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-
approach 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.”

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. 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. 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. This balance can be challenging, because losing sight of software work’s creative, design-intensive nature leads to stifling rigidity, but losing sight of the need for
discipline leads to chaos. In software, as in jazz, discipline enables creativity.  

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

### Table 1

**Exploitation versus Exploration**

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

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

**References**


**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.