

CHALLENGING SOFTWARE PROCESS IMPROVEMENT BY DESIGN

Ivan Aaen

Department of Computer Science, Aalborg University
Phone: +45 9635 8911, Fax: +45 9815 9889
Email: ivan@cs.auc.dk

ABSTRACT

Software process improvement (SPI) today is based mainly on a perception of software processes as artifacts and this perception has led SPI efforts to focus on perfecting such artifacts as a means to improve the practices of the people supposed to execute these software processes. Such SPI efforts thus tend to view the design of software processes as separate from their use. In this approach process designers are expected to provide process knowledge to software developers, and software developers are expected to provide experiences and problems to the process designers. This focus on software processes as artifacts implies an emphasis on formalization and externalization of process models possibly at the expense of the process knowledge in the heads of the process users. The paper points to problems related to separation and externalization from a theoretical standpoint and suggests an alternative to Improvement by Design: End-user SPI, where process users individually and collectively design their own software processes assisted by process experts.

1. INTRODUCTION

Throughout its existence the software industry has sought for remedies against the missed schedules, blown budgets, and flawed products that haunt so many software projects. In the last decade much of this effort has concentrated on Software Process Improvement (SPI) based on the software capability maturity model (CMM) (Paulk, Curtis et al. 1993) or similar norms. The aim of SPI is to build an infrastructure and a culture that support effective methods, practices, and procedures so that they are the ongoing way of doing business. Experience has shown that such aims are difficult to obtain. SPI involves organizational change on a scale and with a complexity that requires commitment, resources, and skill at all organizational levels and also with a large risk of failure.

It is unknown how many companies start doing SPI and likewise we cannot know for sure how many companies abandon their SPI efforts without achieving the goals set for the effort. A recent report from the Software Engineering Institute (SEI) suggests that a substantial proportion of organizations initiating SPI run into stalemates or simply abandon the effort: By August 2001 a total of 1505 organizations had reported from assessments conducted since 1987 through June 2001. Only 379 of these 1505 organizations had been reassessed corresponding to 25 per cent (SEMA 2001). The time organizations spend to move up maturity levels is also considerable according to the same report. The median time to move from maturity level 1 to 2 is 24 months, 21.5 months from level 2 to 3, 33 months from level 3 to 4, and 18 months from level 4 to 5, which shows that progression through CMM maturity levels is time-consuming and difficult. About 90 per cent of the reporting organizations are at level 3 or less.

There is widespread agreement that organizations realize gains in development cycle time and productivity when improving their software processes (Diaz and Sligo 1997; Haley 1996; Hollenbach, Young et al. 1997; Humphrey, Snyder et al. 1991). One SEI study report on SPI programs in 48 organizations spending 1375 USD (median figure) per software engineer (Herbsleb, Zubrow et al. 1997). This study also indicate high ratios of benefits to cost, with figures in the range of 4:1 to almost 9:1. Unfortunately such reports on gains are generally based on anecdotal evidence (Ravichandran and Rai 2000).

Software development is a highly complex undertaking characterized by strong interdependencies among tasks, uncertainties about objectives and resource requirements, combined with limited similarity from one project to another. Such complexities can easily explain why SPI efforts use a lot of time and money with varying degrees of success. Few reports are available on factors related to SPI failure. Some experience reports point to "organizational politics", turf guarding, previous improvement failures, beliefs that SPI "gets in the way of real work", and lack of guidance on *how* to improve as being possible sources of SPI failures (Herbsleb, Zubrow et al. 1997).

The main contribution of this paper is to question the conventional way to do SPI by asking *what* a software process is, *where* it resides, *who* develops it, and *what* a software process design is. The paper is mainly theoretical and even speculative. Hard empirical evidence or extensive references to literature will not back most propositions in the paper. The propositions will however question what software process improvement is about and ask if there are better ways to obtain real improvements.

The paper starts out by presenting key ideas in software process improvement as an introduction to the field. The traditional approach to SPI - labeled *Improvement by Design* - is described and characterized with a focus on two aspects: Separation of process design from process use, and externalization of process knowledge. After this the traditional approach is challenged from two perspectives: Knowledge management and organizational design, and alternatives to Improvement by Design is outlined under a common label: *End-user SPI*.

