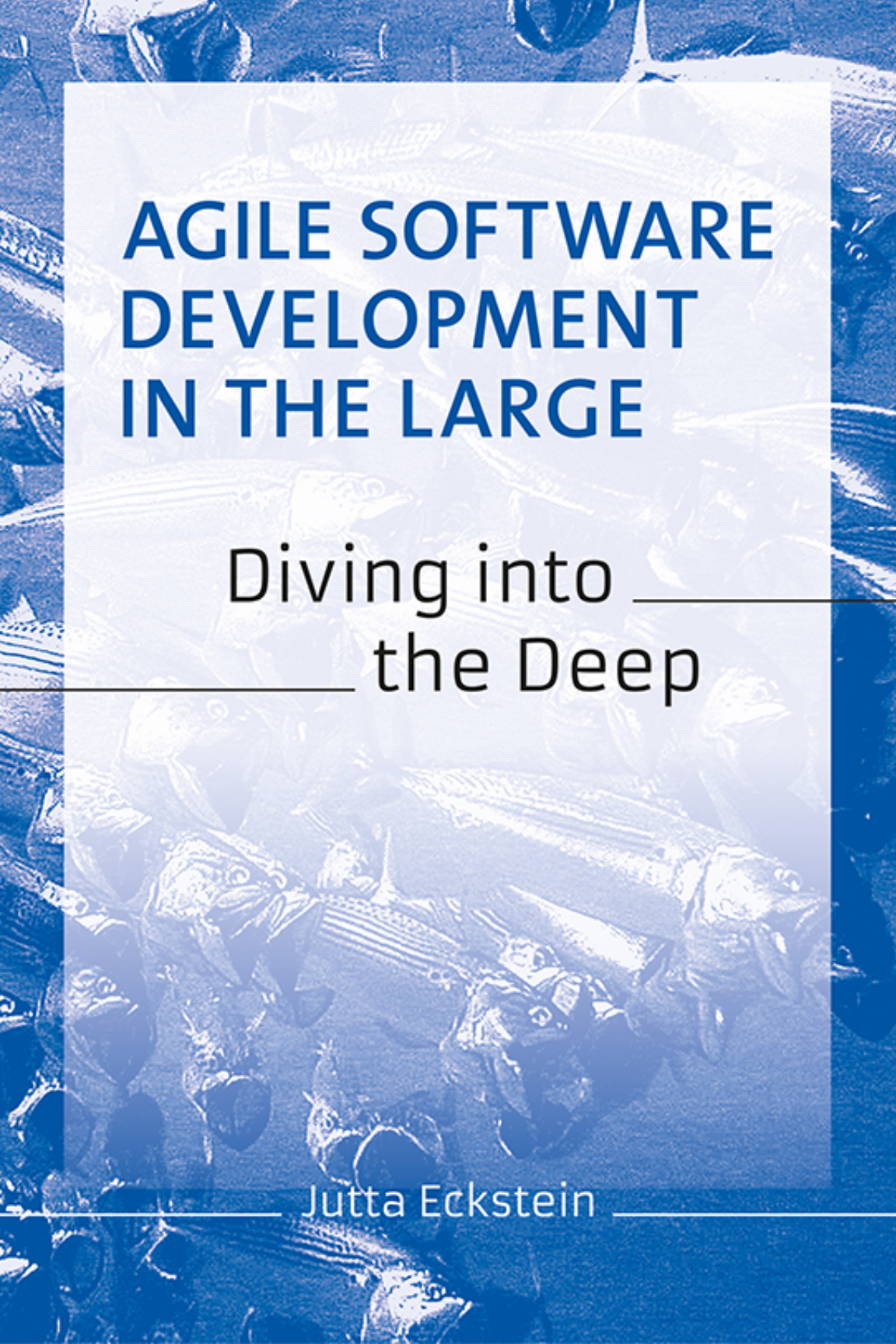

AGILE SOFTWARE DEVELOPMENT IN THE LARGE
Diving into the Deep
Jutta Eckstein

Agile Software Development in the Large

Diving into the Deep

Jutta Eckstein

This book is for sale at http://leanpub.com/agileinthelarge

This version was published on 2023-01-18

ISBN 978-3-947991-26-6

Also By Jutta Eckstein

Diving For Hidden Treasures

Retrospectives for Organizational Change

Company-wide Agility with Beyond Budgeting, Open Space & Sociocracy

Agile Software Development with Distributed Teams

Agiles Projektmanagement Kurz und Bündig

Agilidad empresarial con Beyond Budgeting, Open Space y Sociocracia.

En busca de tesoros ocultos

Contents

1. Dealing with Large Teams

Trust is the sister of responsibility.

– Asian proverb

The reasons for implementing a system with a large team are varied. The most common one is that the scope of the project is too large for a small team to handle. Of course, there are some large projects that would be better off if implemented by a small team. So, even if its scope is large, it might still be faster (or even better) to develop a project with a small team, mainly because communication is not as likely to prove a problem as it is in a large team.

The use of a large team could also be politically motivated. The size of a team sometimes reflects the importance of the project and, often, of the project management as well. This alone could be reason enough to implement the system with a large team. Tom DeMarco discussed this problem during OOPSLA 2001[1]. He indicated that surprisingly often, the manager of a failed, but large project will be valued higher than the manager of a successful but small project.

Furthermore, it could be the case that the team is already established and the project is shaped (and sized) to suit the team. For instance, I witnessed a situation in an organization where a lot

[1]OOPSLA is an ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications

of people just sat around, waiting for the project to start. Nobody questioned if this mass of people was really required for the project. Instead, everybody tried to shape the project in a way that kept all these people busy. Granted, for some companies (in some countries) it might be easier to shape the project according the team size than to get rid of the employees—mainly because of legal issues—but this is not usually the case.

Of course, it is always worth questioning the reasons for working with a large team, but this is neither the topic of the book in general nor of this chapter in particular. Instead, the assumption is that the project will be run by a large team and you (still) want to use an agile process to succeed. When changing to agile development with a *large* team, you have to deal with several issues involving people, teams, interactions and communication structures.

This chapter focuses on those aspects of agile processes that work differently in large teams than in smaller teams. First we shall look at the people aspect. There we will discuss how taking up responsibility can work in a large team and what kind of consequences respect, acceptance and trust have for successful collaboration. Next we will talk about how a large team can be organized in several subteams and what kind of team roles have to be occupied. In the section on interaction and communication structures we shall focus on encouraging communication in large teams. Next, in the section trouble shooting I will present typical team problems and their possible solutions. Finally we will look at the difficulties that can occur when developing with dispersed teams.

1.1 People

Size matters. The size of a team provides a special risk—a team that is too large can prove to be a hindrance to the project. One of the reasons is that the quality of decisions typically suffers. For example, the larger the team the more often you will find that

decisions are unclear or postponed. The main reason for this is that within large teams you will often find a tendency among people to shun responsibility. Because there are so many people on the team, there is a collective mentality of "someone else will decide."

Unclear or postponed decisions confuse the team and make it difficult for team members to decide which direction to take. This leads either to project paralysis, because nobody has the courage to move on without being told, or to a lot of individual decisions as one thinks best. Often, those individual decisions contradict each other, which in turn leads to a form of project stagnation, based on contradictory development. Both symptoms are very frustrating for the whole team. I once consulted on a restart of a failed project. I interviewed the team about what they believed was the trap that the restart was most likely to fall into. Interestingly enough, most the people named a lack of clear decisions as the highest risk.

Therefore, although it might seem unusual, it is preferable to make a clear but eventually wrong decision and to correct it later on. Making a wrong decision enables you to learn; postponing a decision does not. If you postpone a decision, you do not know until it has been made whether it is the right or the wrong one. However, if you make the wrong decision, you will learn from the consequences of that decision and have the possibility of correcting your mistake, based on your new experience.

Making decisions is one side of the coin, the other is making sure that they are not only communicated to everybody involved, but are also carried out. A decision that is made but not carried out is essentially the same as a postponed decision. Only when realizing the decision and living with its consequences will you know if it was right or wrong.

Although this all sounds very obvious, it is common to find the same problems popping up over and over again, which is a sure sign that those decisions have either never been clearly made, or they have not been realized.

