conradi, G. Clemm, W. Tichy, and D. Wiborg-Weber, "The Impact of Software Engineering Research on the Practice of Software Configuration Management", *ACM Transactions on Software Engineering Methodology*, 14, 4, Oct. 2005, pp. 383-430.

[6] S. Gill, "Thoughts on the sequence of writing software" in P. Naur and B. Randell, eds., *Software Engineering, Report on a conference sponsored by the NATO SCIENCE COMMITTEE*, Garmisch, Germany, 7-11 October 1968. Scientific Affairs Division NATO, Brussels, Belgium, pp. 186-189. Also available at http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF.

[7] J. Goguen, J. Thatcher, and E. Wagner, "An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types," in *Current Trends in Programming Methodology, V. 4, Data Structuring*, R. Yeh (ed.), Prentice-Hall, Englewood Cliffs, NJ, 1978, pp. 80–149.

[8] B.H. Liskov and S.N. Zilles. "Specification Techniques for Data Abstractions" *IEEE Transactions on Software Engineering*, v. 1, #1, 1975, pp. 7-19.

[9] A.I. Llewelyn and R.F. Wickens, "The testing of computer software", in P. Naur and B. Randell, eds., *Software Engineering, Report on a conference sponsored by the NATO SCIENCE COMMITTEE*, Garmisch, Germany, 7-11 October 1968. Scientific Affairs Division NATO, Brussels, Belgium, pp. 189-200. Also available at http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF.

[10] M. Mahoney, "Software as Science—Science as Software", in *History of Computing: Software Issues*, U. Hashagen, R. Keil-Slawik and A. Norberg (eds.), Springer-Verlag, Berlin, Germany, 2002, pp. 25-48. Also available at http://www.princeton.edu/%7Emike/softsci.htm.

[11] M.D. McIlroy, "'Mass Produced' software components", in P. Naur and B. Randell, eds., *Software Engineering, Report on a conference sponsored by the NATO SCIENCE COMMITTEE*, Garmisch, Germany, 7-11 October 1968. Scientific Affairs Division NATO, Brussels, Belgium, pp. 138-151. Also available at http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF.

[12] P. Naur and B. Randell, eds., *Software Engineering, Report on a conference sponsored by the NATO SCIENCE COMMITTEE*, Garmisch, Germany, 7-11 October 1968. Scientific Affairs Division NATO, Brussels, Belgium. Also available at http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF.

[13] L.J. Osterweil, "Software Processes Are Software Too, Revisited", *Proceedings of the 19th International Conference on Software Engineering* (ICSE 1997), Boston, MA, May 1997, pp. 540-548.

[14] L. Osterweil, C. Ghezzi, J. Kramer, and A. Wolf, "Editorial", *ACM Transactions on Software Engineering Methodology,* 14, 4, Oct. 2005, pp. 381-382.

[15] R. Park. "Software Size Measurement: A Framework for Counting Source Statements," Software Engineering Institute, Carnegie-Mellon University Technical Report # CMU/SEI-92-TR-020, ADA258304, Pittsburgh, PA. Also available at http://www.sei.cmu.edu/publications/documents/92.reports/92.tr.020.html.

[16] D.L. Parnas, "On the Criteria to be Used for Decomposing Systems into Modules", *Communications of*

the ACM, v. 15, #12, Dec. 1972, pp. 1053-1058.

[17] A.J. Perlis, Keynote speech, 1968 NATO Conference, in P. Naur and B. Randell, eds., *Software Engineering, Report on a conference sponsored by the NATO SCIENCE COMMITTEE*, Garmisch, Germany, 7-11 October 1968. Scientific Affairs Division NATO, Brussels, Belgium, pp. 135-138. Also available at http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF.

[18] T.B. Pinkerton, "Performance monitoring and systems evaluation", in P. Naur and B. Randell, eds., *Software Engineering, Report on a conference sponsored by the NATO SCIENCE COMMITTEE*, Garmisch, Germany, 7-11 October 1968. Scientific Affairs Division NATO, Brussels, Belgium, pp. 200-204. Also available at http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF.

[19] Plato, *The Republic, Book VII*, 360BC. Translated by Benjamin Jowett, P.F. Collier, New York, copyright 1901 The Colonial Press. Also available at http://www.ilt.columbia.edu/publications/plato_republic.htm, Markup, Copyright 1995, Institute for Learning Technologies.

[20] A. Pretschner, M. Broy, I. Krüger, T. Stauner: "Software Engineering for Automotive Systems: A Roadmap", in *Future of Software Engineering 2007*, L. Briand and A. Wolf (eds.), IEEE-CS Press, 2007.

[21] B. Randell, "Towards a methodology of computing systems design", in P. Naur and B. Randell, eds., *Software Engineering, Report on a conference sponsored by the NATO SCIENCE COMMITTEE*, Garmisch, Germany, 7-11 October 1968. Scientific Affairs Division NATO, Brussels, Belgium, pp. 204-209. Also available at http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF.

[22] W. Schäfer, H. Wehrheim, "The Challenges of Building Advanced Mechatronic Systems", in *Future of Software Engineering 2007*, L. Briand and A. Wolf (eds.), IEEE-CS Press, 2007.

[23] C. Strachey, in J.N. Buxton and B. Randell, eds., *Software Engineering Techniques, Report on a conference sponsored by the NATO SCIENCE COMMITTEE*, Rome, Italy, 27-31 October 1969. Scientific Affairs Division NATO, Brussels, Belgium, pp. 9-12. Also available at http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1969.PDF.

[24] R.N. Taylor and A. van der Hoek, "Software Design and Architecture: The Once and Future Focus of Software Engineering", in *Future of Software Engineering 2007*, L. Briand and A. Wolf (eds.), IEEE-CS Press, 2007.