2. KEY IDEAS IN SOFTWARE PROCESS IMPROVEMENT

SPI is about changing software organizations populated by technical and highly specialized people working with complex tasks using a variety of methods and tools. This complexity means that SPI also is a complex undertaking. SPI efforts may involve:

- A design where the architecture of the software process is developed and where the interfaces between process elements are defined.
- Establishing and realizing a vision for future technical and organizational arrangements in the software organization.
- Developing tools to support the software process.

Traditional SPI is plan-oriented in the sense that it is mainly concerned with producing a blueprint i.e. a description or model of some future desirable state of the software organization.

SPI people are mostly perceived as specialists interacting with managers and developers and SPI work is seen as the province of experts in cooperation with high-profile developers carefully selected as having valuable process knowledge. The SPI expert may act as process manager, process architect, process developer, consultant, mentor, teacher, researcher, reformer or perhaps even process therapist.

Based on a comprehensive study of the SPI literature and from experiences practising SPI in software organisations Aaen, Arent, Mathiassen, and Ngwenyama (Aaen, Arent et al. 2001a; Aaen, Arent et al. 2001b) have identified 3 fundamental concerns for SPI and for each of these concerns 3 key ideas informing SPI efforts. Together these concerns and ideas form a conceptual MAP of SPI. The three concerns are *Management (M)* - the principles that are used to manage the intervention; *Approach (A)*

- the approach taken to guide the intervention process; and *Perspective (P)* - the perspectives used to focus attention on the intervention target.

Management of an SPI intervention effort is based on three ideas. (1) Organizing those responsible for SPI efforts in dedicated projects. (2) Planning the effort with goals, activities, and responsibilities of the overall intervention as well as specific actions. (3) Using feedback through systematic measurements and assessments of the effects on software engineering practices.

The approach to SPI is guided by three additional ideas. (1) Evolutionary changes focusing on experiential learning and stepwise improvements. (2) Using norms - idealized, and a priori defined and stable models of software engineering. (3) Building on commitments between the involved actors in order to ensure dedication and legitimacy.

Finally, the perspective of the intervention process is also influenced by three ideas. (1) Using the software process that integrates people, management and technology as the main lever for improvement. (2) Building on software developer competencies as the key resource for the software process. (3) Changing the context of the software operation to establish sustainable support for the software process.

Not all of these ideas are used in every SPI effort, but several of the ideas are used in most efforts. Together these 9 ideas illustrate the scope and complexity of SPI efforts. To cope with this scope and complexity we need advice on how to do SPI in practice. Traditionally this advice has been to approach SPI by designing a desired software process followed by implementation activities, i.e. Improvement by Design.

3. IMPROVEMENT BY DESIGN

Watts Humphrey and colleagues developed the first ideas on SPI in the second half of the 1980s. These ideas became the foundation for the CMM. Historically CMM was developed at a time where one major line of software engineering research focussed on software factories and software infrastructures. A time where process modeling emerged as a field related to the development of third generation CASE tools - tools that would support method specification, adaptation and extension. This new generation of tools would allow for tailor-made and individual software processes given that such processes were defined in formal models.

This focus on formal models relates to a paper that appeared at the time Watts Humphrey developed the first version of the Maturity Model. That paper was Leon Osterweil's *Software processes are software too* (Osterweil 1987). A decade later this paper was named the most influential paper of ICSE 9. Leon Osterweil mentions that the earliest impetus for the ideas of process programming arose out of meetings with Watts Humphrey and his team at IBM in the early 1980s (Osterweil 1997). In his paper Osterweil suggests

that we describe software processes by "programming" them much as we "program" computer applications. We refer to the activity of expressing software process descriptions with the aid of programming techniques as process programming, and suggest that this activity ought to be at the center of what software engineering is all about.

There are good reasons for developing explicit process models. One reason is that uniform software processes across software projects support the use of experiences from earlier projects in new projects. Another reason is that it allows for supporting software development and maintenance with effective techniques and tools.

Feiler and Humphrey (Feiler and Humphrey 1991) state that the software process can be viewed in much the same way as software. Referring to Osterweil (Osterweil 1987) they argue that software processes have many of the same artifacts and require similar disciplines and methods. Feiler and

Humphrey suggest that we therefore think about the software process development life cycle in software development terms.

