Software Process Improvement: Blueprints versus Recipes

Ivan Aaen, Aalborg University

Over the past decade or so, organizations have spent vast resources on software process improvement. Based on the software capability maturity model (CMM) and similar norms, SPI aims to build an infrastructure and culture that support effective methods, practices, and procedures and integrate into the ongoing way of doing business. This goal has proven difficult to reach, however, because SPI involves organizational change on a scale and of a complexity that requires commitment, resources, and skills at all organizational levels, and it carries a large risk of failure.

This article analyzes the roots of these difficulties and challenges the traditional SPI approach, here labeled Blueprint SPI. Plan-oriented and mainly concerned with the static, this method creates a blueprint of a future software process. Its three primary aspects—externalizing process knowledge, separating process design from use, and structuration by melding process parts into organized wholes—focus on structure and the objects building it. Charts, functional elements, and the entities such elements produce or exchange typically describe a process. A blueprint view provides concepts and patterns for recognizing and communicating how we understand software processes. Process designers are expected to provide process knowledge to software developers, and software developers are expected to provide experiences and problems to process designers.

A blueprint view does not, however, reveal how such structures emerge. Looking at SPI in the light of knowledge management and organizational design concepts helps us see that a recipe can better elucidate structural dynamics. Recipes for software practices describe actions that generate the objects—such as established software practices—that blueprints specify. Recipes model emergence and form a basis for improvisation to suit a given situa-
Recipes are open-ended in that the resulting blueprint—the new software process—emerges only after the fact. Where blueprints define preplanned actions, recipes express values, principles, and goals for such actions. Understanding these gives us a better view of an organization's software development culture and thus a better chance of understanding how it may change.

**Blueprint SPI**

SPI targets software organizations in which highly specialized people execute complex tasks using various methods and tools. SPI experts typically interact with managers and high-profile developers carefully selected for their valuable process knowledge.

Watts Humphrey and his colleagues began developing SPI in the 1980s, and these early ideas became the foundation for the CMM. At the time, one major line of software engineering research focused on software factories and infrastructures. Process modeling emerged to assist the development of third-generation CASE tools that would support method specification, adaptation, and extension. These new tools would enable tailor-made software processes defined in formal models.

The emphasis on formal models traces back to a highly influential 1987 paper Leon Osterweil published around the time Watts Humphrey developed the first CMM version. The earliest impetus for process programming arose from Leon Osterweil’s meetings with Watts Humphrey and his team at IBM in the early 1980s. 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.

Peter Feiler and Watts Humphrey adopt a similar view, arguing that software processes have many artifacts in common and require similar disciplines and methods. They suggest we therefore think about SPI in software development terms.

Such an approach, however, separates software process design from use in time and space, much as the classic linear sequential waterfall paradigm does. Although Feiler and Humphrey suggest using evolutionary (prototyping-like) approaches to process development, the programming approach ultimately disconnects process design from process use. The IDEAL process improvement model, for example, specifies discrete steps for software process assessment, diagnosis, analysis, and design followed by process rollout to process users. This model, which structurally resembles the waterfall approach, is probably the most widely known user’s guide to process improvement.

The programming approach also externalizes process knowledge by representing and formalizing it outside process users’ heads. Some key CMM concepts reveal how formalization pervades the CMM. For example, a capability maturity model describes the stages software organizations go through as they define, implement, measure, control, and improve their software processes. Software process maturity characterizes how explicitly a specific process is defined, managed, measured, controlled, and effective. Similarly, a software process description operationally defines a major software process component identified in the project’s defined process or the organization’s standard process. It completely, precisely, and verifiably documents a software process’ requirements, design, behavior, or other characteristics. Describing the software process in such terms epitomizes structuration—defining process parts to form coherent systems.

Although we need explicit models of software processes or process parts to support software developers with methods and tools, I argue against using such models as the primary means to achieve SPI ends. Externalization, separation, and structuration might be useful for mapping people’s present behavior, but they’re dubious strategies for changing it.

