Software Processes Are Software Too, Revisited: An Invited Talk on the Most Influential Paper of ICSE 9 *

Leon J. Osterweil
University of Massachusetts
Dept. of Computer Science
Amherst, MA 01003
USA
+1 413 545 2186
ljo@cs.umass.edu

ABSTRACT
The ICSE 9 paper, "Software Processes are Software Too," suggests that software processes are themselves a form of software and that there are considerable benefits that will derive from basing a discipline of software process development on the more traditional discipline of application software development. This paper attempts to clarify some misconceptions about this original ICSE 9 suggestion and summarizes some research carried out over the past ten years that seems to confirm the original suggestion. The paper then goes on to map out some future research directions that seem indicated. The paper closes with some ruminations about the significance of the controversy that has continued to surround this work.

Introduction
"Software Processes are Software Too." How many times I have heard that phrase quoted back to me in the past ten years! And how many times it has been (sometimes amusingly) misquoted too. Often I have been flattered to have had the ICSE9 paper [15] and its catchy title referred to as being "classic" and "seminal". But often I have also been asked, "what does that really mean?" The idea is, alas, still misunderstood and misconstrued in some quarters. But amazingly, and gratifyingly, the phrase is still used, and the discussion of the idea still continues, even after ten years.

The suggestion that software, and the processes that deal with it, might somehow be conceptually similar remains a powerfully appealing one that seems to have led to a considerable body of investigation. The suggestion was immediately controversial, and continues to be argued. Subsequently I discuss why I believe this discussion indicates a pattern of behavior typical of traditional scientific inquiry, and therefore seems to me to do credit to the software engineering community.

But what of the (in)famous assertion itself? What does it really mean, and is it really valid? The assertion grew out of ruminations about the importance of orderly and systematic processes as the basis for assuring the quality of products and improving productivity in developing them. Applying the discipline of orderly process to software was not original with me. Lehman [13] and others [18] had suggested this long before. But I was troubled because I had started to see the development of a whole new discipline and technology around the idea of software process, and to notice the emergence of many notions and tools that seemed eerily familiar. I was starting to see the creation of a software process universe parallel to the universe of notions and tools surrounding application software development. The more I looked, the more similarities I saw. Processes and applications are both executed, they both address requirements that need to be understood, both benefit from being modelled by a variety of sorts of models, both must evolve guided by measurement, and so forth. Thus it seemed important to suggest that software process technology might not need to be invented from scratch (or reinvented), but that much of it might be borrowed from application software technology.

I have often been reminded that application software technology is still badly underdeveloped and that using it as a model for software process technology might be of dubious value. This, however, overlooks clear evidence that, while we have not mastered application software technology, we have, nevertheless, created a powerful assortment of tools, principles, and techniques in this domain. Thus, there is much to be gained from using obvious parallels to hasten the maturation of software

*This work was supported in part by the Air Force Materiel Command, Rome Laboratory, and the Defense Advanced Research Projects Agency under Contract F30602-94-C-0137.

process technology. It seemed important to suggest that the community should look to the more traditional and better-developed disciplines of application development to see what might be borrowed or adapted. It seemed clear that there were strong similarities, but likely that there were differences as well. Investigation of the extent of each seemed to be in order. The ICSE 9 talk invited community investigation of how processes and application software are the same and how they differ, so that relevant findings, approaches, and tools of one could be of use to the other. It has been gratifying to see that this invitation has been taken up and that these explorations are still ongoing.

Conversely it has been disappointing to see the way in which the suggestion has continued to be misconstrued in some quarters. Subsequent sections will deal with these misconceptions in more detail, but the following brief summary seems in order here.

Software is not simply code. Neither are software processes. Application software generally contains code. This suggests that software processes might also contain code. Coding software processes thus seems to be an interesting possibility. Research has borne this out.

Programming is not the same as coding, it entails the many diverse steps of software development. Software process programming should, likewise, not simply be coding, but seemed to entail the many non-coding steps usually associated with application development. Process modelling, testing, and evolution research seems to have borne that out.

There are many examples of application code that are not inordinately prescriptive, authoritarian, or intolerable to humans (eg. operating systems). Thus, there should be no presumption that process code must be overly prescriptive, authoritarian, or intolerable either. Process programs need not treat humans like robots– unless that is the intention of the process programmer. Process modelling and coding languages demonstrate this.

