Improvisation in Small Software Organizations

Improvisation can give valuable insights into the relationship between action and learning in small, software-intensive organizations. As this article describes, a specific challenge involves balancing the refinement of the existing skill base with the experimentation of new ideas to find alternatives that improve on old ideas.

Tore Dybå, *SINTEF, Norway*

We are witnessing an explosive growth in the size and complexity of problems that software can address. Software organizations and their environments—which includes their market conditions, customers, suppliers, and recruitment base for future employees—are also experiencing complex interactions, as well as the implications of rapid changes in technology. But there is an important distinction between acknowledging complex interactions and understanding or controlling them.

Unfortunately, researchers and managers within the software community tend to share a belief that success depends on the ability to predict changes in the environment and to develop rational plans to cope with these changes. Predictability, however, is a property of simple systems. Reality is different—an environment is not a simple system, is not predictable, is not entirely knowable, and is definitely not controllable by the software manager or software organization. As I'll use this article to show, the sooner we admit this to ourselves, the sooner we can develop more useful models for improving software development.

Alice and the Croquet Game

To most small software organizations, their environment is like the croquet game in *Alice in Wonderland*—everything constantly changes around the player. In Alice's game, the croquet balls were live hedgehogs, the mallets were live flamingoes, and the arches were card soldiers. The players all played at once, without waiting for turns, and they had to fight the hedgehogs, which constantly crawled away, the flamingoes, which often twisted themselves around into the opposite direction, and the card soldiers, who frequently abandoned the game. As Alice did, small organizations face environmental turbulence. They require an improvement approach that recognizes

- the need for a dramatically shorter time frame between planning and action;
- that planning an action does not provide all the details of its implementation; and
- that creativity is necessary to make sense of the environment.

Improvisation is an improvement ap-

0740-7459/00/$10.00 © 2000 IEEE

proach that can help us better understand the relationship between action and learning in small software organizations. For that reason, small software organizations should become more improvisational to survive in an increasingly turbulent and complex environment. Most research focuses on large organizations, but I'd like to explore the environmental turbulence that small organizations face. Software organizations should pay more attention to future demands and seek the opportunities that are inherent in experimenting. Improvisation links a strong skill base with such experimentations and could be a viable alternative for improvement in small software organizations.

Improvisation

Improvisation deals with the unforeseen. It involves continual experimentation with new possibilities to create innovative and improved solutions outside current plans and routines. The explorative nature of improvisation necessarily involves a potential for failure, leading to the popular misconception that improvisation is only of a spontaneous, intuitive nature that happens among untutored geniuses or in immature organizations. However, organizational improvisation does not emerge from thin air. Instead, it involves and partly depends on the exploitation of prior routines and knowledge. Paul Berliner asserted that "Improvisation involves reworking precomposed material and designs in relation to unanticipated ideas conceived, shaped, and transformed under the special conditions of performance, thereby adding unique features to every creation."[1]

Generally, there are different levels of improvisation, ranging from interpreting or minimally adjusting an already existing model (through embellishment and variation) to radically altering the original models.[1]

Improvisational activities that fall under interpretation and minor adjustment depend on the models with which they start, while extreme improvisation depends more heavily on past experience and memory. Berliner explained the role experience and knowledge plays in jazz improvisation when he described good jazz improvisers as having large vocabularies, repertory storehouses, and a reservoir of techniques. Hence, improvisation mixes previously learned lessons with the current setting's contingencies. This mix, however, points to improvisation's core paradox (which is also at the heart of software process improvement (SPI)): Too much reliance on previously learned patterns tends to limit the explorative behavior necessary for improvisation. Yet too much risk-taking leads to fruitless experimentation and repeated failures.

Improvisation in Software Development

Today, the dominant perspective on software development is rooted in the rationalistic paradigm, which promotes a product-line approach to software development using a standardized, controllable, and predictable software engineering process. From this perspective, the software literature advocates discipline and replacing routinized human labor with mechanical routines and factory-like automation. As a result, a large part of the SPI community has promoted a rational, "best practices" approach to SPI (see the "Current Practices" sidebar). The Software Engineering Institute has advocated the use of *statistical process control* techniques (specifically control charts) in recent technical reports, and it has proposed changes for the CMM to explicitly support the use of rigorous statistical techniques.