**Challenging Blueprint SPI**

Concepts from knowledge management and organizational design research illuminate some of the problems that can follow from externalization, separation, and structuration. One fundamental problem in SPI is ensuring that software process users share a common understanding of the process. SPI thus entails changing and building process knowledge in the individual as well as in the group and organization. In this sense, it is about knowledge
management. Liam Fahey and Laurence Prusak identified 11 knowledge management errors relating to what organizations know and how they learn. This inspired the 11 corresponding Blueprint SPI errors listed in Table 1, which I will discuss in detail.

**Externalization**

When SPI efforts focus too much on documentation and too little on proficiency, organizations begin to care more about description and prescription and less about competence and behavior. Externalization confuses information publication with knowledge building, sees process models as ends rather than means, and disregards the importance of tacit knowledge.

**Confusing information with knowledge.** Process data and information aren’t the same as process knowledge and competence. As Bo Dahlbom and Lars Mathiassen put it, “information is something we provide and receive, knowledge and competence are something we have.” Failure to distinguish between these concepts will lead to uncertainty regarding what to improve in SPI efforts and how.

**Reifying process knowledge.** Process knowledge is only meaningful in conjunction with “process knowers.” Organizations that view process knowledge as an object in itself might start to develop and maintain elaborate information structures with ever more complex process descriptions and only limited relation to practice. This could divert attention from the practitioners who should be both the sources and objects of process knowledge.

**Paying little heed to the role and importance of tacit knowledge.** Externalization might cause a confrontation between tacit and explicit process knowledge by promoting explicit knowledge and limiting the use of tacit knowledge. In turn, practitioners might reject explicit knowledge if it contradicts their tacit knowledge, and process designers might downplay or ignore tacit knowledge.

**Substituting technological contact for human interface.** Software organizations make process descriptions, best practices, process templates, and process data available on corporate intranets for good reasons but sometimes confuse access to information and data with access to organizational knowledge through face-to-face dialogue. Externalizing process knowledge favors building repositories with process information and data and downplays the importance of dialogue for process knowledge communication, interaction, exchange, and learning.

**Seeking to develop direct knowledge measures.** How can organizations know if their SPI efforts have had the desired effect? Understandably, senior managers want to quantify SPI payoffs and are tempted to focus on ways to measure results. Such metrics could include process description size, number of intranet hits, hours spent on SPI, number of SPI initiatives in the corporation, amount of data collected, and so on. Externalization makes such metrics meaningful, but they say little about the stock or flow of process knowledge and its effect on performance. This focus on direct measures of knowledge puts the measurement cart before the knowledge horse, according to Liam Fahey and Laurence Prusak. It reinforces many other errors, maintains a view of knowledge as being outside people’s heads, and consigns process users to a secondary role.

**Separation**

When SPI efforts become too much the responsibility of process specialists and too little

---

**Table 1**

<table>
<thead>
<tr>
<th>Externalization of process knowledge</th>
<th>Separation of process knowledge from use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Confusing information with knowledge</td>
<td>Gold-plating process models at the expense of knowledge flow</td>
</tr>
<tr>
<td>Reifying process knowledge</td>
<td>Lack of reflective dialogue between process users and designers</td>
</tr>
<tr>
<td>Paying little heed to the role and importance of tacit knowledge</td>
<td>Process rollout by distributing process information</td>
</tr>
<tr>
<td>Substituting technological contact for human interface</td>
<td>Process ossification—downplaying thinking and reasoning</td>
</tr>
<tr>
<td>Seeking to develop direct (perhaps misguided) knowledge measures</td>
<td>Focusing on past and present, not the future</td>
</tr>
<tr>
<td></td>
<td>Standard processes leaving little room for experimentation</td>
</tr>
</tbody>
</table>
the responsibility of process users, defined software processes take on a life of their own, grow out of proportion, and have little interaction with actual software projects. When users implement defined process parts, they might do so as a tradition or mindless ritual.