One consequence of adopting such a "programming" approach is the *separation* in time and space of the software process design from the use of the software process much like the classic linear sequential waterfall model or other well-known paradigms for software development. Although Feiler and Humphrey point to using evolutionary - prototyping-like - approaches to process development the bottom line in the programming approach is that the process design is separated from process use. The IDEAL model for process improvement (McFeeley 1996) is structurally almost identical to the waterfall approach with separate assessment, diagnosis, analysis and design of software processes followed by process rollout from process designers to process users as the way to change practice. This model is probably the most widely known user's guide to process improvement.

Another consequence of the programming approach to process improvement is a strong focus on formalization which in turn involves *externalization*, i.e. the representation of process knowledge outside the heads of the process users. The idea of formalization is pervasive in the CMM as can be seen in some of the key concepts of the CMM (Paulk, Weber et al. 1993). A *capability maturity model* is defined as a description of the stages through which software organizations evolve as they define, implement, measure, control, and improve their software processes. A *software process description* is the operational definition of a major software process component identified in the project's defined software process or the organization's standard software process. It documents, in a complete, precise, verifiable manner, the requirements, design, behavior, or other characteristics of a software process. Similarly *Software process maturity* is defined as the extent to which a specific process is explicitly defined, managed, measured, controlled, and effective. These concepts and several more signify the importance put on formalization and thereby externalization in the CMM.

Although explicit process models are needed in order to support software developers with methods and powerful tools this paper argues that improvement by design of explicit and externalized models may in fact impede the improvement effort and chances of success.

4. CHALLENGING IMPROVEMENT BY DESIGN

In the following we will identify some of the problems for SPI that may follow from separating process design from use and from externalization of process knowledge. We will do so with inspiration from two other research traditions: Knowledge management and organizational design.

4.1 SPI as knowledge management

One fundamental problem in SPI is to ensure a common understanding of the software process among those who are to follow it. In this respect SPI is about changing and building process knowledge in the individual and in the group and organization - in other words SPI is about knowledge management. Fahey and Prusak (Fahey and Prusak 1998) identify a number of errors of knowledge management relating to what organizations know and how organizations learn.

The 11 errors identified by Fahey and Prusak are listed in Table 1. These 11 errors basically lead us to address a number of ontological questions about SPI, e.g.:

- *What* is a software process? Is it a thing - an artifact? Is it a set of actions - something we do?
- *Where* is the software process - where can it be found? In documents? In products? In peoples minds?
- *Who* develops the software process? Process developers or process users?
- *When* are software processes developed? At distinct times? All the time - continuously?

In the following the errors will be described and related to the conventional approach to process improvement by design.

Table 1

Errors of Knowledge Management	
1.	Not Developing a Working Definition of Knowledge
2.	Emphasizing Knowledge Stock to the Detriment of Knowledge Flow
3.	Viewing Knowledge as Existing Predominantly Outside the Heads of Individuals
4.	Not Understanding that a Fundamental Intermediate Purpose of Managing Knowledge Is to Create Shared Context
5.	Paying Little Heed to the Role and Importance of Tacit Knowledge
6.	Disentangling Knowledge from Its Uses
7.	Downplaying Thinking and Reasoning
8.	Focusing on the Past and the Present and Not the Future
9.	Failing to Recognize the Importance of Experimentation
10.	Substituting Technological Contact for Human Interface
11.	Seeking to Develop Direct Measures of Knowledge

(Fahey and Prusak 1998)

Error #1 Not Developing a Working Definition of Knowledge. Process data, process information, and process knowledge and competence are not the same. As Dahlbom and Mathiassen (Dahlbom and Mathiassen 1993) put it *information is something we provide and receive, knowledge and competence are something we have*. Failing to distinguish between these concepts will lead to uncertainty regarding what to improve - and how - in SPI efforts. A focus on externalization in SPI implies a focus on data and information possibly at the expense of attention to the process of building knowledge and competence among process users.

Error #2 Emphasizing Knowledge Stock to the Detriment of Knowledge Flow. If process information is not distinguished from process knowledge SPI projects may tend to view process knowledge as an object that can be captured, stored, retrieved, and transmitted among individuals. Conversely a focus on knowledge flow between people may lead to a fundamentally different notion of process knowledge - a notion where process knowledge is in a constant flux and change, and a notion where individuals create it. Following Fahey and Prusak process knowledge *connects, binds, and involves individuals and is inseparable from the individuals who develop, transmit, and leverage it*. Both externalization of software processes and separation of process design from practice are logically related to this type of error. This error may result in gold-plating process models and failure to achieve an effective flow of process knowledge by settling for transmission and capture of process information.