Finally, good software code is written at all levels of detail. Code contains fine scale details, but they emerge at lower levels, after high level code addresses larger issues. Similarly process code contains details that are nested below higher abstract levels. Process code, like application code, can demonstrate that precise implementation of broader notions in terms of lower level engineering details. Contemporary process coding languages demonstrate this too.

The following section summarizes some research that suggests continued and broadened research into these issues.

Parallels Between Software Processes and Application Software

Much work seems to demonstrate the existence of significant parallels between software processes and application software, although not all of this work was intended to do so. This section briefly surveys what has been learned.

Process Modelling

There has been a great deal of study of how well various application software modelling formalisms model software processes. For example, Petri Nets [1], [5], Finite State Machines [6], [11], and data flow diagrams [19] have been used to model software processes. These activities have clearly demonstrated that application software modelling approaches can be strong aids in conceptualizing processes, in helping people to communicate about processes and collaborate in their execution, and in raising intuition about processes.

As with application software modelling, different types of process models are good for different things. Petri Net models, for example, are quite useful in elucidating parallelism and concurrency, but are less useful in modelling artifacts. Petri Nets process models seem to have very similar properties. They help to identify parallelism in processes, but have generally required augmentation in order to effectively elucidate the flow of software artifacts through processes. Other similar examples could readily be pointed out.

In general, models, by their nature, abstract away details in order to focus on specific narrow issues, which are thereby made correspondingly clearer and more vivid. Thus, Petri Net models depict parallelism clearly in part because depictions of other less relevant details are specifically omitted. Thus, any particular model should be expected to be useful in some contexts, but less helpful in others. To support understanding of various aspects of a software product different models are generally needed. Thus, a number of modelling systems (eg. [6]) support the development and coordination of multiple models of application software. Experience in the software process domain has been similar. Statemate was used as a process modelling tool [11], and its support for multiple models was useful precisely because the different models supported understanding and reasoning from a variety of aspects. In the application software domain there is a growing understanding of which modelling tools and formalisms best elucidating which issues. We expect similar understandings to emerge in the software process domain.

But, as with application software modelling, it has also become clear in process modelling that there are reasons why models, even multiple models, are sometimes inadequate. The very lack of certain types of details

in models means that models inevitably lack specifics that can be very important. In addition, many modelling formalisms (eg. graphical models) are based upon weak and shallow semantics. Because of this it is usually impossible or unsafe to reason about such models. Models expressed in a formalism with a weak semantic base may convey an intuitive impression, but they usually cannot support precise, reliable reasoning. For example, many modeling notations (especially graphical notations) can indicate parallel activities, but offer no semantics for defining the precise nature of the parallelism. This lack of semantics leaves human interpreters free to suppose whatever form of parallelism they like. Inevitably this leads different interpreters to different conclusions about what the model represents. The result is often miscommunication and misunderstanding. Where the intent of the model was presumably clarity, the effect will have been quite the opposite. Even where the semantics of such constructs as parallelism are incorporated in the modelling formalism, it is unusual for there to be much variety in the sorts of parallelism. This semantic sparseness usually causes such formalisms to be inadequate to depict the full range of parallel constructs needed to represent the full range of parallelism that process modelling seems to require. Thus, there seems to be a growing understanding that models of processes meet some needs (eg. raising one's intuition about processes), but that there are more needs that are unlikely to be met by single process models, or even combinations of process models.

Process Coding

While there is good evidence that processes need to be represented by executable code, as well as by models, as in the case of application code, it is difficult to draw a sharp distinction between coding languages and modelling languages. Certain coding languages are imprecise about the execution semantics of certain constructs, and certain modelling languages have very precise execution semantics. There are often disputes about whether particular application languages should be considered to be coding or modelling languages. The process community has experienced similar disputes and disagreements about process languages during the past years.