**Gold-plating process models at the expense of knowledge flow.** If SPI efforts don’t distinguish process information from process knowledge, organizations might view process knowledge as an object to be captured, stored, retrieved, and transmitted. Process knowledge “connects, binds, and involves individuals” and “is inseparable from the individuals who develop, transmit, and leverage it.”7 Trying to separate this knowledge could result in gold-plating process models and failure to achieve an effective process knowledge flow by settling for process information transmission and capture.

**Lack of reflective dialogue between process users and designers.** Given that practitioners are both the source and object of process knowledge, organizations must build and maintain a shared context. This context should explicate and align process knowledge among practitioners and permit questioning such shared contexts as part of experience building.

**Process rollout by distributing process information.** Disentangling process knowledge from use occurs when knowledge initiatives, projects, and programs become ends in themselves. Traditional SPI separates process users from designers, focuses on methods or quality functions, and primarily publishes massive amounts of process documentation on the corporate intranet. Many such efforts roll out new processes through training and incentive schemes that encourage process users to incorporate these SPI deliverables in their daily practice, and such strategies often fail.

**Process ossification—downplaying thinking and reasoning.** Software projects vary widely. For this and other reasons, software processes develop over time through thinking and reasoning. Both tacit and explicit knowledge tend to coagulate into ideology unless challenged by discussions. Producing explanations about why and how to do a particular task or arguing a software paradigm’s relevance to a new project or customer helps keep software processes current. Separating process design from use downplays such explanations and disputes, leading to process ossification.

**Focusing on the past and present, not the future.** Explicit consideration of the future rarely drives software process design. By nature, a focus on best practices and process use examples will be retrospective and selected with present needs in mind. Given that the future and, consequently, future software process demands are ambiguous, software organizations need alternative projections to consider options. Separating process design from process use will make it more difficult to obtain such diverse projections because process designers will favor unambiguous and clear recommendations.

**Standard processes leaving little room for experimentation.** Improving and adapting software processes to better suit given conditions requires experimentation. The IDEAL model integrates pilots—initial implementations of improvements, usually on a small, controlled scale5—into the improvement effort. One implication of this approach is that experiments become part of the process design but separate from regular process use.

**Structuration**

SPI efforts seek to change how people do things through software process design. In this respect, we can also view SPI as organizational design.

Karl Weick describes traditional organizational design in terms of architecture focusing on structures.9 Applied to SPI, this approach outlines the assumptions on which structuration is based:

- A software process is constructed at a single point in time.
- Software processes produce order through intention.
- A software process creates planned change.

The software process is designed at a fixed point in time. The design activity is largely about defining a set of process structures. These structures should meet the software organization’s current needs, which are expected to change only incrementally.

Structuration thus reflects a conservative, rational view, where project conditions are rel-
atively stable and new projects tend to resemble previous projects. In this view, process designers decide on intentions and produce process models that incorporate these intentions. Process users base their projects on these models and thereby adhere to the intentions. Changing the software process in an organization essentially is a matter of process rollout.

Problems arising from structuration stem from systematic planning for eventualities and contingencies—for the expected rather than unexpected. Given many software projects’ complexities and uncertainties, structuration and process standardization might not always be the best answer. Rather than taking structuration and process standardization for granted as SPI goals, we should seek better ways to improve organizations’ ability to manage the unexpected.

**Recipes for SPI**

As an alternative to Blueprint SPI, we can conceive of SPI in terms of a recipe—guidelines that we can tailor to specific conditions and that let us account for SPI’s improvisational nature. Using a recipe, process users collectively design the software processes through facilitation, reflection, and improvisation.

**Facilitation**

The errors that stem from externalization raise two questions about SPI:

- What is a software process? Is it a thing, an artifact? Is it a set of actions, something we do?
- Where do we find the software process—in documents, products, or people’s heads?

