Overcoming Barriers to Self-Management in Software Teams

Nils Brede Moe, Torgeir Dingsøyr, and Tore Dybå, *SINTEF*

Lack of redundancy and conflict between team and individual autonomy are key issues when transforming from traditional command-and-control management to collaborative self-managing teams.

The basic work unit in innovative software organizations is the team rather than the individual. Such teams consist of "a small number of people with complementary skills who are committed to a common purpose, set of performance goals, and approach for which they hold themselves mutually accountable."[1] Work teams have many advantages, such as increased productivity, innovation, and employee satisfaction. However, their implementation doesn't always result in organizational success. It isn't enough to put individuals together and expect that they'll automatically know how to work effectively as a team.

A central question is thus, "How should you organize teamwork for software development?" Most development methods argue that teams should self-organize or self-manage.[2,3] Scrum, for example, is inspired by new product development teams in Japan,[4] in which self-management is a defining characteristic. Compared with traditional command-and-control management, self-managing teams represent a radically new approach to planning and managing software projects.

However, the notion of self-management isn't new; research in this area has been around since Eric Trist and Ken Bamforth's study of self-regulated coal miners in the 1950s.[5] Members of self-managed teams are responsible for managing and monitoring their own processes and executing tasks. They typically share decision authority jointly, rather than having a centralized decision structure where one person makes all the decisions or a decentralized decision structure where team members make independent decisions. So, whereas the traditional perspective of a single leader suggests that leadership is a specialized role that can't be shared without jeopardizing group effectiveness, shared leadership represents teams whose members are empowered to share leadership tasks and responsibilities.[1] Such leadership is, thus, a shared process in which important decisions about what to do and how to do it are made through an interactive process involving many people who influence each other, not just a single person.

It's one thing to talk about self-managing teams; it's quite another to enable such teams to work in practice. Here, we focus on the main barriers and challenges we've seen in empirical studies on the team and organizational level when introducing development methods that rely on self-managed software teams.

Benefits and Challenges

Self-managing teams offer potential advantages over traditionally managed teams because they bring decision-making authority to the level of operational problems and uncertainties and thus increase the speed and accuracy of problem solving. Companies have implemented such teams to reduce costs and to improve productivity and quality. Some research suggests that such teams result in higher employee satisfaction, lower turnover, and lower absenteeism.[6] Furthermore, having cross-

0740-7459/09/$26.00 © 2009 IEEE

Table 1

Teams and data collection sources

	Number of developers	When was agile introduced?	Team number	Team size	Project length (mos.)	Number of interviews	Number of observations
Company A	16	At the beginning of the projects	1	6	11	12	75
			2	6	12	12	45
Company B	50	In the middle of the projects	3	7	20	13	9
Company C	150	In the middle of the projects	4	8	30	11	10
			5	7	30	10	10

trained team members increases functional redundancy and, thus, flexibility in dealing with personnel shortages.

However, team performance is complex, and a self-managing team's performance depends on not only the team's competence in managing and executing its work but also the organizational context management provides. Although most studies report positive effects from self-managing teams, some present a more mixed assessment; they can be difficult to implement and risk failure when used in inappropriate situations or without sufficient leadership and support.[7]

What then is required for a team to successfully become self-managing? Understanding how to foster self-managing software development teams requires more than just examining the team's inner workings. We must also understand the organizational context surrounding the team because that's an important determinant of effectiveness.[7] For example, aspects of the organizational context such as reward systems, supervision, training, resources, and organizational structure can strongly affect team functioning.[6] Likewise, relationships with key stakeholders outside a team can influence task performance.

Study Context and Methods

For three years, we observed five teams in three companies (see Table 1) that introduced agile development practices to improve their ability to deliver iteratively and on time, increase software quality, and improve teamwork and team communication. Company A develops customer-specific software on a contract basis, developing systems for planning and work coordination. Company B manufactures receiving stations for meteorological and Earth observation satellite data. Company C develops both mass-market and customer-specific software for maritime, offshore, and process industries.