Such disputes are unproductive. The important distinctions among these languages are the nature, depth, and scope of the semantic details that they provide. As noted in the previous sections, modelling formalisms tend to offer relatively weak, shallow, or narrow semantics. Thus, while a strong modelling formalism may support deep and reliable reasoning about a narrow aspect of the software it models, such formalisms are at best helpful only in narrow contexts. When broad categories of powerful precise, reliable reasoning is required stronger, broader semantics and greater detail are essential. In reasoning, for example, about the presence or absence of deadlocks and race conditions in processes it is essential for the process to be defined in a formalism that supports precise definition of parallelism and shared access to data and resources. The semantics needed to support such reasoning must be quite precise and powerful, and are generally consistent with semantics found in full coding languages, rather than in modelling languages. Processes, like applications, at times benefit from the existence of codelike representations that offer a wide range of semantic power and definition detail. At some times the detail will be undesirable, interfering with clarity and intuition. But at other times it will be essential as the basis for effective reasoning and actual execution.

There are other reasons why it is important to reduce software to code. Application developers know that, until software has been coded, it is unknown whether the possibly myriad models that have preceded it can actually be reduced to practice. Similarly a set of software process models may depict an enticing view, but can still leave open to question just how a process consistent with all of those views will actually work. It is the interplay of all of the details, both present and absent, from all of the models that characterizes and defines the actual application or process. Only a language that can specify and interrelate all of these details can support definitive demonstrations of the realizability of the desired product. In short, real code provides real assurances; models mostly provide enticements.

The original ICSE 9 paper emphasized yet another reason for defining processes in coding languages. That paper suggested that processes should be viewed as prescriptions for the synergistic coordination of the efforts of humans, computers, and software tools. Process code was suggested as the vehicle for specifying the precise details of this coordination. Because coding languages have executable semantics, the paper suggested that computers could execute such code and could, in doing so, supervise the integration of the efforts of people, machines and tools.

This point has been the subject of much unfortunate misinterpretation and caricature. Careless reading of this point has taken it to suggest that all processes could, or should, be reduced to computer executable instructions. This was neither the intent nor the proposal of the original paper. Indeed, the paper stated that software development processes should refrain from elaborating the details of how humans should carry out their tasks. Human tasks should be represented as functions or procedures for which the definition is omitted, thereby leaving the human free to execute the task as he or she sees fit. The level to which any human task

is elaborated by the process code is the choice of the process coder, who, in doing so, specifies the extent to which the process is authoritarian and prescriptive, or permissive and relaxed.

The vehicle of process code is thus not a device for dictating what a human must do, but rather a vehicle for specifying the degree to which human activities are to be circumscribed by the defined process. The act of defining the process by executable code does not necessarily unduly restrict the human, although the nature of the code may do so. Indeed, the JIL [23] language, is an example of a process coding language that supports considerable latitude in the degree of specificity of process definition.

Here too, experience suggests that a wide range of process coding languages and coding styles seem to be of value. Less detailed process code is preferable, for example when the process is to be performed by seasoned experts who can exercise good judgement in devising sequences of process steps to carry out a task. More detailed and precise process code is of value in other circumstances, for example in restricting and regulating the work of software developers who are novices, or whose actions may eventually be subject to careful scrutiny (as, for example in the case where an organization wishes to provide protection against possible subsequent legal claims of carelessness in software development).

As suggested above, detailed process code specifications are also of particular importance in specifying how tools and automated procedures are to be integrated into processes, and how the activities of humans are to be coordinated with them. This requires precise specifications of how various software artifacts are to be fed into tools and extracted from their outputs, precise specification of how such artifacts are to be made available to the right humans at the right time, and how human development artifacts are to be channeled to the right colleagues and tools. All of this requires a great deal of precise specification that is consistent with the levels of detail and precision found in the executable semantics of coding languages.

Experimental research of the past few years seems to confirm that coding languages are particularly adept at expressing the specifics of the interactions of process steps and software artifact operands, while modelling languages tend to be particularly ineffective at this. Modelling languages tend to focus on either activity or artifact modelling, thereby failing to support the choreography of artifacts through tools and humans. Coding languages tend to be superior in this regard.

Thus there seems to be considerable evidence that software processes require and benefit from both modelling and coding for very much the same reasons that software applications benefit from both of these activities.