If we view a software process as something we do and as residing in people’s heads, we see a new purpose for gathering and publishing software process information: *facilitation* of knowledge work. Moving from externalization to facilitation shifts our focus from control and standardization to supporting and enabling process users in deciding what a specific situation requires.

Facilitation entails providing access to *process information*, *tools*, and *organizational knowledge*. First, process specialists support knowledge building and stimulate imagination by gathering and providing information on methods, procedures, and experiences to users. Such specialists can also provide process information covering various situations and alternatives. Process users can mine for options relevant to a given situation and for information on the options’ strengths and weaknesses.

Second, access to powerful tools can greatly enhance a software development team’s capabilities by supporting and perhaps guiding task execution. For example, modern integrated development environments let development teams develop joint working standards for activities such as architectural design, refactoring, coding styles, and version control over time. Because it’s complex and modular, an IDE can support these activities effectively and flexibly.

Finally, access to people with relevant domain, technical, and process knowledge plays an obvious role for staffing development teams. If knowledge resides in software practitioners’ heads, tacit knowledge plays a central role in sharing and creating knowledge. Embodied process knowledge facilitates the capture, assimilation, creation, and dissemination of explicit process knowledge, which the organization can support through such methods as pair programming, collective code ownership, and team building. Process specialists can facilitate sharing by coaching and mentoring the process users and helping them solve problems as they arise. The specialists can also help with process idea diffusion across the organization.

**Reflection**

Errors that stem from separation raise several questions about SPI:

- Who designs the software process? Process developers or process users?
- How are software processes designed? Through decision-making or interaction?

The errors indicate that we should avoid separating process design from process use and instead focus on how to make process improvement an everyday concern of process users. If we accept this idea of software processes being improved by individuals and groups as they go about developing software, we arrive at SPI strategies focusing on everyday practices and professional reflection. Such strategies should seek to establish infrastructures that support learning and reflection.

Etienne Wenger identified three infra-
structures of a learning architecture that provide essential support for individual and collective reflection: engagement, imagination, and alignment. These infrastructures establish a shared context among process users that helps them understand particular market conditions, customer relations, previous software projects, process information, and process norms. A shared context forms the basis for shared practices and reflective dialogue on process rationale and improvement.

Engagement refers to the flow of knowledge between people, which provides a context for learning. An infrastructure of engagement should support continuity, mutuality, and competence. Continuity comes in the form of memories stored as information, embedded in tools, or accumulated in human recollection. Mutuality occurs in most software projects as project members engage and respond to other members’ actions. Competence evolves during any software project as it encounters problems that engage energy, creativity, and inventiveness. The occasions that arise during a software project for exercising judgment, mutual evaluation, and negotiating joint enterprises prove valuable for learning. Activities such as project postmortems and reviews exemplify such occasions.

Imagination involves creating new images of the software process and seeing connections through time and space by extrapolating from experience. Facilities that support imagination include knowing software processes and patterns, making comparisons with other practices, having opportunities and tools for trying things out, performing simulations, developing alternative projections, and so on. A project kick-off meeting where team members establish a common working framework for a project can serve as an ideal occasion for stimulating and employing imagination.

Alignment involves coordinating energy and activities to fit within broader structures and contribute to broader enterprises. An infrastructure of alignment should include facilities of jurisdiction, coordination, and convergence. Jurisdiction covers policies, contracts, and the like, and coordination encompasses standards, methods, communication, and more. Convergence covers shared understanding, creed, values, and principles. The goals of a CMM key process area or the fundamental values in and principles of software engineering paradigms such as object orientation or agile processes typify sources of alignment.

Improvisation involves selecting and transforming process elements, not pulling ideas out of thin air. Structuration builds on a conservative view characterized by stability, similarity, expectation, and planning. This view compels several ontological questions, however:

- When do people design software processes? At distinct times or continuously?
- Why do people design software processes in the first place? To ensure projects always get done the same way or to help them manage the unexpected?