But software development is not manufacturing. It is not a mechanical process with strong causal models appropriate for a passive improvement approach based on SPC. On the contrary, software development is largely an intellectual and social activity, requiring the ability to deal with change above the ability to achieve stability.[2] Therefore, we need to distance ourselves from the assumptions underlying the rationalistic, linear model of software engineering and admit that reality for most small software organizations is a nondeterministic, multidirectional flux that involves constant negotiation and renegotiation among and between the groups shaping the software.

Balance

We shouldn't abandon discipline altogether—there needs to be a balance between discipline and creativity in software development.[3] This balance can be challenging, because losing sight of software work's creative, design-intense nature leads to stifling rigidity, but losing sight of the need for

Too much reliance on previously learned patterns tends to limit the explorative behavior necessary for improvisation.

Basically, there are three ways in which a software organization can improve its process capability using "best practice" models such as the CMM:[1] it can increase the average performance (the mean of the performance distribution), it can reduce the variance in performance (increase predictability), or it can use a combination of both. Increased average performance is a general feature of experiential learning, and it is clearly beneficial for competitive advantage. Reduced variance, however, is not necessarily an advantage. In fact, for most small software companies, competition can turn reduced variance into a major disadvantage.

To examine the best practice approach to improvement and its consequences for competitiveness in small software organizations, let's consider a simple model James March devised,[2] which assumes that survival is based on comparative performance within a group of competing organizations. Furthermore, each single performance is drawn from a performance distribution specific to a particular organization. The mean of the distribution reflects the organization's ability level, and the variance reflects the organization's reliability.

For small software organizations, performance samples are also small. Relative position does not depend on ability alone but is a consequence of ability and reliability.[3] Moreover, the competitive environment of most small software organizations is such that only the best survive—and survival depends on having an extreme performance draw. Thus, improving average ability helps relatively little, and increasing reliability (reducing variance) can detrimentally affect survival.

In an extreme case, where the organization faces only one competitor, increases in average performance always pay off, whereas changes in variance have no effect on competitive advantage. However, when there are several competitors, increases in either the mean or the variance have a positive effect. As the number of competitors increases, the variance's contribution to competitive advantage increases. Ultimately, as the number of competitors goes to infinity, the mean performance becomes irrelevant.[2]

The argument behind CMM's improvement approach is that as the organization standardizes software processes and the developers learn techniques, the time required to accomplish development tasks will reduce and productivity and the quality of task performance will decrease together with the reliability of task performance.[4] March's model implies that if the increase in reliability comes as a consequence of reducing the performance distribution's left-hand tail (see Figure A1), the likelihood of finishing last among several competitors is reduced without changing the likelihood of finishing first. But, if process improvement reduces the distribution's right-hand tail, it might easily decrease the chance of being best among the competitors despite increases in the organization's average performance.

An improvement strategy that simultaneously increases average performance and its reliability is, therefore, not a guarantee of competitive advantage (see Figure A1). The consequence of such a strategy is that it helps in competition to avoid relatively low positions, whereas it has a negative effect in competition, where finishing near the top is important. Thus, the price of reliability is a smaller chance of primacy.

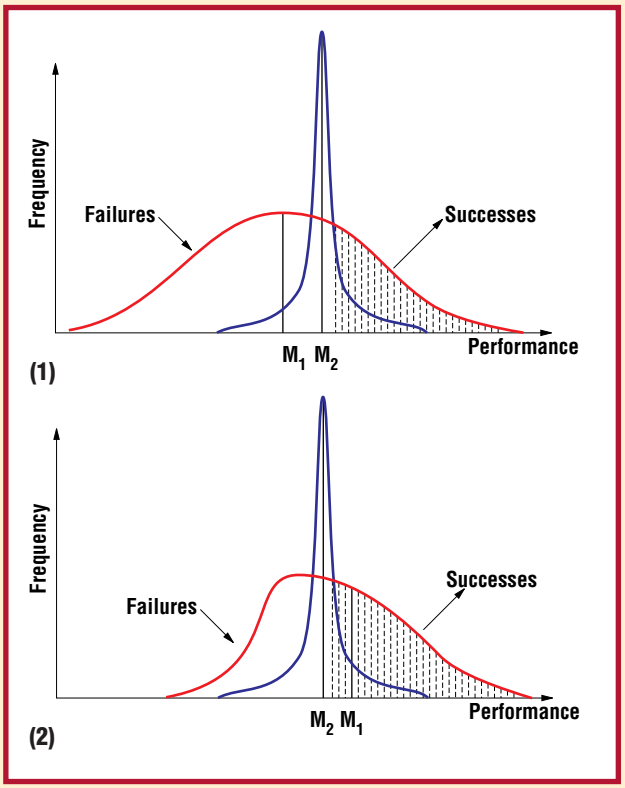

Figure A. The impact of (1) ability and (2) reliability on performance. M denotes the mean performance of the distributions.