Process Evaluation

There is also considerable evidence that software processes are amenable to evaluation using approaches that bear important similarities to the approaches used in evaluating application software. Indeed, the past ten years have witnessed explosive growth in work on the evaluation of software processes. Most of this work has grown out of the proposal of Humphrey and his colleagues at the Software Engineering Institute, of the Capability Maturity Model (CMM) [7] [16]. The aim of the CMM is to provide an evaluation vehicle for determining the quality of an organization's software development processes. Organizational software process evaluation is done by a variety of means, but is usually based upon questionnaire-based surveying, and by examination of the artifacts produced by past software development projects.

Although the CMM does not take the explicit position of viewing software processes as software, it seems useful for us to do so. Taking the position that an organization has a process that it executes in order to develop its products leads to the conclusion that such products are reasonably viewed as the outputs of the execution of that process. If the quality of the process is evaluated through examination of such outputs, then doing so is essentially a testing activity. This leads us to conclude that the CMM establishes the structure for a process testing regimen, and that such instruments as the CMM-based questionnaires function as process test plans.

These observations demonstrate that testing and evaluation of software processes has been a prevalent activity over the past several years, even despite the fact that explicit process representations may not have been available. The lack of explicit process definitions forces the process evaluator to examine output artifacts, and to take a testing-like approach to process evaluation. From our perspective of viewing processes as software, we suggest that this is analogous to testing quality into the software process. Experience in such fields as manufacturing suggests that it is preferable to build quality in, rather than test it in. Building quality into processes seems to require the explicit representation and definition of the processes. We view this as yet another key reason why processes should be precisely defined using formalisms with strong semantics.

Indeed, going one step further, we observe that carrying out a CMM-based evaluation or assessment is in fact the execution of a process-testing process. As such, this sort of process too should be amenable to specification by process formalisms. Such formally specified

process-testing processes are examples of higher-order processes that should be developed for the evaluation of processes and the feeding back of such evaluations as part of larger process improvement processes. Formal specifications of such higher-order processes should facilitate more precise and sure reasoning about these key processes. These ideas are developed more fully in [14].

Process Requirements

The observed parallels between modelling, coding, and evaluating application software and software processes might suggest that similar parallels have been demonstrated between application requirements specification and process requirements specifications. It is startling to note that such parallels have not been demonstrated yet due to an apparent lack of interest in studying software process requirements.

Especially in view of the intense interest in supporting the modelling, coding, and evaluation of processes, it seems almost bizarre that there has been virtually no work in supporting the specification of process requirements. Indeed early suggestions that more attention be focussed on process requirements sometimes brought disbelief in the very existence of process requirements. The enterprise of process modelling should instantly raise in trained software engineers important questions about the validation of such models. That, in turn should suggest that the models are there in large measure to demonstrate that a particular process approach is effective in addressing certain process problems, as enunciated in a set of requirements.

Software processes generally have clear (albeit unstated) performance requirements, (eg. deadlines for completion of the entire process or various of its steps). Further, these requirements function very much in the way that application software requirements do, often influencing important choices about what steps to parallelize, in which way, and to what degree. Similarly processes often have robustness requirements, specifying how processes must react to such adverse situations as the loss of key personnel or artifacts. Replication, redundancy, and backups are the standard application software approaches to addressing these requirements, and they are also process architecture approaches to similar process requirements. Processes also have functional requirements, for example specifications of the range of software artifacts to be produced as the final output of the process, and the nature of the required demonstrations of internal consistency.

Despite these rather obvious types of process requirements, and the fact that they should function as essential baselines against which to measure both process models and process test plans, there has been virtually no interest in developing and using process requirement formalisms. Thus, although the parallelism between application software requirements and software process requirements seems apparent, there has been scant research to demonstrate it. This seems to be an area that is very much in need of considerably more investigation.

Looking Ahead

It seems increasingly clear that the weight of evidence is supporting the hypothesis that software processes are indeed very much like application software in many important ways. That being the case, we should expect to be able to exploit the similarities in a number of ways. The previous section has suggested some of these ways. In this section we suggest some others.

Programming Key Processes

It seems clear that it is time to get on with the important work of developing models and code of key software development processes. There are important benefits to be gained from this. Software engineering (indeed any sort of engineering) has as two of its key objectives the reduction of costs and the improvement of the quality of products. Processes play a key role in both of these. As software costs derive almost exclusively from the cost of human labor, cost reduction must come from reduction in labor. Explicit software process representations can be analyzed to identify needless and unproductive human labor, and to identify process steps that might be performed by automated devices. Both then lead to reductions in labor costs. Further, as noted above, quality is generally understood to be built into products through periodic careful testing and analysis of the product as it evolves through the development process. Here too, explicit process representations should be effective bases for identifying where and how to carry out these periodic tests and analyses.