As I mentioned earlier, the main reason for the poor quality of decisions on projects with large teams is probably based on an aversion to taking responsibility. You will find that the more people are working on a project, the harder it is to tell who took up the responsibility for which task. Often this results in an *undefined task zone*, which is defined by:

- **Multiplicated task responsibility**: A lot of people are responsible for the same task. The problem is that they do not know about one another. Therefore, if you are lucky, this task will be carried out repeatedly. If you are unfortunate, they will do the task in ways that contradict each other.
- **Null task responsibility**: Nobody takes up the responsibility for the task. Everybody assumes that it is someone else's job. Funnily, enough, this can result in everybody blaming everybody else for not taking up the responsibility.

To make things worse, you can be assured that with each additional team member, the risk will rise and that more of such problems will arise.

Responsibility

Due to the departmental organization, people in large companies are not usually used to having complete responsibility for any particular task. This is because there is almost always somebody higher up the hierarchy who has *ultimate* responsibility for the task. This is especially true for developers. They often consider themselves to be the ones who only do whatever somebody else tells them. No wonder they act as they do whenever somebody "accidentally" gives them the responsibility for a specific task. They feel uncertain when given responsibility because they are not used to having it, and because they do not know what it implies.

On the other hand, agile processes require that everybody be responsible for her task, and for the effects that task might have

on the whole project. As well as those tasks, there is also the shared responsibility of the ultimate performance of the whole system, project, and even the process that will lead the development. Thus, each team member is responsible in some way for every task, even those assigned to other team members.

For example, *Extreme Programming* has a practice called *collective ownership,* which refers to a shared responsibility for all kinds of things: the code, the integration, the process, and so on. Best known among these shared responsibilities is probably collective *code* ownership, which enables and obliges everybody on the team to improve every piece of code, no matter if he or she is the original author of the code or not.

With collective ownership, every team member bears the same responsibility for all aspects of the project. While allowing everybody to steer the project at the same time is a challenge, and, some fear, a big burden. For instance every developer would want to have a hand in shaping his or her development environment. On the other hand, this increased responsibility is likely to increase fear among the developers of making the wrong decisions.

This is why, when people first sign up for a task but aren't used to the responsibility it entails, you have to lead them gently into this new ground. For example, ask the developers which task they want to be responsible for, and then assist them in estimating the task. Not only should you make yourself available to answer any questions they may have, it is very important that you also ask them regularly if they are doing ok, or if they need any help, because they might be afraid of bringing up such issues themselves.

For example, I remember one project I was working on, where people had problems taking responsibility. I visited all the team members regularly and asked them how they were getting along with their tasks. It did not take long before some of them started complaining that they were not able to get their work done, for various reasons. The most common reply was that they were

waiting for something from another team: either the other team had not yet provided some interfaces, or the interface they had provided turned out to be different than expected. The obvious problem was that these people did not have a consciousness of problem solving. Instead, they complained that their peers were responsible for the problems. However, the real, hidden problem was that they were not taking enough responsibility. If they had, they would not have complained, but rather would have started solving their problems. In other words, they might have started talking to this other team and attempted to find out why the interfaces were not ready and addressed the situation.

Whereas the typical reaction of people not used to responsibility is to get annoyed at the situation without taking any action to change it. Of course, it could be worse. If, for instance, they could neither complain nor take up the responsibility, you would never learn about their problems.

Therefore, you have to be proactive in asking them about the status of their assigned task. Only then you will have an idea of any problems they may have. I am not talking about the typical status reports, instead I am talking about walking up to the people and talking face-to-face about their current situation. You should encourage them to look at the big picture, and regard their assigned task as a part of the whole. Explain to them that in doing so, they will sometimes have to do things that are only partly related (if at all) to their assigned task, but are important for the completion of the project.

If people are spoon-fed responsibility, they will not learn to make an effort to take it up themselves. Or, as *Fast Company* puts it:

> "Telling people what to do doesn't guarantee that they will learn enough to think for themselves in the future. Instead, it may mean that they'll depend on you or their superiors even more and that they will stop taking

> chances, stop innovating, stop learning. [2]"

Thus telling people what to do is not enough, they have to commit themselves for taking up a task. The focal point of this philosophy is that the value of team productivity is much more important than the individual effort. Therefore, every now and then you have to point out that only the team's success is the individual's success. An individual's success without the success of the team is of no value. Among other things this means that a well functioning team does not rely on its official manager—it takes up the responsibility itself whenever the situation requires it. As Kent Beck said:

> "Leadership happens every time, every minute by everybody on the team. [3]"

For this ideal situation to become a reality, the organization has to change from management by command and control to management by responsibility, trust, and teamwork.

Trust is the foundation on which such a management strategy is built. Because when someone takes on a responsibility, you trust that he or she is capable of handling this responsibility. However, at the start of an organization's first agile project, this culture of trust and responsibility will not be in place yet. Most team members will not be able to take up responsibility because they are not used to it. However, I suggest that you demonstrate to them how you take up responsibility, and that you encourage them to take responsibility even if they do not feel ready. This shows your team members that you trust them, even though at this early stage in agile development they might not be able to justify your trust. When this is the case, the most common reaction is to refrain from giving them any responsibility in the future. But this prevents them

[2] *Fast Company* on the Web: https://tinyurl.com/2p8ed7e6

[3] Kent Beck, Keynote at the International Conference on eXtreme Programming and Agile Processes in Software-Engineering 2001, Sardinia, Italy.

from ever getting the chance to learn how to take up responsibility, and simply reinforces their own mistrust in their capabilities. Just as Ulrich Sollman and Roderich Heinze[4] say, you should give people the chance to learn how to deal with responsibility:

> "The more often you are in an uncertain situation the better you can handle this kind of situation, or rather the longer it will take till you will again feel uncertain. [5]"

If you want to train your team members to take up responsibility, you have to be aware that this is an investment in their future. This "training" is two-sided: you might also have to train the leaders to pass on responsibility and to trust their team members. Like every other learning process, it will be some time before you see results, but it is worth the effort.

Respect and Acceptance

A development team is not usually organized like a team, in the strictest sense of the word, assembled by peers with equal rights, but more like a hierarchy. The typical hierarchy in a development team, which can be found mainly in traditionally led projects, follows Taylor's[6] theory about centralizing a team's knowledge. The individual team members take up specific roles and corresponding tasks. Analysts, designer, developer and tester often work independent from one another in a linear process.

As a consequence of this separation of tasks and roles, a hierarchy is created. Although this hierarchy might not officially exist, it

[4]Ulrich Sollmann and Roderich Heinze, *Visionsmanagement. Erfolg als vorausgedachtes Ergebnis* (Vision Management. Success as the predefined result.).(Zürich: Orell Füssli, 1994)

[5]Ulrich Sollmann and Roderich Heinze, *Visionsmanagement. Erfolg als vorausgedachtes Ergebnis* (Vision Management. Success as the predefined result.) (Zürich: Orell Füssli, 1994), p.32

[6]Taylorism is characterized by the division of labor, repetitive operations, extreme labor discipline, and the supervision of work.

is formed by the different roles in the team, some of which have greater prestige and/or importance (acceptance level) than others. Often, the acceptance level is defined by the linear development. This means that analysts have the highest acceptance level, whereas coders, testers, and, even worse, maintainers are at the very end of the acceptance level chain, doing all the "dirty" work. The presented sequence of acceptance levels is just one example, but an oft-encountered one.

However, the major problem is that nobody wants to be at the low end of this acceptance-level chain. Therefore (as in the example above), everybody tries to climb up the ladder from maintainer to designer or, even better, analyst. Or, if we look at it from another perspective, you will find the largest percentage of novices in maintenance or implementation. Consequently, there are often too few experienced coders in a team.

On the other hand, most agile processes require teams to have shared knowledge and shared skills. This means knowledge cannot serve to form a hierarchy. Therefore, the first step in forming an agile team is to get rid of the tayloristic split. Assemble teams that cover all the knowledge, where each member of the team is aware of the big picture and takes her responsibility to contribute to the whole team's success. As a consequence, the individual role of each member is not so obvious anymore, in terms of individual knowledge, but in terms of contribution to the team's success. So acceptance is then based on performance and not on roles. However, it is important to note that a main difference between a small and a large agile team, is that in a small agile team, typically every individual is requested to be a generalist. On the other hand, in a large agile team, a whole *subteam* (see below in this Chapter and not necessarily every individual team member should cover this general knowledge.