If we accept that software projects are basically unstable and dissimilar, perhaps we should shift our expectations from having process users follow a given software process to inventing the process as they go—in other words, to move from a paradigm of planning to one of improvisation. In so doing, we abandon the idea of controlling projects through elaborate planning and embrace the unexpected as a source of inspiration and experience.

Improvisation involves selecting and transforming process elements, not pulling ideas out of thin air. It builds on elements drawn from the three areas of facilitation—process information, tools, and organizational knowledge. Organizations nurture improvisation through the SPI strategy of reflection, encouraging engagement, imagination, and alignment. Facilitation emphasizes process use, reflection emphasizes process learning, and both help create supportive infrastructures. Improvisation depends on everyday interaction, where process users test and reconstruct such infrastructures to keep them current and evolve competency to manage the unexpected.

The ability and latitude to improvise under common standards is critical to professional work. Cooperation among group members occurs not so much because they share a common defined process but because they share a view of what is happening and what it means. In practice, software processes are emergent and open to improvisation that respects shared process goals.

Following Weick,9 we can describe three characteristics of improvisation.

Continuous reconstruction involves people rebuilding software processes over time. Every
Agile processes such as Extreme Programming show that we can delegate SPI to process users without sacrificing discipline.

new project, technology, customer, market, and grouping of people on a project provides new challenges and opportunities. Process users learn from such experiences, and this learning provides input to reconsider the software process. So why would we try to develop a standard process in the back office rather than promoting what practitioners actually do to a current standard? Why would we restrain improvisation by insisting that a 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? We should welcome continual process reconstruction because it adapts the process to the project.

Improvisation suggests modeling what process users actually do and that SPI efforts integrated into ongoing software projects define what team members do now, not what they should do in the future. We should also curb the modeling process’ level of compulsion, defining key objectives and deliverables rather than obligatory procedures. Then, instead of mandating ever more elaborate process models, we can offer simple and flexible process models for highly motivated and competent people. As part of their SPI efforts, organizations should supply a range of alternative methods and procedures to facilitate developers’ work.

SPI is a problematic activity with considerable implications for practitioners’ beliefs, values, emotions, perceptions, and behaviors. Traditional process designers tend to be obsessed with control—we seem to believe that we should control software processes much as we control software. This belief has little merit, and it’s time to let go of it and stop approaching SPI in a linear step-by-step fashion, where we translate software process goals into solutions, then practice. Instead, we should put competent process users at the center of process design, using and building simple, open process models suited for improvisation—as Weick put it, a little structure goes a long way.

But if we let go of Blueprint SPI, will we end up with lax, ad hoc software processes? Will we have to give up the idea of discipline and structure in software development? Agile processes such as Extreme Programming show that we can delegate SPI to process users without sacrificing discipline. XP is a highly disciplined process and complementary with CMM and other models. Indeed, the CMM seeks to standardize at the goal level while permitting great discretion about how to achieve these goals. I propose that we administer this discretion at the process user level, and that recipes offer a robust means to do so.
Acknowledgments
I presented parts of this article at the Tenth European Conference on Information Systems (ECIS 2002), Gdansk, Poland, 6–8 June 2002. I am deeply indebted to Lars Mathiassen, Jesper Arent, and the anonymous reviewers for their comments and suggestions, and I also thank the software organizations, practitioners, and researchers participating in the Danish SPI initiative for stimulating the ideas in this article.

References


About the Author
Ivan Aaen is an associate professor of computer science at Aalborg University, Denmark. His interests include software engineering and information systems. He holds a PhD in computer science from Aalborg University and is a member of the ACM and IEEE Computer Society. Contact him at the Dept. of Computer Science, Aalborg Univ., Fredrik Bajersvej 7E, DK9220 Aalborg, Denmark, aamen@ieee.org.

September/October 2003 IEEE SOFTWARE 93