Thus, the development, demonstration, and reuse of demonstrably superior software processes still remains the goal that it was as enunciated in the original ICSE 9 paper. However, now, ten years later, we should have greater assurance that this goal is achievable, and a weight of experimentation suggesting how to proceed. We have demonstrated a variety of modelling formalisms (largely borrowed from application software technology). We have also begun to understand the demanding requirements of process coding languages. But, here application software coding languages have proven less useful. Experimentation has demonstrated the value of various programming paradigms, such as the procedural programming paradigm (eg. with the APPL/A language [22]), the rule based programming paradigm (eg. with MSL, the Marvel Specification Language [8]), and real-time programming approaches (eg. with Adele [2]). But this experimentation has also shown the inadequacy of each of these by itself. Experience has shown that representing in a clear and straight-

forward way all of the details and complexities of software processes by means of a language with executable semantics is far more difficult and challenging than was expected ten years ago. Second generation languages such as JIL [23], which enable the blending of the benefits of various programming language paradigms, seem to hold promise. More experimentation and evaluation of such languages is clearly indicated.

In order for the cost and quality improvements mentioned above to be realized, execution engines for such languages will have to be developed. Recent research is leading to understandings that such engines must have highly flexible distributed architectures. The Amber project [9], and the Endeavors project [3] offer good examples of such architectures. These projects should provide encouragement to believe that the superior process code to be written in the new generation of process coding languages will be effectively executable to provide the sort of strong support needed to reduce costs and improve quality in developed software.

Once these languages and execution engines are in place the development of exemplary software processes should begin. Some examples of processes that should greatly benefit from such encodings are: processes for collaborative design, processes for integrated testing and analysis, processes for configuration management, processes for tracking bug fixing, and processes for effecting successful reuse. Indeed, the last ten years has seen a growing awareness of the broad range of processes that are executed in the course of developing software. As these processes have been more clearly identified, they have become increasingly important targets for understanding and improvement. Detailed encodings should support reliable analyses and detailed dynamic monitoring of these processes that should then lead to the kinds of deep understandings that are needed in order to effect improvements reliably.

Creating a practice of software process engineering that will lead to reliable techniques for systematic improvements to processes through engineering of process program artifacts is clearly some distance off. But the progress of the past ten years seems to indicate that it is still a worthy goal, and to justify greater belief in assertions that it is definitely achievable than could have been justified ten years ago.

Scientific Classification and Comparison of Software Processes

While most of the process technology research of the past ten years has focussed on supporting the synthesis of new processes, there has also been an important demonstration of the use of process technology to support the analysis of existing processes. As noted above, there has been a growing recognition of the number and diversity of processes in use to support software development. Thus, designing, debugging, requirements specification, and configuration management have come to be recognized as key software development processes. In some of these areas, for example software design, there has been a long history of suggested approaches to performing the process. These suggestions have all too often taken the form of imprecise and/or incomplete prose articles, courses, and books, often consisting largely of examples. Attempts to compare and contrast these suggested software design approaches have been reduced to similarly informal, often anecdotal, treatments of the various approaches. The lack of definitive, precise characterizations and comparisons of these design approaches frustrates practitioners who must choose from among them, and impedes progress towards the establishment of a scientific discipline of software engineering.

Regarding software design as a process that can be expressed in precise modelling and coding formalisms seems to help considerably. This perspective suggests that the writings about various software design approaches might be considered to be specifications of requirements and/or architectures of contrasting software design processes. It further suggests that detailed models and encodings of these processes, using formalisms that are based on precise and deep semantics, can be bases for correspondingly precise characterizations, classifications, and comparisons.

A series of papers published over the past five years demonstrates the viability of this approach [20, 21, 17]. In these papers popular software design methods (SDM's) are modelled using popular software process modelling formalisms (eg. HFSP [10] and Slang [1]). Comparison frameworks are hypothesized to guide classification of SDM features. A carefully defined SDM comparison process is executed to extract comparison results from the classifications of the models of the SDM's. The papers demonstrate that this approach can be used to produce classification and comparison results that agree with and extend classifications and comparisons arrived at based on informal models and comparison techniques. The precision and specificity of both the models and the comparison process itself (it is a process programmed in process modelling and coding languages) suggest that these classification and comparison results are reproducible by different human comparators.