This implies that agile teams require more generalists than specialists. At the least, everybody should be able and willing to understand the big picture, and not find themselves interested solely in

digging in some specific details while ignoring the interests of the whole project.

So, as Don Wells said, in an agile project, you will find that

> "Everyone is of equal value to the project. [7]"

But this is only true if every team member bears responsibility for the whole project. Of course each team member will still have her individual capabilities and abilities, but they will all contribute equally to the team and the project.

Trust

It is natural for people to be skeptical of a change like switching to an agile process. The team members themselves, along with a lot of people only partially involved in the project, might not have trust in the success of this new process. The possibility that the team can change the process over time is often even more frightening than following a defined, but indigestible recipe.

The best argument against this mistrust is working software. Therefore, try to complete the first low-functional version of the software as early as possible. Another strategy for building trust is transparency. Make everything transparent for everybody involved in the project.

Different practices help to make things more transparent:

- **Shared Ownership**: Ask everybody on the team to take up responsibility for all kinds of things (for instance, the code, the process). This shows your trust in them.

[7] Don Wells, invited talk: *Transitioning to XP or Fanciful Opinions of Don Wells*, at the International Conference on eXtreme Programming and Agile Processes in Software-Engineering 2001, Sardinia, Italy.

- **Shared Knowledge**: This practice is often based on shared ownership. The knowledge about the information, —for example, the system— is transferred from one team member to another. This makes the system more transparent and understandable for everybody, and helps in turn to build confidence in the system.
- **Shared Skills**: The team has a variety of backgrounds and skills. This knowledge is accessible not only for the individuals, but for the whole team. Using a different process, the expert knowledge is often on the individual's guard against the whole team. Making knowledge transparent makes the team more trustworthy. Furthermore, it allows every team member to add new skills to their repertoire.

It is important that this transparency is always open and honest. Do not hide any negative information. Knowing about the bad things makes it easier to deal with them. Moreover, everybody should be invited to comment on the information and to help improve the situation. Thus, transparency includes controlling, auditing, and, most importantly, the customer.

Occasionally, when coaching a project, I find that project members assume that transparency stops right before the customer. For example, I sometimes have to lead long discussions in order to open the project's wiki web[8] for the customer, because the customer will then be aware of all insights of the project. Often, when asking for more transparency towards the customer site, project management tells me that it is afraid that the customer will recognize the problems inside the project. This is exactly the point! The customer should always be aware of the problems because it is her money the project is spending. These arguments are typical when discussing the impacts of having the customer on-site. As

[8] A Web based collaboration platform, which allows interactive communication and vivid documentation by editable html pages. Originally developed by Ward Cunningham. See Bo Leuf and Ward Cunningham, *The Wiki Way: Collaboration and Sharing on the Internet* (Reading, Mass.: Addison-Wesley, 2001).

soon as the customer becomes somewhat of an unofficial project member, the fear disappears from both sides: From the project's side because team members realize that the customer is a real person, and from the customer side because she understands the difficulties the project members are struggling with.

This reminds me of how I was before I started scuba diving: I liked swimming in the open sea, but I was always a bit afraid of the creatures underneath me, and I was pretty sure that sooner or later one of them would bite me. As soon as I started scuba diving, I did not even fear sharks or other predators. Being close to these creatures gave me the feeling of actually being a part of the living sea.

1.2 Team Building

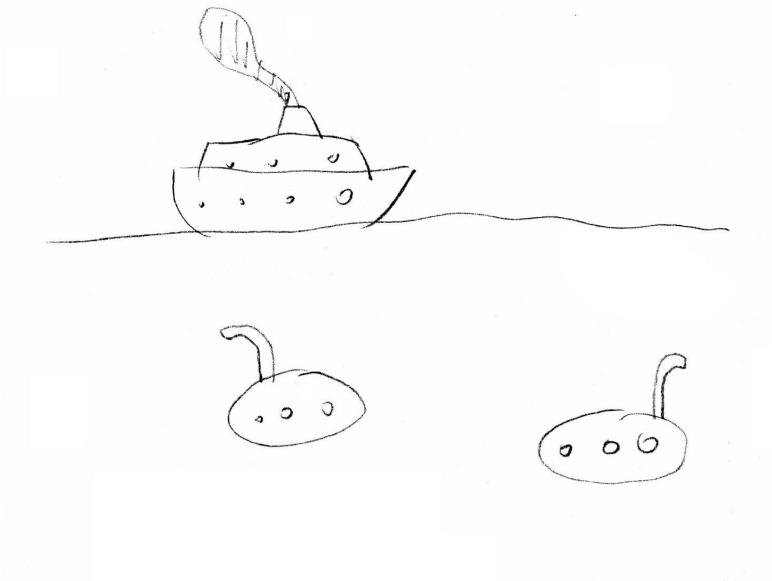
A large team is hardly manageable as a whole. Thus, in order to establish a flexible team, the team is usually divided into subteams of no more than ten members.

The typical structure used by large teams (and in large companies) is still based on Taylor's theory of building teams according to their knowledge. Therefore, you will often find an analysis team, a design team, a test team, and so on. The developers are typically further subgrouped into smaller subteams, each responsible for a specific function like presentation, database, network services, and the like. This tayloristic split is also known as *horizontal* team division. Taylorism works quite well for jobs that are repeatable. It doesn't work as well if a lot of creative and holistic thinking is required. You can furthermore consider defining *vertical* teams, which are focused around business functionality. These teams are also known as domain or feature teams, as Peter Coad terms them in the process, *Feature Driven Development*. On the other hand, if you are dividing the team vertically, you might find that not every team has all the necessary skills, or even worse, that every team might start to address the same problems.

Therefore, do not make this an *either-or* decision, but an *as-well-as* one. For example, if you start with a small team, and build slowly, you will come to the conclusion that on future projects your starting team should be staffed with people who not only have a good domain knowledge, but also a major technical background. This starting team most often defines the first architecture and verifies that the system can actually be built, and can furthermore serve as a model for the formation of the other teams. The horizontal and more technical focused teams should then support these new (vertical) subteams.

Building Teams and Subteams

As mentioned earlier, dividing the whole team in several subteams should not be a decision between a vertical *or* a horizontal division. Instead it should be an *as-well-as* decision, to provide a better mix of knowledge in the teams.



Subteams ...

Either virtual or real technical service teams[9] could be installed to further support those domain teams. For example, in one of the projects I worked for, we defined domain teams focusing on a specific (banking-) domain area, like one team focusing on accounting and another one on customer management. Each team had the knowledge needed to implement the features belonging to this domain, including the graphical user interface, the connection to the host, the business logic, and all the other required technology. If, for instance, the accounting team required some functionality from the customer management team in order to implement a feature, the accounting team just bilaterally discussed the requirements with the customer management team. The customer management team then in turn provided the required service within the development cycle.

[9] A virtual team is not apparently recognizable as a team. Be it, that the team members are not co-located and thus communicate via electronic media, or that the team members belong in fact to a different (real) team, and are just getting together every now and then for working on a specific task.

We established in this case not virtual, but real, technical service teams, responsible for supporting the domain teams by providing some base functionality. For example, we assembled an architecture team responsible for the business logic, and a presentation team for all graphical user interface aspects. Those technical service teams were requested to regularly visit all the domain teams. On request, members of a technical service team supported domain teams as regular team members for a specific amount of time.

Technical service teams should always regard themselves as a pure service provider for the domain teams. For instance, the technical service team responsible for building and supporting the architecture should always shape the architecture according the requests of the domain teams, not vice versa so that the domain teams have to use whatever the architecture team creates as it is often the case.

Depending on the actual size of your team, either you will establish virtual technical service teams, or you will establish real technical service teams. The members of the virtual teams are usually regular members of domain teams. In contrast, members of real teams usually lack a close connection to the domain teams. For this reason you have to ensure that real teams do not develop the *best* architecture, but the most adequate. You have to avoid that features are implemented just because somebody *believes* they are needed. Technical teams have to understand itself to be service teams, which deliver services to *their* customers, where their customers are the domain teams. The big advantage of this strategy is that the architecture only contains what is required. This makes the architecture much easier to maintain and, as a side effect, cheaper. Additionally, it eliminates the often-occurring social discrepancies between the technical and domain teams. One often gets the impression that those teams are working on different projects (not least from the way they talk about one another). Unfortunately, this impression is seldom wrong, and those teams have different objectives. Where technical teams' objective is to make use of a

specific technology and develop perfect frameworks not requested by the domain teams, the domain teams's goal is to implement the domain, not caring if they can profit and learn from one another (or from the frameworks the technical teams provide).