All five teams received the same training in agile development: one day of general introduction to scrum and one day of tailoring agile practices to their projects. Author Nils Brede Moe participated in the data collection from all companies, Torgeir Dingsøyr participated in companies A and B, and Tore Dybå participated in Company C.

Using a flexible research design, in which the research approach might change during the conduct of the research, we observed daily work and meetings (stand-up, planning, review, and retrospective), conducted interviews, and inspected documents. The observations lasted from 10 minutes to a full day. The interviews lasted from 20 to 45 minutes and were audio-recorded and transcribed. We interviewed most team participants twice and focused on understanding changes in the development process after introducing scrum, team communication, decision-making, planning, and coordination.

We documented our observations as field notes and pictures. We collected iteration and project plans, progress (burn-down) charts, and index cards used on project walls. We integrated all our notes to produce a detailed record of each session.

On the basis of this broad material, we used meta-ethnographic methods[8] to analyze and synthesize data from the transcribed interviews, dialogues, and field notes. In the first stage of the synthesis, we identified the main concepts from the transcriptions and field notes, using the subject's own terms. Then, we organized the key concepts in tabular form to enable comparison across teams and translate findings into higher-order interpretations. This process is analogous to the constant-comparison method used in qualitative data analysis.[9]

Self-management emerged as the key higher-order topic, which recurred across all teams and was challenging throughout all projects. This led us to focus the analysis on explaining key project events and how they related to the teams and organizations. We presented results to the companies in separate feedback meetings.

Figure 1. Team- and organizational-level barriers to self-managing software teams. The actual performance of a self-managing team depends not only on the competence of the team itself in managing and executing its work, but also on the organizational context provided by management.

Team-Level Barriers

On the team level, we found that barriers to self-management are related to individual commitment, failure to learn, and individual leadership (see Figure 1).

Individual Commitment

Self-managing teams are responsible for planning and scheduling their work, so team members need to genuinely commit to the team plan. When moving from more control and command-oriented management to collaborative self-management, many team members gave too much priority to individual goals over team goals. The main reason for this low team-level commitment was specialization; in all the teams we studied, developers worked independently on particular modules according to specialization (for example, user interfaces, map interfaces, and databases). Both managers and developers thought this was the most efficient way of working. One scrum-master (Company A) explained,

> *Let the person that knows most about the task solve it! It will take too many resources if several persons are working on the same module, and there is no time for overlapping work in this project. The tasks are delegated and solved the best possible way today. Maybe we can do more overlapping in the next project.*

Because developers usually worked alone on a module, they often created individual plans. Developers got full control over their own schedule and task implementation, which resulted in even more individual autonomy. Consequently, team members in companies A and B had less interaction than needed, which made team-level commitment difficult.

In addition, several developers said it's difficult to commit to work they aren't involved in and that if team-members got sick or side-tracked, part of the iteration plan wouldn't get done. So committing to the iteration plan meant committing to doing as much as possible on a developer's "own" module.

Another common obstacle to achieving shared commitment was that four teams created unrealistic plans. In each iteration, the team selected more tasks than they could feasibly complete. The plans became unrealistic because the team tried to make their plans too flexible. The idea was that everyone would find something to do even if the context around the team changed. One certified scrum-master (Company A) on his second scrum project commented on why they added too many tasks:

> *In the beginning there were so many dependencies to other projects and subcontractors, so we did not know which tasks we could complete during the sprint. It is also about optimization. You should always have the possibility to work on something else if you are stuck.*

Another barrier found in all teams was unclear completion criteria. Problems with defining what it means when a task is categorized as "done" or "completed" reduced the possibility of committing to the team goal. If a task was "done" but not tested well enough, the team started the new iteration without doing the work necessary to make the task of the previous iteration truly complete. One developer on his second scrum project (Company A) told us,