If our main goal is to increase the competitive advantage of small software organizations, it's time to move away from the model-based, one-size-fits-all thinking of the 1990s. Instead, we should proceed to improvement strategies that focus on learning from our success to increase the performance distribution's right-hand tail while at the same time reducing the left-hand tail by learning from our failures (see Figure A2). Improvisation is thus a potential approach for such strategies to succeed.

References

1. W.A. Florac and A.D. Carleton, *Measuring the Software Process: Statistical Process Control for Software Process Improvement*, Addison-Wesley, Reading, Mass., 1999.
2. J.G. March, "Exploration and Exploitation in Organizational Learning," *Organization Science*, Vol. 2, No. 1, Feb. 1991, pp. 71–87.
3. D.A. Levinthal and J.G. March, "The Myopia of Learning," *Strategic Management J.*, Vol. 14, Winter 1993, pp. 95–112.
4. M.C. Paulk et al., *The Capability Maturity Model: Guidelines for Improving the Software Process*, Addison-Wesley, Reading, Mass., 1995.

discipline leads to chaos. In software, as in jazz, discipline enables creativity.[4]

A distinct characteristic of software development, as with all improvisational processes, is the fact that we cannot specify results completely at the outset of the work process. This

means that we can only outline the planned software products. Therefore, improvisation differs from the rational approach in that there is no detailed plan and that planning and executing an action converge in time. This last point is particularly important because time to market often determines competitiveness.

Decisions

Improvisation in software development leads to an emphasis on how developers interpret the environment and on how we make choices in an open situation (where there is more than one possibility). We make such choices by selecting the aspects we consider relevant for modeling, making available modes of interaction with the computer, determining the system's architecture, and deciding how to use technical resources for system implementation. Moreover, we make choices when creating tools and constraints for users and other concerned parties. Ultimately, we choose how we conduct the development process itself.

Only a small part of these choices are made explicit in terms of predecided plans. Usually, they are implied by the course of action we take, or as Donald Schön argues: "Our knowing is *in* our action."[5] Furthermore, each practitioner treats his or her case as unique and consequently cannot deal with it by applying standard theories or techniques. Also, our interactions with others constrain our choices. When seen in these terms, the task of software development clearly involves a large portion of improvisation, and thus social context and technological content are essential to a proper understanding of software development.

As in the case of a jazz band, close and sustained interaction between professionals stimulates creativity in such a way that the team performs better than its individuals could do alone.[4] The best teams are those that can honor the individualism of their members and at the same time act as a unit.

Implications for SPI

Having discussed the concept of improvisation and the consequences of CMM-based improvement (see the sidebar), let's now turn to the implications for SPI in small software organizations. There are many challenges for an improvisational approach to SPI to succeed. Two of the most important challenges are to sustain exploration and to learn from failure.

Exploitation and Exploration

Software organizations can engage in two broad kinds of improvement strategies. They can engage in exploitation—the adoption and use of existing knowledge and experience—or exploration—the search for new knowledge, either through imitation or innovation.[6] The basic balance problem is to undertake enough exploitation to ensure short-term results and, concurrently, to engage in exploration to ensure long-term survival.[3] A software organization that specializes in exploitation will eventually become better at using an increasingly obsolete technology, while an organization that specializes in exploration will never realize the advantages of its discoveries (see Table 1).

Improvisation requires both exploitation and exploration. Determining the appropriate balance is a complicated dynamic that involves considerations of both organizational size and environmental factors. Because exploitation generally generates clearer, earlier, and closer feedback than exploration, the

The task of software development clearly involves a large portion of improvisation, and thus social context and technological content are essential to a proper understanding of software development.

Table 1

Exploitation versus Exploration

Exploitation	Exploration
Refinement, routinization, and elaboration of existing ideas, paradigms, technologies, processes, strategies, and knowledge.	Experimentation with new ideas, paradigms, technologies, processes, strategies, and knowledge to find alternatives that improve on old ones.
Provides incremental returns on knowledge and low risk of failure.	Provides uncertain but potentially high returns on knowledge and carries significant risk of failure.
Requires personnel who are skilled in existing technologies.	Requires personnel who are skilled in emerging or innovative technologies.
Can generate short-term improvement results.	Often requires a long time horizon to generate improved results.