But how do the technical service teams know which service is required and, more importantly, which requested service has the highest priority? The team has to come up with a strategy, so not every requirement from each and every domain team will be implemented, because certain requirements might contradict each other. Or, worse, implementing these requirements will cost so many resources, that other teams will not be able to get their (more important) requirements done.

Therefore, like real customers, the domain teams have to speak with one voice. Retrospectives[10] could serve as a forum for deciding on new or changed requirements because all teams (or team leaders at least) are present, and the focus of the retrospective is the project's status and progress anyway. If one team states that it cannot proceed because it needs some special technical service, all teams can decide jointly if this is a requirement they support, so if approved, this will be a joint requirement for the technical service team. Otherwise, the requesting domain team has to implement the service on its own. These requirements are then scheduled in the same way as the domain teams schedule their requirements. Thus, the technical service team schedules requirements with the highest priority first, and does not schedule more than it can accomplish within the next development cycle. It might have to negotiate workload with the domain teams. It might happen that especially at the beginning of the project, the domain teams define many requirements for the technical service team, but there could be other times where there are so few requests, because for example the architecture can just be used as is. During "high season", you should ensure that the technical service team does not accept more work than it can accomplish. And in contrast during "low season",

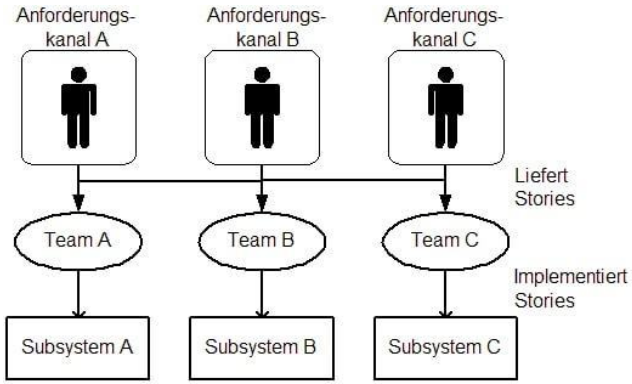
[10]Retrospective: Reflection at the end of a development cycle (see Chapter 4)

you should ask the members of the technical service team to join the domain teams, instead of implementing unnecessary additional features.

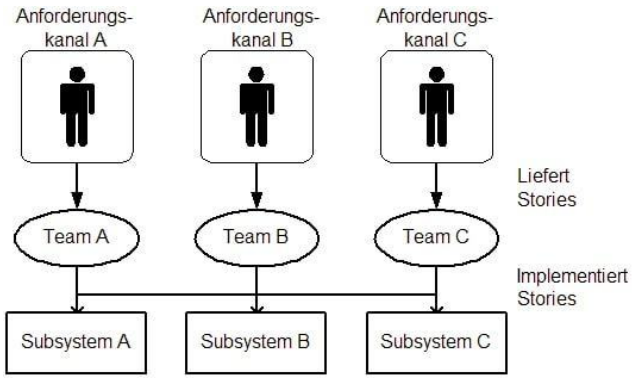
Requirements Channels by Stefan Roock

In this project we had to implement a system supporting multiple channels, which should address different user groups with various frontend technologies (desktop, web, laptop). Our starting project team consisted of five people from the development company and two consultants. With seven people, it was a size typical for an agile project. We had all the *Extreme Programming* practices in place when the project had to scale up and accept additional manpower—mainly developers. The goal was to have about twenty-five people in the project.

When scaling up, we had to address the issue of project structure. It became clear that it would not be possible to integrate all these people in one large team in the project. Therefore, we decided to split the project up into teams. But, we asked ourselves, what are the criteria for the division of teams? Do we use the architecture as the structuring mechanism and assign each subsystem to a team? Or do we assign each requirements channel to a team? In the first case, each requirements channel had to talk with every team. In the second case, each team had to modify classes all over the system. Since the planning games[a] seemed to be too complex in the former, we choose the latter.



One Team per Subsystem



One Team per Requirements Channel

We then got three teams for the three requirements channels, and a technology support team. The teams were rather small (four people), which supported taking over responsibility.

One thing we learned was that reorganizing teams takes more time than we thought. When we changed the organizational structure, the developers needed several weeks to get used to the new structure and get up to their development speed again.

Because teams were not assigned to subsystems, every developer was able to modify every part of the system. This was no

problem because the developers were able to master the code base (about twelve-hundred classes).

As time went by, additional developers joined the project and the code base grew. We ended up with about thirty developers with different programming skills. Now some developers weren't able to modify every part of the system without it breaking. Our first step was to tag core classes and the very complicated parts of the system as "expert code," which had to be modified by a so-called system expert.

That solved the problem, but it doesn't seem to be a very smart solution since there is no way to guarantee that only system experts modify the crucial part of the system. Currently, we are searching for better mechanisms for assigning code to teams. The main idea is to take the layering of subsystems into account. Some subsystems are specifically for a requirements channel and should be assigned to the relevant team. Other subsystems are relevant to several user groups and can't be assigned to one of the existing teams. These subsystems are assigned to a virtual "base subsystem" team, which is created on demand from the system experts sitting in the existing teams.

[a]The planning game is an *Extreme Programming* technique. The customers select and prioritize the tasks for the developers for the next development cycle and the developers estimate the effort for these tasks.

Team Roles

The idea is that a team has all the required knowledge. Thus, each team is a generalist on the domain covered. For instance, a domain team will be assembled by domain experts, graphical user interface developers, and database developers. But although the team consists of these different experts, this does not mean, that those experts will only ever work in their field of speciality. Instead

it is required that team members take different roles. For example, it is rather typical for the database developer to learn from the graphical user interface specialist how to build the presentation, and to then contribute to the user interface development. Thus, the goal of having generalists rather than specialists in a team is attainable by spreading the available knowledge.

The goal of this approach is not egalitarianism of all team members. Distinct skills and experiences are still necessary for specific tasks. However, the goal is to avoid the general tendency towards head monopolies and to spread knowledge and skills.

Additionally, each agile team also covers the required administrative knowledge necessary to perform, for example, integration and configuration management. The person who takes this role concentrates mainly on issues based on internal team integration and configuration, but will also be the contact person regarding this topic, for people external to the team. However, individual team members can have multiple roles: For instance, the person responsible for integration and configuration can be the domain expert as well.

It is very helpful to establish a team lead for every team. This person acts as a contact person for reaching the whole team. Often, the team lead coordinates who will attend a specific meeting (for example a retrospective).

Team Jelling

The goal is that the whole project team pulls together, that all team members communicate honestly and openly, and that everybody has the same big picture in mind. Or, as Tom DeMarco puts it, that the team *jells.*[11] The pulling together especially must be supported, so it becomes natural. As well as the more official aspects of project

[11] A team jells when it has a good chemistry, comparable to the one good jelly has. For more on this subject, see Tom DeMarco, *The Deadline. A Novel about Project Management.* (New York: Dorset House) 1997.