> *The tasks were not tested well enough, and we knew there was still work to be done. These tasks are then not on the list of the next iteration since they officially are done. … Each iteration starts with doing things that we have said were finished, and then you know you will not finish the iteration. … When during an iteration you realized that you would not finish, you do not care if you are 70 percent or 90 percent done.*

Finally, meetings that weren't engaging lowered shared commitment. Teams in companies A and B used a scrum tool to support planning. Typically, one person—the scrum-master—was in charge of registering the tasks in this tool during the meetings. When he typed in the tool, discussions effectively stopped. And when the person in charge of the tool needed to clarify some issues, he or she usually started talking directly to the developer most likely to be assigned to the task. Because other developers

often found these discussions irrelevant, they didn't listen. We even observed developers falling asleep in companies A and B. These meetings weren't a good arena for discussion and team commitment.

Failure to Learn

Teams must be able to change the operating norms and rules within the team, as well as in the wider environment, to become and remain self-managed. We observed teams frequently discussing the need for change in daily stand-ups, planning, and retrospective meetings. Nevertheless, all teams had process improvement problems.

Why was it so difficult for these teams to improve? One reason was the low level of *team autonomy*. The team needs to affect managerial decisions that influence its ability to improve its internal processes. People outside the team also need to respect its efforts at improvement; otherwise, it will only experience *symbolic self-management*, a well-known obstacle to true self-management.[10]

The product owner moderated team-level autonomy. In one project (Company C), after visiting customers, he always had some emergent issues that he needed to fix, an urgent request for a new demonstration, or new goals for the project. He usually expected the team to solve these issues during the ongoing iteration, which made it difficult for the team to stick to their plan. It also interrupted collaboration. This was how he'd always worked in the past, and introducing a scrum-master to protect the team against disturbances didn't make much of a difference. Also, when planning, the product owner usually presented features with many uncertainties, which made it difficult for the team to estimate the effort required and resulted in poor planning meetings. Team members told us their relationship with the product owner was their biggest challenge but that they found it difficult to confront him about these matters. Their inability to improve relationships with people outside the team discouraged the team members from trying to improve in other areas and clearly limited the possibility of continuous learning.

Another reason for failure to learn was *impression management*. Some scrum-masters and team members gave the impression that the team was better than it actually was. The desire to keep the schedule overrode knowledge of serious problems with, for instance, a third-party component, testing, integration, or performance. One developer from Company A said,

> *We classified tasks as finished before they were completed, and we knew there was still work to be done. It seems that the scrum-master wants to show progress, and make us look a little better than we really are.*

Impression management is a face-saving process where the team members seek to protect themselves from management.[11] This generates shared norms and patterns of "groupthink" that prevent people from addressing key issues. We never observed anyone protesting when the plans were too optimistic or when unfinished tasks were reported as finished.

One reason for the impression management was that four projects had to fight for their resources. However, when management in Company A understood that the team wasn't performing as well as reported, they lost trust in them.

A third barrier to learning was specialization. Problems related to developers' "own" modules were often seen as personal and subsequently not reported to the group. A developer from Company A said during a retrospective,

> *When we discover new problems, we feel we own them ourselves, and that we will manage to solve them before the next meeting tomorrow. But this is not the case; it always takes longer time.*

Individual Leadership

In all the teams we studied, implementing shared leadership was troublesome because the team members didn't change their individual, decentralized decision-making process. Consequently, the highly specialized developers in companies A and B focused only on their "own" modules and had little interaction with others. This resulted in difficulties aligning decisions on the operational level because team members didn't know what others were doing.

In Company A, one developer decided to spend three days implementing features for future projects without informing any other team members. Such *decision hijacking* was an important barrier to implementing shared decision-making. As a result, this developer lost his team's trust. In addition, the scrum-master, who'd changed his behavior and seldom used his former decision-making authority, started to give him direct instructions. We observed the same phenomena in Company C. One scrum-master, after working on a project for 30 months, explained,

> *We divide tasks among ourselves, and then people are responsible for implementing them.*

The desire to keep the schedule overrode knowledge of serious problems.