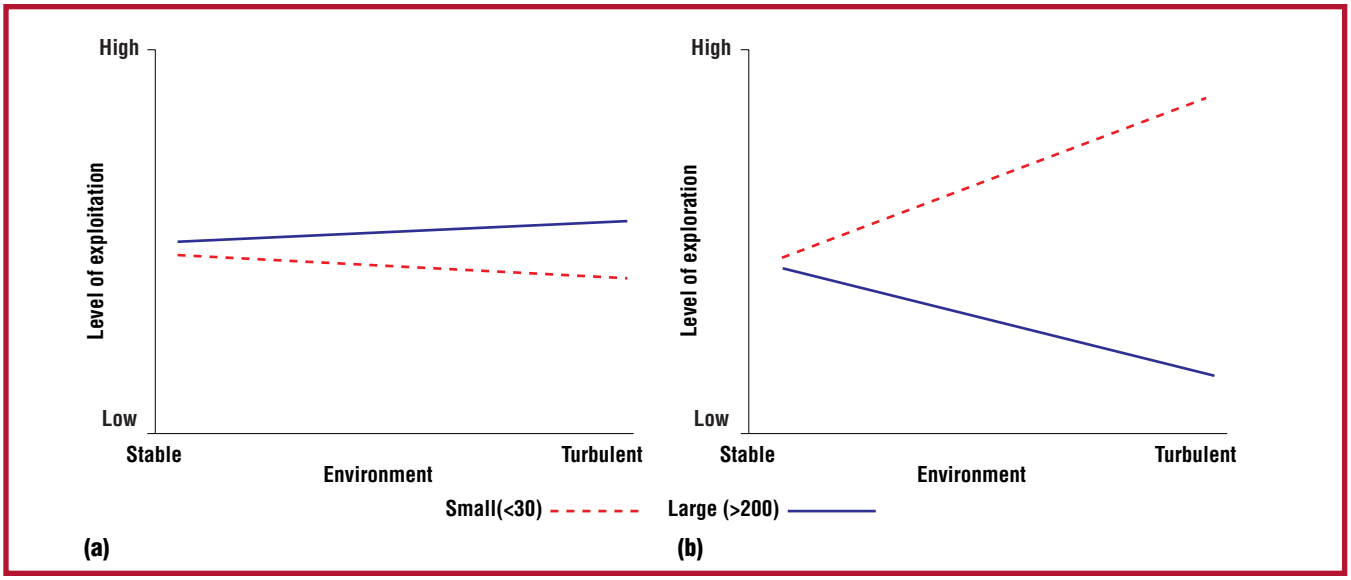

Figure 1. Improvement strategy versus organizational size and environmental turbulence for (a) exploitation and (b) exploration in small and large organizations.

most common situation is one in which exploitation tends to drive out exploration.[7] To make improvisation possible for organizations operating in increasingly more complex and turbulent environments, it is therefore of vital importance that they can increase exploration while sustaining exploitation.

Survey

I studied the balance between exploitation and exploration in a survey of 120 software and quality managers representing whole organizations or independent business units within 55 Norwegian software companies. Specifically, I studied how organizational size and environmental turbulence affected balance. This was done by comparing the effects of organizational size by contrasting large organizations with small ones. I defined the groups such that large organizations consisted of the upper third of the distribution (more than 200 developers) and small organizations of the lower third (fewer than 30 developers). Similarly, I examined the effects of environmental turbulence by contrasting the upper and lower third of the operationalized environment distribution. (For further details about the survey, contact me at tore.dyba@informatics.sintef.no.)

Results showed that small software organizations kept the same level of exploitation both in stable and turbulent environments. However, they engaged in significantly more exploration in turbulent environments than they did in stable environments. The increased level of exploration did not drive out exploitation. In other words, we found support for the proposition that increased environmental turbulence required increased levels of improvisation.

Similar to small organizations, large software organizations did not differ significantly in their level of exploitation between stable and turbulent environments. In contrast to the small organizations, however, increased turbulence did not lead to increased levels of exploration. On the contrary, the larger organizations seemed to lower their levels of exploration during turbulent times (see Figure 1). Thus, the results showed that small software organizations engaged in significantly more exploration in turbulent environments than large software organizations. This supports the assertion that small software organizations in turbulent environments require improvement strategies that are more closely aligned with explorative behavior, while simultaneously promoting the exploitation of past experience. This is at the heart of an improvement strategy based on improvisation.