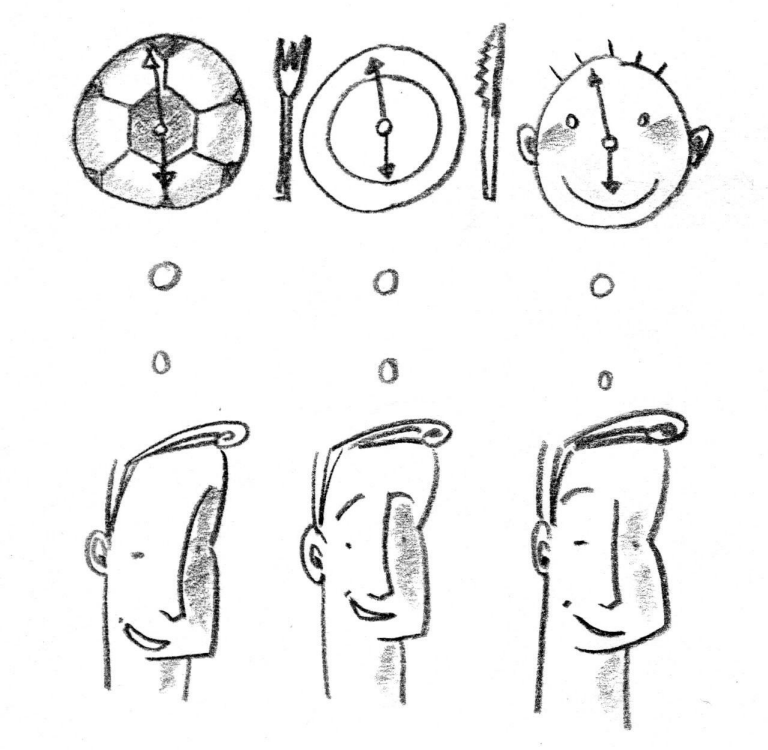
development, other, more enjoyable and motivational tactics must be employed to keep your project on track:

- **Food**: If you provide food, or just snacks—healthy or otherwise—the area where you placed the food will soon become an extremely popular part of the office. And when groups of people are there, taking advantage of the free food, they will start talking. You might also want to make use of team lunches. Although you should ensure that lunch time is also a break time, which allows the team members to relax and recover from their work. On the other hand, breaking bread together always helps people come closer together.
- **Party**: Organize a party once in a while— after the delivery of a major release, for example. This does not have to be something big. It would be enough to serve some sandwiches and beverages for a couple of hours or so. This will help people who wouldn't otherwise have the chance, to sit and talk to each other. Try to convince the company of the importance of such project parties, so it will approve them.
- **Recreation**: Organize some sort of recreational outing. It can be a sporting event, like a volleyball match, or some other social event, like bowling, go-kart racing, or something along those lines. Doing something as a group, this will help team members to get to know each other, especially when people are asked to team up with someone they do not work with regularly, and will hopefully reinforce respect and acceptance among all. Ensure that everybody can participate in the event, taking into account handicapped people, for example.
- **Project Identity**: Encourage the team members to instill a sense of project identity. Mary Lynn Manns and Linda Rising also stress the importance of having a group identity in *Introducing new Ideas into Organizations*[12] with a separate

[12]Mary Lynn Manns and Linda Rising, *Introducing new Ideas into Organizations.* (Reading, Mass.: Addison-Wesley, not yet released).

pattern, called *Group Identity*. Special tee-shirts, project-specific food and beverages or even project-specific phrases and slogans help to develop a project culture. In one of the projects I was working for we even came up with project cocktail. However, the project should not demarcate itself from the outside, instead it should be easier for newcomers to identify themselves with the project.

- **Regeneration**: Ensure that project members have time to regenerate. Even when under pressure to deliver, make sure that people are taking their vacations and that they are not working overtime. A project is better comparable to a marathon, not to a sprint.



Regeneration ...

- **Communicate and visualize results**: You cannot overestimate how motivational the growth of the system or the customer's feedback can be. Therefore, make sure everybody knows about the project's progress.

All the strategies suggested (which are just an excerpt of possibilities) reinforce communication and will ensure that your team members will get to know each other better and, more importantly, learn to respect one another. Try to ensure that members from different subteams interact with another. For example, if you organize a sporting activity, you can request that each side contain

no more than two people from each subteam. It is astonishing how much this contributes to a sense of a communal identity among team members, which usually results in the project running more smoothly.

Some strategies are not self-evident for a company. For instance organizing a party with temporal and financial support could be a problem. This is a sure sign that the importance of communication is still underestimated. Thus it will be necessary to convince the organization otherwise. It is worth the effort.

1.3 Interaction and Communication Structures

Communication is the most important factor in the success or failure of the whole project. Communication is difficult even when only a few people are involved, but it gets exponentially harder the more people there are involved. When setting up a communication structure for a large team, you have to consider the following constraints:

- **Direct communication** is the safest form of communication, and you know immediately if the receiver of your message understood what you said. However, the more people involved in a communication effort, the harder it is to get a message across. One reason for this is that there will not be enough time for everybody to actively participate in the conversation. Another reason is that typically only a few extroverts will participate, whereas all the introverts will accept the message, because they are uncomfortable discussing anything in big groups.
- **Different sensory modalities**: Every person obtains information differently. Some people, known as visuals, learn most effectively by watching; auditories, by listening; and kinesthetics, through action.

- **Overdose on communication media**: Additionally, it seems to be a law that as soon as a communication path works, it is abused until it does not work anymore. For example, if messages are transferred via e-mail, you will read your e-mails and respond to them. However, once your inbox begins to overflow with new e-mails when you get to work each morning, you are likely to either be very selective about which messages you read and respond to, or you will ignore them all. This, of course, is bound to eventually result in your getting in trouble for not reading an e-mail that the sender assumed you read.

Changing Communication Channels ...

Therefore you should also be agile and flexible with communication. Use various modes of communication, which address different persons differently respecting their different sensory modalities. Change the the communication channels from time to time. A manageable, average-size agile project will always require direct communication.

Open-plan Office

Ideally, the whole team sits in one room together with the customer. Because, as Craig Larman said in his book, *Applying UML and Patterns,* :

> "Having a team on another floor of the same building has as much impact as if it were in a completely separate geographical location. [13]"

However, in a large project with a team of a hundred or more, space constraints make it difficult to have everyone in one office. Open-plan offices are valuable in both creating space and enhancing communication. They can be created by removing cubicles, or rather positioning the cubicles around teams rather than individuals. Open-plan offices can sometimes accommodate forty to fifty people. So if you could have two or three such offices next to each other, project members would be sitting in as close quarters as possible.

Open-plan Offices by Nicolai M. Josuttis

I have no idea how others experienced this, but when I started my professional life open-plan offices had a bad reputation for me. They represented the idea of treating human beings like machines, which can be located close together in a big hall for saving money for the walls. And in fact, in a work life that assigns each employee a stupid and almost communication-free task, there is a lot to be said against putting all these people together in a huge room like in a laying battery. Especially, if the phone calls of the colleagues are nothing but annoying, and one has to fight tediously for each square meter of individuality.

[13]Craig Larman, *Applying UML and Patterns.* (Eaglewood Cliffs, New Jersey:Prentice Hall, 1998), p.448.

> Yet, since my first large agile project I look at open-plan offices from a different perspective. The circumstances are changing tremendously, if the job focuses on teamwork that enables several people to actually create something together. All of a sudden, moving to another room is painful, all of a sudden it is important to know what the colleagues are working on. All of a sudden it is important also to work physically *together*. The value that is created by this kind of communication, can't be estimated high enough.
>
> However, this does not mean that it is the best for an agile project to pack all project members
>
> together in a dreary open-plan office. Because even more important are some other things: For example, there is still the need to have a meeting without disturbing others. Adequate soundproofed meeting rooms are an obligation. Also individual workplaces are important for people, who need to think, design, or make a phone call untroubled in silence. I had the best experience with glass. Vitreous meeting rooms, individual vitreous workplaces, or vitreous dividing walls between teams allow the necessary transparency without raising the noise level to a degree that disables working seriously.
>
> In a sense, an agile working place has a spot of all, which is again typical for agility in general.

Always ensure that the individual subteams can sit together, even inside an open-plan office. Although this might seem like common sense, it is not as common a situation as it should be. Again, whatever your constraints, the distance between team members has a major influence on the success or failure of your project. Be aware that this distance does not necessarily have to be physical. For instance, if certain team members listen to music through headphones while they work, the headphones establish a distance between them and their peers. Therefore, the least you can do is to

try and make it possible for all members of the team to be on the same floor or, at the very least, in the same building. But everything that improves the seating situation pays off during development time.

Some people argue that the noise levels are too high in open-plan offices. This is not usually an issue. Mainly because everybody is concentrating too intensely on his or her work to be disturbed by the conversations of others. However, you may have some individuals in the team with particularly loud voices. In which case, you should ask them to lower their voice. If this is not possible, you should consider locating said individuals to a place where they will not disturb their peers. But this is a highly unlikely situation. As I said earlier, in my experience, the noise level on projects is almost always acceptable, and the advantages of the close proximity of team members far outweigh those of having a quiet environment.