Error #3 Viewing Knowledge as Existing Predominantly Outside the Heads of Individuals. Process knowledge is only meaningful in conjunction with "process knowers". If organizations view process knowledge as an object that has a life of its own they may tend to develop and maintain elaborate information structures with more and more complex process descriptions. This externalization is not a problem in itself for SPI efforts but it may turn SPI efforts into projects that are mainly done in the back-office thereby seriously distracting attention away from the practitioners who should be both the source and object of process knowledge.

Error #4 Not Understanding that a Fundamental Intermediate Purpose of Managing Knowledge Is to Create Shared Context. Given that practitioners ultimately are both source and object of process knowledge, organizations must build and maintain a shared context that explicates and aligns process knowledge among practitioners and even allows for questioning such shared contexts as part of gaining experiences. A shared context built from among other things an understanding of particular market conditions, customer relations, previous software projects, process information, and process norms is necessary in order to have a dialogue on process rationale and process improvement and indeed in order to have a shared practice. Separating process design from process practice will likely impede a reflective dialogue between users and designers.

Error #5 Paying Little Heed to the Role and Importance of Tacit Knowledge. If process knowledge resides in the heads of software practitioners the concept of tacit knowledge (Nonaka and Konno 1998; Polanyi 1962) takes on a central role in understanding, learning, and creating explicit process knowledge. It is via the use of embodied process knowledge that explicit process knowledge is captured, assimilated, created, and disseminated. SPI efforts that focus on externalization may experience a confrontation between tacit and explicit process knowledge by rewarding the use of explicit knowledge and limiting the use of tacit knowledge. The result may be counterproductive: practitioners may reject explicit knowledge if it is contrary to their tacit knowledge, and process designers may downplay or ignore tacit knowledge in their commitment to manage explicit process knowledge.

Error #6 Disentangling Knowledge from Its Uses. Disentangling process knowledge from its uses is seen when knowledge initiatives, projects, and programs become ends in themselves. Following Fahey and Prusak such a development is a result of a number of false assumptions: (1) Equating access to information with insight, value, or utility; (2) assuming that the value of data and information is evident; and (3) believing that process knowledge "users" and "producers" can be segregated. Separating process users from process designers is the traditional way to do SPI and many SPI efforts are done mainly in methods or quality functions and many have as their main deliverable publication of massive amounts of process documentation on the corporate intranet. In many such efforts process rollout is done via training and incentive schemes encouraging process users to use these SPI deliverables in their daily practice.

Error #7 Downplaying Thinking and Reasoning. Software organizations as any other type of organization are always undergoing changes and software projects are always different. For these and other reasons software processes develop over time adapting to new situations. For Fahey and Prusak such development of knowledge takes place through thinking and reasoning. Both tacit and explicit knowledge tends to coagulate and fossilize into ideology unless challenged by discussions among individuals and groups. Producing explanations as to why and how a particular task is done or arguing the relevance of a software paradigm to a new project or customer is one important way to keep software processes current. When process design is separated from process use the result may be that such explanations and disputes are downplayed leading to process ossification.

Error #8 Focusing on the Past and the Present and Not the Future. Explicit consideration of the future is rarely a driving force in the design of software processes. By nature a focus on best practices and examples on process use will be retrospective and tend to be selected with a view to the present and one concern of software organizations would be the demands for the future software process. Given that the future and consequently also the demands for the future software processes are ambiguous, software organizations need alternative projections in order to discuss options. Separating process design from process use will make it more difficult to obtain such diverse projections, as process designers will favor unambiguous and clear recommendations to management and process users.

Error #9 Failing to Recognize the Importance of Experimentation. Improving and adapting software processes to better suit given conditions require experimentation. In the IDEAL model pilots experimenting with process changes is an integral part of the improvement effort. A pilot is defined as an *initial implementation of an improvement, usually on a small, controlled scale, before general*

installation (McFeeley 1996). One implication of this approach is that experiments are part of the process design but separated from regular process use. Best practices are often used to encapsulate applied process knowledge and make it available to process users, but such practices may contribute to ossify process knowledge and tend to suppress innovations and improvements gained by process users deviating from existing best practices. In general terms any software project is an experiment, but the separation of process design from process use and the aspiration to achieve statistical process control as part of SPI may pose major obstacles to gain knowledge from these everyday experiments.