Most software managers agreed that changes in their competitive environments are fast and increasingly unpredictable. However, managers of large software organizations still rely on learning from experience to prepare for the future rather than exploring new possibilities. They tended to generate the same responses even when the stimuli changed—they kept doing what they do well, rather than risk failure. One explanation for this, which is also a direct consequence of using improvement models such as the CMM, is institutionalized routines.

Large organizations tend to rely on formal routines for coordination—small organizations can coordinate their work through face-to-face communications and socializations to a much larger extent than large organizations can. In stable situations, such routinization can become an effective way of developing software, but it can also drive out the exploration of new alternative routines. The deeper these routines are grounded in the organizational culture, the more difficult they are to change and the more easily they can turn into an obstacle to improvement.

Learning through Failure

Inherent in the rationalistic approach to software development is to consider failure as unacceptable. This is consistent with the goal of promoting stability and short-term performance, as is the case in exploitation. In this situation, success provides an excellent foundation for increased reliability. Hence, success tends to encourage the maintenance of the status quo—if it ain't broke, don't fix it. However, the absence of failure can result in decreased organizational competence when faced with changing and turbulent environments. Improvisation requires tolerance for failure, and failure is an essential prerequisite both for learning and for challenging the status quo.

In software development, we must be able to turn unexpected problems and failures into learning opportunities.[8] Rather than treat failure as unacceptable and stigmatizing, we should distinguish between failures that result from carelessness and those that are a result of intelligent efforts to experiment outside existing patterns.[9] In this way, we can draw attention to potential problems and stimulate the search for creative and innovative solutions when the environmental factors change. The paradox is that you have to experience failure to have success.

Successful improvisation is difficult because it requires both modes of learning—exploitation and exploration. I hope this article will trigger a fruitful debate on how small—and large—organizations can balance these modes and better learn to improvise to cope with their constantly changing and often unpredictable environments.

I plan to further develop the concept of improvisation and organizational learning processes in software organizations, based on survey data as well as several case studies that I have participated in. Also, because software organizations are social systems with people and activities that interact according to certain theories of action, we should pay more attention to the inherent tensions between discipline and creativity, diversity and consensus, and knowing and doing, to mention just a few. Improvisation is a good metaphor for describing the tension between exploration and exploitation. A better understanding of improvisation could, thus, lead to a better understanding of the other tensions in software development and SPI.

The paradox is that you have to experience failure to have success.

References

1. P.R. Berliner, *Thinking in Jazz: The Infinite Art of Improvisation*, Univ. of Chicago Press, Chicago, 1994.
2. M.A. Ould, "CMM and ISO 9001," *Software Process: Improvement and Practice*, Vol. 2, No. 4, Dec. 1996, pp. 281–289.
3. R.L. Glass, *Software Creativity*, Prentice Hall, Englewood Cliffs, N.J., 1995.
4. W.S. Humphrey, *Managing Technical People: Innovation, Teamwork, and the Software Process*, Addison-Wesley, Reading, Mass., 1997.
5. D.A. Schön, *The Reflective Practitioner: How Professionals Think in Action*, Basic Books, New York, 1983.
6. J.G. March, "Exploration and Exploitation in Organizational Learning," *Organization Science*, Vol. 2, No. 1, Feb. 1991, pp. 71–87.
7. D.A. Levinthal and J.G. March, "The Myopia of Learning," *Strategic Management J.*, Vol. 14, Winter 1993, pp. 95–112.
8. T.K. Abdel-Hamid and S.E. Madnick, "The Elusive Silver Lining: How We Fail to Learn from Software Development Failures," *Sloan Management Rev.*, Vol. 32, No. 1, Fall 1990, pp. 39–48.
9. F.J. Barrett, "Creativity and Improvisation in Jazz and Organizations: Implications for Organizational Learning," *Organization Science*, Vol. 9, No. 5, 1998, pp. 605–622.

About the Author

Tore Dybå is a research scientist at the Department of Computer Science at SINTEF (the Foundation for Scientific and Industrial Research at the Norwegian Institute of Technology). He is also a research fellow in computer science at the Norwegian University of Science and Technology, working on a doctoral thesis investigating the key learning processes and factors for success in SPI. His current research interests include empirical software engineering, software process improvement, and organizational learning. He received his MSc in computer science from the Norwegian Institute of Technology. Contact him at SINTEF Telecom and Informatics, S.P. Andersens vei 15, N-7465 Trondheim, Norway; tore.dyba@informatics.sintef.no.