Flexible Workplace

Nowadays, some companies do not support assigned office space. Instead, they use a system known as *flexible workplace* (also known as floating desks or desk sharing), where people just sit wherever they find some space. Team members either use cell phones, or have calls transferred to wherever they are sitting on a particular day. Typically, filing cabinets are mobile, so team members can have all their papers with them at all times. The underlying idea of the flexible workplace is that it requires less space than a more traditional, assigned-space system. The logic being that on any given day, certain employees will not be in the office: People call in sick, take vacations, make on-site visits to customers, and so on. Utilizing flexible workplaces, then, is a very efficient way to use office space and to save money on workplaces.

However, the catch is that you will never know for sure, where to find a specific person, which is an additional communication problem. Another problem is that at certain times (during the less

popular vacation months, for example), the risk is high that some people may spend their day wandering around looking for an empty space to sit. In some areas, this system is so well known, even people not working in the industry are aware of it. I remember traveling by taxi from the airport to a customer's office, and being asked by the taxi driver if she should speed up to make sure I would have a place to sit at the office. (It turned out not to be necessary, since we had plenty of time to spare; it was just eight o'clock in the morning.)

Another risk that teams utilizing flexible workplaces face is that people may get to work late, and not be able to sit with the other members of their team. At this moment, flexible workplaces are not that beneficial anymore. If you cannot avoid it officially, try to establish an acceptable working environment, for example by defining (flexible) team zones, within the constraints of your office's seating arrangement. Be aware that in your attempts to do so, you might get in trouble with the "office police." As the case may be, you have to fight this out, because as mentioned earlier, the importance of efficient communication can not be valued high enough. By the way, the philosophy of flexible workplaces creates an infrastructure that allows to relocate people and teams, which in turn can solve communication problems easily. However, if your company has the philosphy that every associate will have the same desk over many years, you might discover unbelievable resistance when adapting the workplaces flexibly to better support the project.

Encouraging Communication

The real difficulty of working with a large team is looking for ways to ensure efficient communication. I have found that the following steps are valuable in setting up a communication structure:

- All project members should sit as close together as possible without crowding each other.

- The retrospectives performed after each iteration and release cycle serve as a forum for direct communication. Typically, you will find that optimizing space, and therefore improving direct communication for the daily work, will be a regular topic until it is resolved.
- Regularly scheduled meetings for *all* project members are essential. Such meetings are primarily a mode of information transfer. In my experience, too many people attend these meetings for there to be any effective feedback or extensive discussions, but they work well for one-directional information transfer. Therefore, every project member should have the possibility to contribute—in the form of a lecture about a specific topic, for example. It is a good idea to announce the contents of the contributions in advance.
- Provide a *wiki*[14] on the intranet, not only as a means for documentation, but as a means for communication.

The philosophy of a wiki is to allow all kinds of discussion on the Web. Everybody has the right to make any changes to the web sites. This is possible through editable HTML pages. The wiki web only knows collective ownership, so everybody has the same responsibility for the contents. This helps to establish a community of trust. Furthermore, no deep knowledge of HTML is required to contribute to the wiki web. You can even contribute by writing plain text. If the wiki web is also used to document the project, you can be sure that this will always be a good source of project documentation.

- Establish different e-mail distribution lists that allow you to address everyone involved in the project, as well as specific groups of people.

[14]The term "wiki" comes from Hawaiian and means quick, which refers to the ability to make quick changes. For more information on wikis, see Bo Leuf and Ward Cunningham, *The Wiki Way: Collaboration and Sharing on the Internet* (Reading, Mass.: Addison-Wesley, 2001).

Communication Team

Be warned, however, that even making use of these different channels will not eliminate your communication problems. Another very effective way of improving your team's communication is to establish a separate (virtual) communication team . Depending on the size of your team, the communication team could consist of just one person. The communication team is responsible for visiting all the teams regularly, obtaining feedback, discovering deficiencies and (potential) problems (and maybe even solving them immediately). It is important that this happens proactively by approaching the project members. This way you will recognize problems earlier, than by waiting until they are reported officially or they escalate. Typical topics and tasks of the communication team are:

- **Unified project culture**: The goal is to establish a common culture with regards to such things as guidelines, tests, patterns, and the like.
- **Refactoring**: Uncovering sources for refactoring, not only improves the quality of the code, but also provides a learning opportunities for everybody.
- **Common understanding**: The communication team needs to ensure that all information, decisions, and announcements are understood by the teams.
- **Problem discovery and treatment**: Problems should be detected and at best solved immediately and in a simple manner. The communication team has the advantage of having an overview of all the teams. This way the communication team can establish contact or point to solutions other team might have found. If several teams have the same problems, strategies are required which will solve these problems generally (extending / adapting the framework, or providing patterns for the solution). Furthermore, the communication

team suggests how the process could help to overcome or eliminate the encountered problems.

The members of the communication team should never act as supervisors or controllers, but instead more like a team of ombudsmen. These ombudsmen should be sensitive to the hopes and fears of the individual team members, and should collect suggestions for process improvements. For example, ensuring that the team understands all decisions enables them to either accept these decisions or to come up with a suggestion that supports them better.

It is very important that the members of the communication team have a good overview, are well trusted people with good communication skills, and who are widely accepted and respected by the rest of the project team. These people should be able to take matters into their own hands and who are able to manage the project as a whole, but have also good connections to the individual persons. In smaller teams, this will often be one person only, whose tasks cross boundaries, like running reviews, retrospectives, or coach the process. In larger teams (with at least more than 50 people) this will always be a fulltime job for one or even more persons.

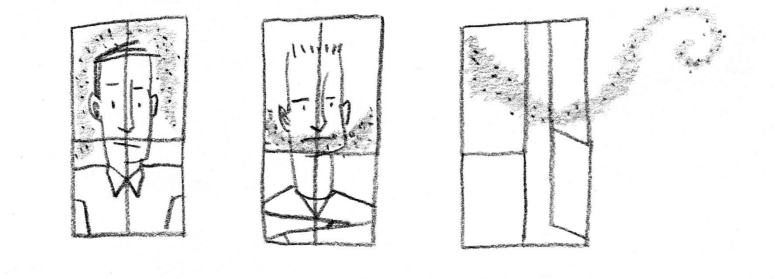
You will hardly find a project organization that is aware of the necessity of this role. This makes it difficult to establish this kind of role. I often call these people communication manager or simply catalysts. Ideally these are persons, of who Tom DeMarco and Timothy Lister[15] write that their sole presence is enough for a project to run smoothly.

1.4 Trouble shooting

Sometimes you need to act quickly; if, for example, one of the teams is completely under stress, one team stops either talking to another

[15]Tom DeMarco and Timothy Lister, *Peopleware: Productive Projects and Teams* (New York: Dorset House, 1987)

team, or two teams start continuously blaming one another and are not able to work together anymore. In such situations, you face two difficult tasks: One is to look at the problem and see exactly what kind of problem it is, and the other task is to solve the problem.



Smells ...

The first task is more difficult, because it depends a lot on the team's culture. Here are some typical problem signs:

- **Cynicism and sarcasm**: Humor is a sign that everything is right on track and that people are having fun doing their jobs. But if the humor turns into sarcasm, this is a clear sign that the team does not jell and does not believe in what it is doing.
- **Blame**: This sign is much more obvious, and therefore easier to spot and tackle. The teams or people blaming each other usually have problems respecting and understanding each other. Although blame can sometimes be a sign of a difficulty communicating.
- **Lack of Feedback**: This is often a sign that the people have given up. They do not believe in reaching the goal and they do not believe that anybody has an interest in their opinion or in their effort.

Whatever the reason is, you can neither accept nor ignore the situation. All these circumstances will slow down the project's progress significantly. Therefore:

- If a team is under stress, and complains that it cannot get its work done because there are too many meetings or its time is spent supporting other teams, protect the team for a couple of hours each day by arranging *quiet times*. It might be necessary to arrange an office-wide quiet time, either temporarily or permanently. For more on quiet times, see Alistair Cockburn's *Agile Software Development,* in which the author suggests defining the period between ten am and twelve pm as quiet time, during which no phone calls or meetings are allowed.[16]

  If instituting quiet hours is not sufficient to bring the team back on track, a more rigorous approach is required: Instead of quiet hours, make sure the team will get one or two quiet weeks, with one or two hours of each day as "regular office hours," so that team members can still process incoming requests.