Error #10 Substituting Technological Contact for Human Interface. Software process descriptions, best practices, process templates, and process data are made available on corporate intranets for good reasons. Unfortunately this may mislead software organizations to confuse access to information and data with access to organizational knowledge, i.e. to think that technology can replace face-to-face dialogue. Externalization of process knowledge favors the building of repositories with process information and data, and separation of design from use tends to downplay the importance of dialogue for communication, interaction, exchange, and learning of process knowledge.

Error #11 Seeking to Develop Direct Measures of Knowledge. How can organizations know if their SPI efforts have the desired effect? Senior managers have an understandable urge to know the payoff of SPI investments and are therefore tempted to focus on ways to measure SPI results. Such metrics could include size of process descriptions, number of intranet hits, hours spent on SPI, number of SPI initiatives in the corporation, amount of process data collected, etc. A focus on externalization of process knowledge makes such metrics meaningful but basically this kind of indicators tells little about the stock or flow of process knowledge or the effect of this knowledge on process performance. As Fahey and Prusak state this view on direct measures of knowledge *put the measurement cart before the knowledge horse*. It reinforces many of the above errors, it maintains a view of knowledge as being outside the heads of people, and it consigns process users to a secondary role in SPI.

4.2 SPI as organizational design

SPI is about changing the ways things are done in organizations that develop and maintain software - a change to be brought about via designed software processes. In this respect we can also see SPI as an effort to do organizational design.

Karl Weick contrasts two perceptions on organizational design (Weick 1993). The first and more traditional idea is to see organizational design as architecture focussing on structures, whilst the alternative idea is to see design as improvisation focussing on dynamics. Transformed to SPI concepts the implied assumptions of these two ideas are shown in Table 2.

Table 2

Software process as architecture	Software process as improvisation
A software process is a blueprint	A software process is a recipe
A software process is constructed at a single point in time	A software process is continually reconstructed
Software processes produce order through intention	Software processes produce order through attention
A software process creates planned change	A software process codifies unplanned change after the fact

Based on (Weick 1993)

Karl Weick has a view on organizational design that is quite different from the worldview on which traditional SPI and software engineering is built. Transposing his view to SPI a software process is what process users believe it to be. What they believe it to be reflects what the software process was. Perceptions of what it was are the basis for what process users do. What people do *is* the software process. Organizational concepts such as interpretation, negotiation, emergence, enactment, etc. are part of Weick's perspective.

As Weick notes the word design is both a noun and a verb, and the same is true for the word process. The traditional perspective on software process tends to focus on tangibles, on things like written procedures, data, job descriptions, etc. Features that are less tangible - less thing-like - and more continuous, fleeting, and emergent, are easily overlooked. Perhaps this is why SPI tends to focus on *structure* rather than *practice* and - as Weick notes for organizational design - to *contain few provisions for self-correction*.

To Weick a blueprint is a description of features that in a precise and complete fashion defines a phenomenon, while a recipe have an open-ended quality allowing for improvisation by the practitioner.

When we see a software process as a blueprint we focus on structure and on the objects building this structure. A blueprint view is helpful in providing concepts and patterns can be used for recognizing and communicating how we understand software development processes. A blueprint view is however not helpful for understanding how any such structure emerges. A recipe can provide such an understanding. A recipe describes actions that generate the objects - e.g. established software practices - to be specified in a blueprint.

The traditional separation of process design from process use in SPI reflects an architectural perspective on SPI. From this perspective the software process is a blueprint description of the software process to be implemented and a rollout plan is a recipe describing how to get there. In other words: The blueprint is the goal and the corresponding recipes provide the means to achieve it.

A perspective of improvisation on SPI would reject relegating recipes to a secondary role in SPI. Practitioners who improvise would treat software processes not as givens but as emergents. The givens for such practitioners are the recipes and routines that generate the actions to create a future blueprint - a software process - based on established practice. Recipes do not specify in advance the structures that will emerge as the recipes are used. As Weick puts it *events are set in motion, but the orderliness they will create remains to be discovered*.

From the perspective of improvisation a software process is emergent, evolutionary, difficult to control, tied to action, and continuously updated as people and conditions change. The blueprint gives meaning to past activities explaining why actions happen and what to pay attention to. Designs thus are formed in retrospect and are used to influence attention of practitioners. This perspective sees designing as a mixture of a shifting pattern of attention and meaning imposed on an ongoing stream of activities in the software organization rather than a stable pattern of intentions regarding a future practice.