Work in this area is just now beginning to proliferate, and it seems that this kind of work could be most critical to fostering the maturation of software engineering. If software engineering is to mature into an orderly discipline it seems that it must develop a core set of well-understood, well-supported standard processes, and a cadre of practitioners who understand what the pro-

cesses are and how to use them. Certainly the older, better established engineering disciplines, such as Chemical Engineering and Industrial Engineering, exemplify this sort of use of process. In order for such a core set of standard processes to emerge there must be a considerable amount of differentiation and sorting out of the processes that are currently in existence, and an orderly way of dealing with the steady flow of new process proposals, especially in such active areas as software design.

The work just described seems particularly promising because it suggests that structures and processes can be put in place that will serve to support standardized comparisons and evaluations of the processes that must form the core of disciplined software engineering practice. It is unfortunate that debates about the relative merits of different approaches to such key activities as software design are currently argued in the advertising pages of *IEEE Software*, rather than in the scholarly works of *IEEE Transactions on Software Engineering* or *ACM Transactions on Software Engineering Methods*. If our discipline is to mature satisfactorily that must change. The frameworks and processes suggested in the papers referred to above are suggested initial starting points, and it can only be hoped that the community will take them as such and work collaboratively to develop them into agreed upon standards. With such standards in place it should then be possible for objective evaluators to produce specifications and descriptions that characterize clearly and understandably the merits of competing methods. Such evaluations should also then be usable in estimating the costs and results of applying these methods.

This suggests a line of experimental research that focuses on performing software engineering process classifications and comparisons, but with an eye towards evaluating the standard classification frameworks, the standard process modelling formalisms, and the standard comparison process. Evolution of all of the above is to be an expected outcome of this experimentation. A steadily growing and improving stream of classifications, characterizations, and comparisons should also result. This line of research seems to be converging interestingly with research being done by the Method Engineering community (see, eg. [4]).

Beyond Software Engineering

In examining the hypothesis that software processes are software, there seems to be nothing particularly special about *software* processes. This suggests a hypothesis that processes in general are also software. Confirmation of that hypothesis would be of particular interest as it would suggest that application software technology can also help support the development and evolution of all kinds of processes. In particular it suggests that software engineers might have something of particular value to offer those who engineer manufacturing systems, management systems, classical engineering systems, and so forth. A variety of private conversations and preliminary investigations seem to confirm that these systems often have (or should have) architectures, that they are intended to satisfy understood requirements, and that their implementations are generally on virtual machines consisting of people, devices, and computers. In addition, these systems are usually continuously being evaluated and evolved. All of this suggests that they are software in the same sense in which we believe that software processes are software. That being the case, it suggests that software process researchers ought to widen their sights and study the applicability of the emerging software process technology to manufacturing, management, and allied disciplines.

Conclusions

The foregoing sections of this paper have been intended to suggest that there are numerous technological benefits from considering software processes to be software, and that examining them should lead to a considerable amount of worthwhile research. But there is yet another aspect of this work that seems worth remarking upon, and that is its contribution to the scientific underpinnings of software engineering. It was clear from the moment I concluded the original talk at ICSE 9 that the suggestion that software processes might be software had initiated a type of discussion that was different from other discussions following other papers that I had given. The substance of the discussions and debates that have followed has rarely been at the level of technical details, but rather at more philosophical levels. There were debates about whether it was seemly or possible to use the rigorous semantics of programming languages to describe what people did or should do. There were debates about whether processes were a subtype of application software, or vice versa. There were debates about whether processes have a different character than applications.

The distinguishing characteristic of most of these debates has been the fact that there did not, and still does not, seem to be much possibility that these debates and questions can be resolved definitively. One reason is that there is no agreed upon definition of what software is. Likewise there is no firm agreement on what programming is, or what a process is for that matter. Thus, the debates and discussions that have swirled around the original suggestion have been largely philosophical, and the opinions expressed have been based largely upon personal aesthetics. The suggestion that software and processes are made out of basically the same stuff sets well with some people, and not so well with others. The suggestion implies that what we know and can learn about one transfers to some extent over to

the other. This suggestion has obvious importance for the technologies in these two areas, but this has been met with skepticism and reticence in some quarters.