Misalignment between team structure and organizational structure can be counter-productive.

This usually works fine. However, in the last sprint one of the developers spoke with someone in the market about an issue. The developer used a couple of days changing a feature that was already finished, because the developer thought this was very important. We had to redo what the developer had done.

In companies A and B, the scrum-masters made many decisions just as they had done before self-management was introduced. This was especially evident when projects had problems. In one project (Company A) already running late, the certified scrum-master said,

I probably influence what they do. It is difficult to keep your mouth shut when you feel you have some good recommendations to give. It's an old habit, difficult to change.

Another problem was for the team to identify who should be involved in which decisions. Not everyone can be involved in everything, but people shouldn't be left out. However, the team seldom discussed who should be involved in what. One experienced developer newly hired in Company C told us,

They usually tell me what to do. I think I possess more knowledge than they realize and that I get too simple tasks. I'm also missing a good overview of what we are going to deliver, because many of the discussions regarding the design were done without me. It's probably too expensive if everyone is to participate in every meeting, but for me it is difficult to see dependencies between the modules.

As his fellow team members became aware of his expertise, they involved him more.

Organizational-Level Barriers

Implementing self-managed teams is difficult, if not impossible, if there are critical barriers at the organizational level. Misalignment between team structure and organizational structure can be counterproductive, and attempts to implement self-managed teams can cause frustration for both developers and management.[12] At the organizational level we identified shared resources, organizational control, and specialist culture as the most important barriers (see Figure 1).

Shared Resources

In companies A and B, developers usually worked on two or more projects in parallel. When different team goals or needs were in conflict, it threatened at least one of the self-managing teams. In addition, some developers had to suddenly stop what they were doing and support projects they'd worked on earlier.

In one important project in Company A, the team had problems losing resources. Eight weeks before the first major delivery, the scrum-master was allowed to protect the developers against all kinds of external requests four days a week. The scrum-master collocated with the team, and even made team members leave their telephones unanswered. One developer commented on this situation:

The isolation works how it should, but we [the developers] end up in a dilemma. Like today, it's crazy in the other project. The customers are starting to use the system this week, and I'm the only one who can fix problems. Then I just need to help if they are having problems. I do not like to decide which project will not meet its deadlines.

Developers in companies A and B ended up doing unscheduled work because parts of the organization expected developers to work even if no resources were provided. This was part of the company culture. No one could quantify this unscheduled work. Because a team always knew they'd lose resources during an iteration, it didn't make sense to commit to the team plan.

Furthermore, the companies' highly specialized cultures made it difficult to change how teams were designed and how support was provided. Projects usually fought to get the most skilled employees, and the organizations rarely invested in building redundancy. So, the projects kept losing resources and found self-management difficult.

Organizational Control

In Company B, the tool for organizing project tasks also included information the quality department used. The self-managing team perceived this information as unnecessary, representing a form of detailed control from management, which required extra reporting from the project participants. Such mechanisms can be a barrier for self-management if the teams see them as unnecessary. In this case, the quality department could have chosen instead to follow up on the team's actual products and not on reporting.

In Company A, management was primarily interested in the number of hours reported, not the

actual progress on tasks. One day before a review meeting we overheard the scrum-master telling developers to report more hours than already reported. The team did this to cover up losing resources to other projects and support tasks. Covering up such problems is a type of impression management and subsequently a threat to self-management.

Specialist Culture

A self-managed team needs generalists—members with multiple skills who can perform one another's jobs and substitute as needs arise. However, we found that incentives in all companies often supported a culture of specialists. In Company C, one of the most attractive roles was "chief architect." In the project we studied, the chief architect participated in important decision meetings with management; the management trusted him, and he had much influence on their products' future strategy. He didn't involve others in the decision-making. This *holdup problem*[13] threatened the self-managing team. One developer described the situation:

> *The chief architect made most of the decisions himself. This was frustrating because we were not involved, and everything was hidden in the architecture.*

This chief architect later quit Company C. This allowed the whole team to participate in important decisions and then to self-manage. One manager we spoke with in Company C disliked the new situation, although he observed that his team was more efficient when more self-managed. He felt that no one was taking care of his biggest concern: building an architecture that would last for 10 years.