This line of reasoning inspired by Weick further challenges some of the ideas underlining Improvement by Design:

Striving for designing the software process as a blueprint. Normally processes are used as recipes - the practitioner treats processes as emergents. This being the case the question is why would we try to develop a standard process in the back-office, rather than promoting what practitioners actually do to a current standard? Furthermore: In practical software development it is normal procedure to tailor a standard process to the particular needs of a project. Why then would we restrain improvisation by insisting that this tailored software process be designed as a blueprint for the particular project instead of a recipe that will enable self-correction in the likely event of something unexpected?

Constructing the software process at a single point in time. Every new project, new technology, new customer or new market, and every new grouping of people in a project provides new challenges and opportunities, and so does the learning processes among the process users. They learn from experience and this learning provides input to reconsider the software processes. These sources of change imply that we should accept continual process reconstruction. The IDEAL model (McFeeley 1996) is in fact iterative, but when we try to improve the software process iteration by iteration as suggested by the model, each of these iterations will normally endeavor to produce a complete sub-process. Were we to build a house this approach would correspond to completing one room at a time. Iterating this way codifies sub-processes one by one and impedes later reconstruction based on experience.

Assuming that the software process produce order through intention. Improvement by Design assumes that the blueprint of a process will convey to the process user the intentions embedded in the process, and lead the user to adhere to these intentions as a foundation for the work. In reality blueprints are tools of attention based on past experience by making past actions more sensible. Focussing attention via recipes expressing values and pointing to concerns may have a stronger impact on practice than using directives in the form of elaborate process descriptions. Successful process design implies an alignment of values that make people care about meeting the standards they and their colleagues deem part of their professional repertoire.

Believing that designing a software process creates planned change. Process descriptions may be after the fact. A process design may reflect how something is usually done more than what is thought to be the best way to do something. We should consider alternatives to the traditional sequence of designing a change followed by efforts to rollout the new design. Adopting a perspective of improvisation would suggest that we change practice first and then change the process model according to established practice.

Following Weick we thus arrive at a number of alternative ideas to SPI: (1) that we see process modeling and SPI as ongoing activities, (2) that responsibility for process redesign is dispersed, and (3) software processes should be simple and open to interpretation

The next chapter will propose alternatives to Improvement by Design allowing for improvisations and putting the process user - the practitioner - at the center of SPI. For this reason we will label these alternatives *End-user SPI*.

5. AN ALTERNATIVE TO IMPROVEMENT BY DESIGN: END-USER SPI

Martha S. Feldman (Feldman 2000) describes how organizational routines act as a source of continuous change. Contrary to traditional beliefs that routines are unchanging she claims that such routines have a great potential for change as participants respond to outcomes of previous iterations of a routine. This dynamic is based on the inclusion of routine participants as agents of change. By analyzing how routines change as a result of participants' reflections on and reactions to various outcomes of them Feldman points to the significance of not separating the people who are doing the routines from the routines.

To Feldman a routine develops following a loop involving plans, actions, outcomes, and ideals. Interactions between the elements of this loop support actions of repairing, expanding and striving that change routines and hopefully keep routines aligned with ongoing demands.

People in a position to influence an organizational routine may have various ideas about what should be accomplished by the routine, they may have many interpretations of the effectiveness of the actions taken, and they may have many interpretations of whether the outcome of the actions is a problem or a resource and, in either case, what to do about it (Feldman 2000).

Such deliberations are similar to ideas developed by researchers on learning such as Argyris, Schon, and Nonaka all of whom are well known to the software engineering community. The idea here is to suggest that we learn from these and similar writers in developing alternatives to Improvement by

Design. Common to this line of research is a focus on improving and changing via learning and via focussing on the reflective practitioner. It is beyond the scope of this paper to describe in any detail what such an approach to SPI with the process user as the process designer might look like. However the following may provide some clues:

Use approaches aimed at establishing software processes at the group level. Watts Humphrey's Team Software Process (Humphrey 2000) is one prominent example of SPI at the group level. Kent Beck's eXtreme Programming (Beck 1999) is an example of a lightweight approach targeted at the same level but coming from outside the traditional software engineering field. In both approaches the developers play a key role in developing the software process and the process is stabilized through group level discipline as well as modified through group level interactions.

Use process specialists as coaches or mentors. If software processes are to be developed for, with, and by process users they may need coaching and mentoring from process specialists sitting with team members solving process problems as they come forth. Such process specialists will likely also be instrumental in the diffusion of process ideas across the organization.