  The most extreme solution is to allow the team to go to a closed meeting for a couple of days. This is also the most effective and probably the most expensive solution. It is often used in different circumstances: if, for example, the team does not jell, or has to consider different kinds of solutions. The closed meeting is more often used as an environment for the project kick-off (for making teams jell) and for the project post mortems. [17]

Quiet times have a tradeoff: They can also lead to a complete lack of communication, and should therefore be carefully balanced.

- If two teams stop talking or working together efficiently, locate them next to another. This way each team will recognize why the other team acts like it does and they will start to respect one another.

[16] Alistair Cockburn, *Agile Software Development.* (Reading, Mass.: Addison-Wesley, 2002).

[17] For more on postmortems, see Norm Kerth *Project Retrospectives* (New York: Dorset House Publishing, 2001)

Another strategy is to set up a voluntary exchange program among teams, so that one member from every team switches place with a member of another team.[18]

Both strategies help to improve the understanding for the other team.

- If a meeting culture evolves, where people have to spend more time in meetings than they do working, and if people start complaining about unnecessary meetings, question all meetings that have been established, especially all regular meetings. Furthermore you should question if all the participants are required to attend in order for the meeting to be a success.

  Generally you should introduce the "law of the two feet", as described by the Open Space Technology: anybody who feels that the meeting is a waste of time is allowed to leave. This might require some sensitivity from the organizer of the meeting: If a participant does not contribute, he or she should be politely invited to contribute to the project's success outside the meeting.

  Another possibility for overcoming the meeting culture is by introducing quiet times.
- Finally you can at the beginning of each meeting, ask one of the participants participant to excuse him or herself from the meeting to do something more important (this was first suggested by Tom DeMarco, in *The Deadline*[19]). Take care that the person differs each time, and is not the most junior person.
- If a team is not very well integrated—if, for example, it is often not well informed or is often blamed by other teams for incidents that stemmed from a lack of information— then locate food in the team's area. Normally it is only a matter of hours before other teams find themselves in the food area

[18]Thanks to Mike Cohn for sharing this approach.

[19]Tom DeMarco, *The Deadline – A Novel about Project Management* (New York: Dorset House Publishing, 1997)

and the communication or information flow is reestablished naturally.

1.5 Virtual Teams

According to researchers in the area of work life, the future will be virtual teams. Fewer and fewer real teams will physically come together to work on a project, and more and more teams will be assembled over the internet. This has a lot of advantages:

- Each individual project member is responsible for his or her own work space and environment. Although sometimes the client will provide the equipment, most often the individual project member will have to use his or her own hardware. This saves the client a lot of money.
- You have much better access to different skills. You are not limited to people from your region or your company.
- You are not responsible for the team in the long run. You only have to pay them for as long as they work for you. You are under no obligation to find them their next job.
- If the team is distributed all over the world, another advantage is that at any time of the day at least one team member is most likely working on the project. There is hardly any project-off time.

The main problem with virtual teams, is that they lack the most efficient mode of communication—direct communication. As Erran Carmel and Ritu Agarwal said in their article *Tactical Approaches for Alleviating Distance in Global Software Development* in IEEE Software,

> "Distance negatively affects communication, which in turn reduces coordination effectiveness.[20]"

[20]Erran Carmel and Ritu Agarwal, *Tactical Approaches for Alleviating Distance in Global Software Development*. IEEE Software, March/April, 2001, Vol. 18, No.2, pp 22-29, p.23

In virtual teams not only the communication with the customer but also inside the team is a problem. This way it's very difficult for a virtual team to get a common understanding and to pull together.

Distributed Teams

Large teams are always distributed in one way or another, just because they are too large to contain in one room. However, distributed development is somewhat more extreme, in that the project members are distributed over several sites and, as the term suggests, the project is, as such, developed in a dispersed manner. Sometimes you might find that there is a single team spread over several sites, whereas at other times there might be several teams, each located at a different site. Outsourcing (see Chapter 6) is one example of a distributed team.

One problem in this setting is ensuring that everybody on the team pulls together. On projects like this, you will often find that people blame one another, mainly because they do not know each other, and therefore do not trust each other. Also, technical topics like version and configuration management are even more complicated in distributed teams. Of course, there are tools that can help these more complex areas of development, but they do not make up for the inconvenience and problems caused by distributed development.

If you must have distributed teams, the Internet is likely to be your main form of communication (e-mail, wiki web, chat rooms), and video conferencing is also a good way to communicate. However, be sure that people working out of different locations are able to meet with each other, at least occasionally. Communicating through the Internet will only work efficiently if people know and trust each other—and there is no better way of building trust than through personal contact.

Distributed Teams, by David Hussman

Most agile practices ask project members to keep communication channels open and filled with honest dialog, without regard to the message content. Nowhere is this more important than on an agile project with distributed teams. Along with the usual technical challenges of distributed development, agile development brings even more challenges, mostly aimed at those outside the development teams.

Project managers, coaches, and customers need to be vigilant when it comes to listening to, and addressing, the developer's concerns; tracking successes and failures; the way in which story content[a] is gathered, organized, and presented; and the consistency of the process and the development environments. With distributed teams, the need to embrace change and make the necessary corrections to the direction of the project sooner than later is even greater. Just as the last car of a long train starts moving long after the first, so too does change take longer to move through the distributed teams.

The following list of best and worst practices might help those outside the development group find and address needed changes before the project strays to far from the correct path. Many of the listed items apply to any scaled, agile project, but their importance is heightened when the development teams are distributed:

Best Practices

- **Customer Agility**: Ensure the customer teams can make any necessary changes, by keeping the ratio of customers to developers as low as possible (1 customer to 3 or less developers; the larger the project, the smaller the ratio). The ability of the customer team to react and change direction can be difficult when stories cross team boundaries and teams struggle to bring portions of a software solution to fruition. For example, if five people

are writing a book together with each person working on a different chapter, the lucidity and cohesiveness of the book is proportional to the amount of time the authors spend discussing the book with each other.

- **Group Speak**: The more often project managers, coaches, and trackers discuss planning and development issues, the better. Although this discussion can take place over e-mail, at a wiki site, and through non-verbal channels, ensure that at some point there are conference calls or video conferences. If possible, have a different team lead the discussion every call (this helps all involved to embrace the project). Also, as teams grow and change, each group starts to form its own view of the project picture. Sharing your team's lingo with other teams can aid in maintaining a common view and provide cross-team insight.
- **Common Acceptance**: If possible, ensure that each team is building and testing against a common set of hardware and software (if this is not possible, discuss the environmental differences regularly). The closer the environments, the better the level of acceptance testing. Try utilizing a common acceptance testing strategy, and whenever possible share the data sets used for testing. Try to start automated acceptance testing early, and run the tests as often as possible.
- **Developer Rotation**: Often times, teams in the same building feel as disconnected from each other as teams separated by states or countries. If possible, move players from team to team. If the players cannot colocate, explore the notion of rotating the work done by the teams instead of the team members themselves. This may seem like an unrealistic suggestion, but it may be a way to help maintain a healthy project whose members have a holistic perspective.

Worst Practices

- **Technical Stratification**: It is often a natural fit for distributed teams to work on subjects that they are familiar with (this is often why distributed teams are brought into a project). If at all possible, avoid splitting development tasks across technical boundaries. Try instead to plan and develop toward functional goals and subdivide the work from that perspective (this will avoid the classic producer–consumer relationship between teams, where one team has finished all its work and is left with nothing to do).
- **Failure to Communicate Mock Implementations**: As teams may not have everything that they need in order to complete a particular feature, they may choose to mock / stub out[b] temporary solutions. This is fine (and, if everyone is using the same code base and acceptance tests, it can be quite clear), but make sure that the interfaces to the mocks / stubs are agreed upon by all.
- **Loss of Iteration Synchronization**: In most cases, it is best to keep all teams on similar iteration boundaries. It may be that the teams (and more importantly, those in charge of the planning) find a steady state with varied iteration boundaries. In either case, ensure that the iteration synchronization is as constant as possible. Again, as with scaled, agile projects, if a team has an iteration schedule slip, even of just a couple of days, planning issues may arise that are best dealt with by moving the incomplete features to the next iteration. The difference may seem small, but the consistency will benefit the project plan.