In Company B we found developers protecting their knowledge—that is, protecting their code by not letting others work on it. If the code was important, then the developers became important to the company. Three years before introducing scrum, Company C had to let some developers go, but not any of the "important" specialists. So, letting others work on your code was considered a risk that could result in a loss of power. But being the only one working on important parts of the code was stressful during hectic periods and delayed the team when the developer was occupied with other work.

In Company A, this culture of specialization resulted in developers not wanting to take responsibility for important parts of the project because taking such responsibility meant becoming a specialist and getting stuck supporting the same product forever. During lunch, one developer described this problem of supporting all her old projects as a "quagmire":

> *You get stuck in the quagmire. And for every new project you are on, it gets worse.*

This resulted in a lack of shared commitment.

Overcoming the Barriers

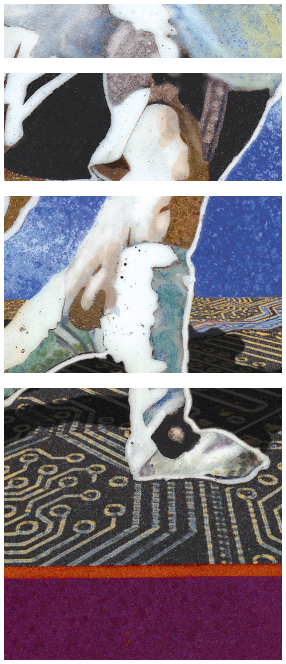
We've seen that team- and organizational-level barriers can challenge a team's self-management ability. Fostering self-management must therefore involve mechanisms that affect all levels.

The individual-commitment and individual-leadership barriers indicate that teams have difficulty balancing individual and team autonomy. Achieving greater redundancy can reduce this problem. When people work together they're more likely to commit to common goals and making team decisions. A greater focus on redundancy will also reduce problems with a specialist culture at the organizational level. However, we observed that the organizational level saw redundancy as unnecessary and inefficient. So, developers could seldom substitute for one another. When the team was also sharing resources, the lack of redundancy led to little flexibility.

The organizational-control barrier indicates that teams have inadequate autonomy. Low team autonomy was a reason we observed teams failing to learn. Team members also didn't see the point of committing to team goals because they knew that the organization often wouldn't respect their decisions—for example, by giving resources to other projects. The teams experienced symbolic self-management, which made it difficult to become self-managed. So, adequate team autonomy is crucial to reduce problems associated with barriers to self-management.

There's no guaranteed how-to recipe to foster self-managing teams, and the question is thus how much redundancy and team-level autonomy should be built into the organization and how to do it. Like the tale of the sitar player asking Buddha how best to tune his instrument, we find the famous answer, "Not too tight, and not too loose," as a good general rule on the degree of redundancy and team autonomy that would meet individual, team, and organizational needs. On the basis of the findings in this study, we recommend the following:

- *Organize cross-training.* This is costly, but the alternative can be even more expensive. With little or no redundancy, the company becomes vulnerable to changes. Pair programming



About the Authors

Nils Brede Moe is a research scientist at SINTEF Information and Communication Technology. His research interests include global software development, process improvement, self-management, and agile software development. Moe has a master's of science degree in computer science from the Norwegian University of Science and Technology. Contact him at nilsm@sintef.no.

Torgeir Dingsøyr is a senior scientist at SINTEF Information and Communication Technology and an adjunct associate professor at the Norwegian University of Science and Technology's Department of Computer and Information Science. His research interests include software process improvement and agile software development. He received a doctoral degree on knowledge management in software engineering from the Norwegian University of Science and Technology. Contact him at torgeird@sintef.no.