See the process as a recipe rather than a blueprint. The ability and latitude to improvise under common standards is critical to professional work. Software processes are emergent and open to improvisation respecting process goals established at the level of the group and the level of the organization.

Improve first, then model the outcome - codify the software process after the fact. Model what process users do while they do it. Integrate SPI efforts into ongoing software projects. Define what IS done, not what should be done in the future. Define the daily practice.

A little structure goes a long way, as Weick puts it (Weick 1993). Rather than building ever more elaborate process models and other kinds of process information we might want to consider the use of simple and flexible process models by highly motivated and competent people. Define essential process elements rather than all process elements. Focus on key objectives and deliverables rather than standard procedures.

Use embedded and continual process assessments at the project level. Feedback to process users is essential to give insight into strengths, weaknesses and developments in the software processes. A number of light assessment methods are available to provide insight into norm conformance at the project level. Such methods can inform an End-user SPI effort by assessing the process in use directly (Daskalantonakis 1994; Wiegers and Sturzenberger 2000) or indirectly by evaluating process artifacts (Aaen, Bøttcher et al. 1996).

How do such propositions differ from conventional wisdom? Commonsense as they may seem their main contribution to practice may be the vision upon which they are based: (1) that process development and use be integrated as the responsibility of the process users, and (2) that software processes be recognized as knowledge and competence rather than as artifacts.

6. CONCLUSION

This paper set out to challenge traditional approaches to software process Improvement by Design. It was suggested that separation of process design from process use together with an overly focus on process externalization could be one reason why SPI efforts are very costly in money and time and appears to have a high failure rate. Based on this End-user SPI was outlined as an alternative to SPI by design.

Far from a simple and straightforward process SPI is a problematic activity with considerable implications for practitioners' beliefs, values, emotions, perceptions, and behaviors. End-user SPI suggests that SPI should not be approached in a linear step-by-step fashion by which software process goals are translated into solutions which are then transferred to practice, but is better approached by

putting competent process users at the center of process design as part of their ongoing activities. End-user SPI calls for simple and open process models - models suited for improvisation.

Process designers tend to be obsessed with control. We seem to believe that software processes should be controlled much like software is controlled. This belief has little merit in our field, and it is time to let go of it. Instead we should focus on commitment, competence and responsibility as a way to achieve profound process improvement.

Letting go of the idea of Improvement by Design may give room for worries that we end up with lax and ad hoc software processes with little room for learning and systematic improvement: will we also have to let go of the idea of discipline and structure in software development? Extreme Programming (XP) may be one example that End-user SPI is indeed possible without sacrificing discipline. In a recent paper Mark Paulk - one of the key authors behind the CMM - describes XP as a highly disciplined process with *good engineering practices that can work well with the CMM and other highly structured methods* (Paulk 2001). Paulk argues that XP and CMM are complementary although several key process areas of the CMM are not covered in Extreme Programming. Paulk further argues that *organizations that want to improve their capability should take advantage of the good ideas in both, and exercise common sense in selecting and implementing those ideas* (ibid.). As the CMM seeks to standardize at the level of goals for key process areas this model allows for great discretion about how to achieve these goals. End-User SPI suggests that this discretion be administered at the level of the process users.

This paper marks only a first beginning of the development of End-user SPI. Further research is needed on topics like how to build processes for improvisation, how to exchange process knowledge in large organizations, how to meet certification requirements, how to adhere to norms - e.g. CMM - with End-user SPI, etc.

Perhaps a quote of Mahatma Gandhi can guide the development of End-user SPI: *I have not conceived my mission to be that of a knight-errant wandering everywhere to deliver people from difficult situations. My humble occupation has been to show people how they can solve their own difficulties.*

ACKNOWLEDGMENT

The author is indebted to the anonymous reviewers for valuable comments and suggestions and also wishes to thank the software organisations, practitioners, and researchers participating in the Danish SPI initiative for stimulating the ideas in this paper in numerous ways.