Skepticism, reserve, and the impossibility of definitive adjudication of these questions, however, should not be allowed to obscure what seems to be the most significant implication of the suggestion, namely its potential to shed some light on the nature of software itself. If it is shown that software is highly akin to something else about which we can have a variety of new and different insights, then those insights illuminate the nature of software. Thus, in the debates about the relationship between process and software I see the reflections of a broader debate about the nature of software. In that software engineering purports to be a discipline devoted to the effective development of software, it seems essential that we as a community have a shared view of what software is. Debates such as these, that help lead to that shared view, are critically important.

In his renowned book, *The Structure of Scientific Revolutions* [12], the historian of science, Thomas S. Kuhn, suggests that progress in a scientific discipline is discontinuous, progressing incrementally within the bounds circumscribed by the current paradigms, but then lurching forward occasionally when a new paradigm is agreed to account better than the old paradigm for natural phenomena or to provide more assistance in solving practical engineering problems. Kuhn argues that the old and new paradigms are generally mutually incompatible and that, therefore, it is impossible to use either to prove the falsity of the other. Thus shifts from an older paradigm to a newer paradigm generally take place over a period of time during which there is considerable intellectual ferment and philosophical dispute. If the new paradigm is to supplant the older paradigm it will happen only after careful research has demonstrated that the new paradigm is more robust and successful than the old paradigm. After the shift has occurred, the shape of the science, its view of its problems, and the manner of its approaches and explanations will have been substantively changed. Most practitioners will accept the paradigm shift, but adherents to the old paradigm may persist.

It seems just possible that what we have been witnessing is a slow paradigm shift to a view of software and software development that is rooted in the centrality of the notion of process as a first-class entity whose properties are very much like those of software itself. The nature of the debates that we have been witnessing are consistent with what would be expected if this were the case, being essentially discussions that are based more on aesthetics than upon the ability to perform definitive demonstrations. As the accretion of evidence of the power of a process-centered view of software grows it seems conceivable that we are seeing the establishment of a new paradigm. The preceding discussions in this paper do seem to suggest that grasping the importance of process, and exploiting its relation to software, does help deal more effectively with important technological and conceptual issues. Pursuing the research agenda outlined in the previous section should go a long way towards confirming this suggestion or to demonstrating its inadequacy. In either case it seems most encouraging to observe that the intense debates and discussions of the premise that "software processes are software too," seems to be quite consistent with the behavior of a responsible community of scientists doing real science. Ultimately this affirmation of our growing maturity as a scientific community may be the most important outcome of the proposal and ensuing discussions.

Acknowledgments

My ideas and work on software processes has been greatly helped and influenced by many people, indeed too many to mention here. The earliest impetus for the ideas of process programming arose out of meetings and conversations with Watts Humphrey and his team at IBM in the early 1980's. The specific proposal of the notion of process programming was honed and sharpened through many conversations with Manny Lehman at Imperial College and John Buxton at Kings College, London in 1985 and 1986. Confidence in the idea was built through intense conversations with many people, but most notably with Dennis Heimbigner. Over the past ten years I have been fortunate to have been able to collaborate with Stan Sutton and Dennis Heimbigner on process programming language design and implementation and Xiping Song on software method comparison formalization. Numerous conversations with Dick Taylor, Bob Balzer, Gail Kaiser, Alex Wolf, Dewayne Perry, Mark Dowson, Barry Boehm, Wilhelm Schafer, Carlo Ghezzi, and Alfonso Fuggetta have also shaped this work in important ways. I would also like to thank the (Defense) Advanced Research Projects Agency for its support of this work, and particularly Bill Scherlis, Steve Squires, and John Salasin for their support, even while these ideas were formative and while they continue to be controversial.

REFERENCES

[1] S. Bandinelli, A. Fuggetta, and S. Grigolli. Process modeling in-the-large with SLANG. In *Proc. of the Second International Conference on the Software Process*, pages 75 – 83, 1993.