Tore Dybå is chief scientist and research manager at SINTEF Information and Communication Technology. His research interests include empirical and evidence-based software engineering, agile development, and organizational learning. Dybå has a doctoral degree in computer and information science from the Norwegian University of Science and Technology. He's a member of the International Software Engineering Research Network and the IEEE Computer Society. Contact him at tored@sintef.no.

and job rotation address this vulnerability by increasing team flexibility.

- *Collocate the team in the same room.* Collocation will make people discuss the tasks they're working on and discuss problems more frequently than if the team is distributed. People not used to talking to each other will start talking after a while if they're collocated. However, it's important to balance individual-level and team-level autonomy by letting team members work uninterrupted when needed.
- *Appreciate generalists.* Company culture and company incentives must appreciate both generalists and specialists to build redundancy into the organization. When recruiting, select people with the potential to develop redundant skills.
- *Build trust and commitment.* To build trust in the whole organization, management should avoid any control that would impair creativity and spontaneity. The teams' need for continuous learning, not the company's need for control, should motivate the need for data collection. Also, the teams should beware of any signs of impression management. When people work together toward a common objective, trust and commitment follow. So, make sure both the organization and the teams know and respect the common objective.
- *Assign people to one project at a time.* If possible, let team members focus on one project at a time. We found this easier in the larger organizations we studied. Resources must be coordinated, and management, not the team members, must decide when other projects or support requests should get resources. If the team members decide which project to work on, they'll yield to the one making the most noise. This will damage the self-managed team's potential.

We hope our study will focus attention on the organizational context of teams and ensure that the transition to self-managed teams isn't taken lightly. Important changes must be identified and discussed on both the team and organizational level. We plan to do more longitudinal studies on the mechanisms of effective teamwork and coordination across teams in software development organizations.

References

1. J.R. Katzenbach and D.K. Smith, "The Discipline of Teams," *Harvard Business Rev.*, vol. 71, no. 2, 1993, pp. 111–120.
2. L. Rising and N.S. Janoff, "The Scrum Software Development Process for Small Teams," *IEEE Software*, vol. 17, no. 4, 2000, pp. 26–32.
3. B. Schatz and I. Abdelshafi, "Primavera Gets Agile: A Successful Transition to Agile Development," *IEEE Software*, vol. 22, no. 3, 2005, pp. 36–42.
4. H. Takeuchi and I. Nonaka, "The New New Product Development Game," *Harvard Business Rev.*, vol. 64, vol. 1, 1986, pp. 137–146.
5. E.L. Trist and K.W. Bamforth, "Some Social and Psychological Consequences of the Longwall Method of Coal Getting," *Human Relations*, vol. 4, no. 1, 1951, pp. 3–38.
6. S.G. Cohen and D.E. Bailey, "What Makes Teams Work: Group Effectiveness Research from the Shop Floor to the Executive Suite," *J. Management*, vol. 23, no. 3, 1997, pp. 239–290.
7. J.R. Hackman, "The Design of Work Teams," *Handbook of Organizational Behavior*, J. Lorsch ed., Prentice-Hall, 1987, pp. 315–342.
8. G.W. Noblit and R.D. Hare, *Meta-ethnography: Synthesizing Qualitative Studies*, Sage, 1988.
9. M.B. Miles and M. Huberman, *Qualitative Data Analysis: An Expanded Sourcebook*, 2nd ed., Sage, 1994.
10. J. Tata and S. Prasad, "Team Self-Management, Organizational Structure, and Judgments of Team Effectiveness," *J. Managerial Issues*, vol. 16, no. 2, 2004, pp. 248–265.
11. G. Morgan, *Images of Organization*, Sage, 2006.
12. M. Uhl-Bien and G.B. Graen, "Individual Self-Management: Analysis of Professionals' Self-Managing Activities in Functional and Cross-Functional Work Teams," *Academy of Management J.*, vol. 41, no. 3, 1998, pp. 340–350.
13. R.F. Freeland, "Creating Holdup through Vertical Integration: Fisher Body Revisited," *J. Law & Economics*, vol. 43, no. 1, 2000, pp. 33–66.