REFERENCES

- Aaen, I., J. Arent, L. Mathiassen, and O. Ngwenyama. (2001a) A Conceptual MAP of Software Process Improvement. *Scandinavian Journal of Information Systems*, 13, 123-146.
- Aaen, I., J. Arent, L. Mathiassen, and O. Ngwenyama. (2001b) "Mapping SPI Ideas and Practices." In Lars Mathiassen, Jan Pries-Heje, and Ojelanki Ngwenyama eds., *Improving Software Organizations*, 23-46. Addison-Wesley, Boston, MA.
- Aaen, I., P. Bøttcher, and L. Mathiassen. (1996) "Delta Measurement in Software Process Improvement based on Product Attributes." Proceedings of the 4th European Conference on Information Systems, Lisbon, July 2-4, 15-28.
- Beck, K. (1999) Embracing Change with Extreme Programming. *Computer*, 32 (10), 70-77.
- Dahlbom, B., and L. Mathiassen. (1993) *Computers in Context - The Philosophy and Practice of Systems Design*. NCC Blackwell, Cambridge, MA.
- Daskalantonakis, M. K. (1994) Achieving Higher SEI Levels. *IEEE Software*, 11 (4), 17-24.
- Diaz, M., and J. Sligo. (1997) How Software Process Improvement Helped Motorola. *IEEE Software*, 14 (5), 75-81.

- Fahey, L., and L. Prusak. (1998) The eleven deadliest sins of knowledge management. *California Management Review*, 40 (3), 265-276.
- Feiler, P. H., and W. S. Humphrey. (1991) *Software Process Development and Enactment: Concepts and Definitions*. Pittsburgh, PA: Software Engineering Institute, CMU/SEI-92-TR-004.
- Feldman, M. S. (2000) Organizational routines as a source of continuous change. *Organization Science*, 11 (6), 611-629.
- Haley, T. J. (1996) Software process improvement at Raytheon. *IEEE Software*, 13 (6), 33-41.
- Herbsleb, J., D. Zubrow, D. Goldenson, W. Hayes, and M. Paulk. (1997) Software Quality and the Capability Maturity Model. *Communications of the ACM*, 40 (6), 30-40.
- Hollenbach, C., R. Young, A. Pflugrad, and D. Smith. (1997) Combining Quality and Software Improvement. *Communications of the ACM*, 40 (6), 41-45.
- Humphrey, W. S. (2000) *Introduction to the Team Software Process* The SEI Series in Software Engineering. Addison-Wesley, Reading, Massachusetts.
- Humphrey, W. S., T. R. Snyder, and R. R. Willis. (1991) Software Process Improvement at Hughes Aircraft. *IEEE Software*, 8 (4), 11-23.
- McFeeley, B. (1996) *IDEAL: A User's Guide for Software Process Improvement*. Pittsburgh: SEI. Handbook, CMU/SEI-96-HB-001.
- Nonaka, I., and N. Konno. (1998) The Concept of "Ba": Building a foundation for Knowledge Creation. *California Management Review*, 40 (3), 40-55.
- Osterweil, L. J. (1987) "Software Processes are Software Too." Proceedings of the Ninth International Conference on Software Engineering, Monterey, CA, March, 2-13.
- Osterweil, L. J. (1997) "Software Processes are Software Too, Revisited." Proceedings of the 19th International Conference on Software Engineering, May 1997, 540-548.
- Paulk, M. C. (2001) Extreme Programming from a CMM Perspective. *IEEE Software*, 18 (6), 19-26.
- Paulk, M. C., B. Curtis, M. B. Chrissis, and C. V. Weber. (1993) *Capability Maturity Model for Software, Version 1.1*. Software Engineering Institute. Tech. report, CMU/SEI-93-TR-24.
- Paulk, M. C., C. V. Weber, S. M. Garcia, M. Chrissis, and M. Bush. (1993) *Key Practices of the Capability Maturity Model, Version 1.1*. Software Engineering Institute. Tech. report, CMU/SEI-93-TR-25.
- Polanyi, M. (1962) *Personal Knowledge*. Anchor Day Books, New York.
- Ravichandran, T., and A. Rai. (2000) Quality Management in Systems Development: An Organizational System Perspective. *MIS Quarterly*, 24 (3), 381-416.
- SEMA. (2001) *Process Maturity Profile of the Software Community 2001 Mid-Year Update*. Presentation. Pittsburgh, PA: Software Engineering Institute.
- Weick, K. E. (1993) "Organizational redesign as improvisation." In G.P. Huber and W.H. Glick eds., *Organizational Change and Redesign*, 346-379. Oxford University Press, New York.
- Wieggers, K. E., and D. C. Sturzenberger. (2000) A Modular Software Process Mini-Assessment Method. *Ieee software*, 17 (1), 62.