[2] N. Belkhatir, J. Estublier, and Walcelio L. Melo. Adele 2: A support to large software development process. In *Proc. of the First International Conference on the Software Process*, pages 159 – 170, 1991.

[3] G. A. Bolcer and R. N. Taylor. Endeavors: A process system integration infrastructure. In *Proc. of the Fourth International Conference on the Software Process*, pages 76 – 85, Dec. 1996.

[4] S. Brinkkemper, K. Lyytinen, and R. J. Welke. *Method Engineering*. Chapman & Hall, New York, 1996.

[5] V. Gruhn and R. Jegelka. An evaluation of FUNSOFT nets. In *Proc. of the Second Eurpoean Workshop on Software Process Technology*, Sept. 1992.

[6] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. STATEMATE: A working environment for the development of complex reactive systems. *IEEE Trans. on Software Engineering*, 16(4):403 – 414, Apr. 1990.

[7] W. S. Humphrey. *Managing the Software Process*. Reading, MA:Addison-Wesley, 1989.

[8] G. E. Kaiser, N. S. Barghouti, and M. H. Sokolsky. Experience with process modeling in the MARVEL software development environment kernel. In B. Shriver, editor, *23rd Annual Hawaii International Conference on System Sciences*, volume II, pages 131 – 140, Kona HI, Jan. 1990.

[9] G. E. Kaiser, I. Z. Ben-Shaul, S. S. Popovich, and S. E. Dossick. A metalinguistic approach to process enactment extensibility. In 4TH INTERNATIONAL CONFERENCE ON THE SOFTWARE PROCESS *(to appear)*, Dec. 1996.

[10] T. Katayama. A hierarchical and functional software process description and its enaction. In *Proc. of the 11th International Conference on Software Engineering*, pages 343 – 353, 1989.

[11] M. I. Kellner. Software process modeling support for management planning and control. In *Proc. of the First International Conference on the Software Process*, pages 8 – 28, 1991.

[12] T. S. Kuhn. *The Structure of Scientific Revolutions*. University of Chicago Press [Chicago], 1962.

[13] M. M. Lehman. The Programming Process. In *IBM Res. Rep. RC 2722*, IBM Res. Center, Yorktown Heights, NY 10594, Sept. 1969.

[14] Leon J. Osterweil. Improving the quality of software quality determination processes. In R. Boisvert, editor, *The Quality of Numerical Software: Assessment and Enhancement*. Chapman & Hall, London, 1997.

[15] L. J. Osterweil. Software Processes are Software Too. In *Proceedings of the Ninth International Conference of Software Engineering*, pages 2–13, Monterey CA, March 1987.

[16] M. C. Paulk, B. Curtis, and M. B. Chrisis. Capability maturity model for software, version 1.1. Technical Report CMU/SEI-93-TR, Carnegie Mellon University, Software Engineering Institute, Feb. 1993.

[17] R. M. Podorozhny and L. J. Osterweil. The Criticality of Modeling Formalisms in Software Design Method Comparison,. Technical Report TR–96–049, University of Massachusetts, Computer Science Department, Amherst, MA, Aug. 1996.

[18] Potts C. (ed). Proc. of the softw. process worksh. In *IEEE cat. n. 84CH2044-6, Comp. Soc.*, Washington D. C., Feb. 1984. order n. 587, 27 – 35.

[19] Richard J. Mayer et al. IDEF family of methods for concurrent engineering and business re-engineering applications. Technical report, Knowledge Based Systems, Inc., 1992.

[20] X. Song and L. Osterweil. Toward Objective, Systematic Design-Method Comparisons. *IEEE Software*, pages 43 – 53, May 1992.

[21] X. Song and L. J. Osterweil. Experience with an approach to comparing software design methodologies. *IEEE Trans. on Software Engineering*, 20(5):364 – 384, May 1994.

[22] S. M. Sutton, Jr., D. Heimbigner, and L. J. Osterweil. APPL/A: A language for software-process programming. *ACM Trans. on Software Engineering and Methodology*, 4(3):221 – 286, July 1995.

[23] S. M. Sutton, Jr. and L. J. Osterweil. The design of a next-generation process language. Technical Report CMPSCI Technical Report 96–30, University of Massachusetts at Amherst, Computer Science Department, Amherst, Massachusetts 01003, May 1996. Revised January, 1997.