Conclusion

Distributed teams using agile practices face many of the same problems that subteams face when agile projects scale to large numbers. The need for vigilant and constant communication is exacerbated. Small issues can quickly grow large or span several iterations if not known to all team members. One

team's frustration can affect other teams without their even knowing it. Strive to keep process, schedules, and environments constant, and when they change (as we know they will), notify all involved as soon as possible. The more that can be shared between teams (source code, data, and so on) the easier this task will be. Ensure that planners listen to the developers, and help them listen to one another.

[a] *Extreme Programming* uses the term stories for requirements, which are defined by the customer for a release cycle. User stories are comparable to Use-Cases in UML.

[b] Mock- and stub objects allow to test partial solutions by simulating the missing code.

Open Source

Open source projects are well functioning examples of virtual teams. Possible reasons for their success are

- All the team members are very idealistic. There is no need to motivate them or try to ensure that they identify themselves with the project. This comes all naturally.
- Everybody feels responsible for the whole project, and takes this responsibility very seriously.
- There is a broad community that provides immediate feedback. This feedback is what drives the whole project. There is no difference in the value of feedback whether it comes from peers or from users.
- Everybody who contributes to the project takes pride in his or her work.

The main underlying principle of open source projects, and the reason for their success, is the gift economy (the culture of giving away capacity and information). This means that everybody working on

an open source project is doing so voluntarily. A lot can be learned from this approach, especially for use on commercial projects. Mary Poppendieck once reported that a new project manager asked her for advice on becoming a successful team leader. She asked him if he had led ever a team of volunteers (of any sort). He replied by saying that, he was a successful choir leader. Poppendieck continues,

> "I suggested that if he used the same techniques with his project team that he did with his choir, he would be a successful project manager. He said it was the best advice he ever received, and he blossomed as a project manager. [21]"

Open Source by Dierk König

The open source movement derives its name from the practice of sharing the source code of a valuable product among an arbitrary number of developers. This so-called collective code ownership means that the code belongs to all these developers. They are entitled to change it and are all responsible for the final result.

In the context of this book, open source projects may be of interest because they share properties of both large and agile projects. They make use of agile practices while suffering from the same problems that large projects have in regards to team distribution.

I'm not so bold as to claim that open source produces better results in general. Sure, lots of open source products are widely known for having excellent quality with zero costs to the user. But I'm the first to admit that there are also numerous sloppy projects out there that will never produce anything useful. However, maybe we can take something from the successful

[21] Mary Poppendieck in a private e-mail correspondence on the gift economy.

ones!

Distribution

It is evident that the physical co-location of all contributors to an open source project is impossible. However, we have observed that members of the core editing team of an open source project sometimes get together to tackle a special issue, often sacrificing personal time and money. The collaboration then looks like an *Extreme Programming* pair programming[a] session.

Even the users of open source software hold events so they can get together and share their experience. The Eclipse code camp[b] is one example.

If the open source people put so much effort into overcoming the obstacle of distribution, can we—in paid time— go upstairs to pair up with the database guy?

Idiosyncrasies

An open source contributor is not forced into anything. If you do not like anything about the project (the setup, the code style, the technology, the people, and so on), you can leave at any time or even *fork* (make a new project based on the old one).

This, and the fact that a lot of people write contributions in their spare time, leads to a project staff that likes the applied work style, or at the very least accepts it. The number of complaints is noticeably small.

One could claim that open source projects are not limited in time, scope, or resources, and therefore do not need the measures of control that are applied to in-house projects. This is not really true. Running an open source project in your spare time, knowing you have only a few hours a week to work on it makes you think hard about what to implement next.

Open source is opportunity-driven. Whoever needs a feature the most will implement it and submit the contribution. Noth-

ing is produced for the shelf. Contributors undertake their the tasks without anyone telling them to.

Developers know that their code will be read, literally, hundreds of times. This is motivation enough for them to achieve high code quality, and it is a good opportunity to show off their professional expertise.

Now, without any imposed order, programmers do what they think is appropriate; surprisingly, this does not result in total chaos, but rather in automated testing techniques, stable frequent builds, ubiquitous version control, flexible architectures, and self-documenting code. Most astonishing is that these programmers manage to achieve something that most organizations do not: mutual respect among team members.

Architecture

Open source projects typically do not start with an up-front architecture (Eclipse may be an exception), but they always have one in the end. The opportunity-driven nature and the resource constraints of open source force contributors to practice reuse. This is especially apparent in the Jakarta project family[c]. Every project is built on other projects, which were built on projects that came before them, and so on.

The new challenge is to manage project dependencies, a well-known problem that most big organizations struggle with. Open source offers an easy yet powerful alternative: Let the user decide.

Another idiosyncrasy of open source architectures is the focus on extension points and pluggability. JUnit[d], ANT[e], and eclipse are perfect examples of this approach.

Just think of the effect that applying the principles of open source software would have on your corporate IT projects.

Project Structure

In open source projects you typically see a core team of

editors with write access to the repository. The requirements for becoming an editor differ on every project. Some do not have any restrictions, while others only grant write access to contributors that consistently submit quality work. Some projects have a fixed group of editors.

The core team decides whether contributions from the outside get incorporated or rejected.

Communication channels are highly self-organized. The flow of information typically takes place on mailing lists.

Project Setup

For open source projects, having a self-contained build is absolutely crucial. It is no wonder that open source projects were the pioneers of build automation. The same holds true for the use of versioning systems, nightly builds, and automated self-testing.

As the source code is highly visible, a new degree of rigor is applied to the end result. The source code is subject to excruciating review and refinement. Its compliance to every standard in use will be checked for all platforms the community uses. The contributor has total control over how to achieve this result. Assessing results rigorously but giving developers freedom to use their own work style is a strong agile move that large projects can follow as well.

Documentation

Typically, in open source software there is not a lot of external documentation. The code must speak for itself. Although it may seem unusual, this strategy works well on open source projects, where the code, especially the test code, must reveal its intention.

Successful projects are often accompanied by articles and even books. The usage documentation that comes with the distribution typically contains only very basic help for the beginner.

It appears that any necessary external documentation gets written on request. For example in Canoo WebTest[f] we had a fairly complicated security use case. A user volunteered to write the documentation for this, provided that someone helped him figure out what to do.

The result seems to be a good medium between the two extremes of lacking all the necessary information and having excessive documentation in which the important information is hidden among pages and pages of unnecessary text.

Planning

Frankly, there is no long-term planning whatsoever. There are exceptions, but for most open source projects, there is only a little short-term planning. The reason for this is to save the developers unnecessary work, by not prescribing long-term directions of any greater detail than a common vision. Directions are derived solely from user feedback.

The most challenging part of adopting open source strategies in corporate IT projects may be trusting in the evolution and refinement of this complex adaptive system that is beyond managerial control.

[a] In *Extreme Programming* exists the rule that all productive code is written by a pair of developers. Thus, a continuous review takes place throughout the development by two developers solving a problem jointly.

[b] The development environment Eclipse (http://www.eclipse.org) is an open source project as well. The developers of Eclipse meat occasionally in so called *code camps* for exchanging experiences and developing the product further.

[c] Jakarta on the web: http://www.jakarta.org

[d] JUnit on the web: http://www.junit.org

[e] ANT on the web: http://www.ant.org

[f] WebTest on the web: http://www.webtest.canoo.com

1.6 Summary

You should never underestimate the value of face-to-face communication, especially with large and distributed teams. You should do everything you can to provide as many opportunities as possible for direct communication. Direct communication, together with transparency, helps to build trust both inside the team and between the team and the outside world. If team members trust one another, they will not fear taking up responsibility.

A tool like the wiki web will help you build trust by empowering everybody through shared ownership. In doing so, such tools change the flow of communication from control-driven to collaborative-driven.

When building subteams, you should ensure that the skills within each subteam are mixed. Furthermore, the technical service teams should serve the domain teams If so desired, they can also be established as virtual teams.

Last but not least, allow the team to have fun. This not only helps teams jell, but more importantly, it also makes the work environment a pleasant one, which will make everybody want to work and succeed with the team. Consider the values of the gift economy like open source does, it will also help to make work much more enjoyable.