1

INTRODUCTION

Big stone is hard to throw.
–German proverb

Imagine you have to develop an application that will support traders in the front office of a financial institution. The traders have to be able to buy and sell products and calculate the risk of each transaction. One major difficulty is that the products available for trading are constantly changing. If you focus on the options market, you will see that almost every other day there is a new product (option) available to trade, which is traded differently than the others, and whose underlying risk must also be treated differently. Therefore, if you ask your client for the system requirements today, you will probably get a different answer than if you were to ask her tomorrow. Furthermore, she will let you know that without the application she is unable to trade these new options. With each day that passes without her being able to use your application, her company loses several hundred thousand dollars. As if that weren't bad enough, she also points out that the competition is already able to trade these options. She assures you that any kind of support your application can provide for these products would help to reduce the loss, because she can perform some of the steps manually. So she does not

insist on having full trading support for the new products, although it would definitely be a plus.

A similar situation could occur in the telecommunications sector, or in any domain with a focus on e-business. The big difference between these more modern applications and more traditional applications is that the modern ones must be available in the market quickly. Otherwise, the application could already be obsolete by the time it is first used, or the company might be run out of business. Sometimes it is more important to serve the customer's *basic* needs *quickly,* than to fulfill *all* her requirements *later,* which might end up being too late.

Heavy-weight processes of the 1980's and 1990's have difficulties dealing with these new requirements. They have instead occasionally been successful in domains with *stable* requirements. In these domains, everything can be formalized, and a detailed plan can be set up at the very beginning. Furthermore, every project can "blindly" follow this plan without needing to worry about updating or modifying it. Examples of this are defense projects, or projects from the airline or nuclear power plant industries. As well as stable requirements, these projects often seem to have limitless cost and time budgets. Because of this, it is more important to fulfill *all* the requirements than to deliver a subset of them on time and in budget. However, this objective is also changing in these domains. For instance, in the defense sector processes that support changing requirements are becoming increasingly important.

Agile processes promise to react flexibly to these continuously changing requirements. That is why agile processes are currently treated as a panacea for successful software development. However, agile processes are almost always recommended for small projects and small teams only—bad news for those large teams that have to deal with speedy changes of the requirements.

That is the reason why this book deals with agile processes in large projects. But before we discuss this topic in detail, I would like to further define the focus and the target audience of this book. At first it is necessary to explain the terms *large, agile* and *agile process* and the context in which they are used.

Questioning Scaling Agile Processes

Software engineers tend to question the feasibility of agile software development in the large, not only because most agile processes claim to work mainly for small teams, but also because most projects that fail are really large ones. The reason most (large) projects fail is a lack of communication: among teammates, between team and manager, between team and customer, and so on. Communication is one of the focal points of agile processes. But can effective communication ever be established successfully for large teams? The popular opinion is that it can't; leading to the idea that if you have a hundred people on a development team, get rid of at least eighty of them and keep the top twenty or (preferably) fewer, and the chances for project success will rise significantly.
However, you can't generally avoid large projects. Sometimes you will face constraints that force you to run a large project with a large team. For instance there are projects, which have such a large scope that it is not possible to realize it with a small team in the defined timeframe.

If you want to take advantage of agile processes, several questions arise: Are agile processes able to scale, that is can they be amplified in order to support large projects? And, moreover, are they able to support large projects? And what kind of problems occur when an enterprise decides to use an agile process for a large, perhaps even mission-critical, project? This book tries to answer these and many questions relating to agile software development. But, before we go into more detail, I should better clarify what I mean by *large* projects.

Examining Largeness

In my experience, I have found that a project can be considered large in many dimensions. For example money, scope, the amount of people involved and the risks can be large. These different "dimensions" of largeness are mostly interrelated. Some dimensions exist as a first-order consequence of the requirements and constraints. Others are derived from these first-order dimensions.
The individual dimensions of largeness and their interrelations are defined as follows:

- **Scope –** Scope is a first-order dimension of largeness, created by the amount and complexity of the requirements. If a project is large in scope, you can either address that issue by allowing a large timeframe, making the project large in the sense of the

time it requires. The other possibility would be to allocate a large staff to the project. Consequently the dimension scope influences the dimensions time and people.

- **Time –** Time is rarely considered a first-order dimension in software development. I mean, I have never encountered a company that decided to work on a project over 20 years, just to kill time. So this is never the reason for starting a project. However, it is typically a dimension that follows another dimension. For example, if the risk of the project is high, because you have a number of unskilled people on your staff, you will need to have them trained, which will take time. Some projects can even go on for ever, because nobody has the courage to cancel them.
- **Money –** Money is also typically a second-order dimension. This means high costs are always a consequence of the growths of some other dimensions. At least, I have never seen a project that was started just because there was a lot of spare cash lying around. On the other hand, I have seen a lot of projects waste enormous amounts of money without batting an eye. But this was always a consequence of one of the other dimensions. For example, a large team could cost a lot of money, but the question of whether it is necessary to have such a large team is rarely raised.
- **People –** This is a different matter. The amount of project members is usually a first-order dimension. It is possible for the size of a project's staff to be a side effect of the scope of the project. However, sometimes projects are staffed with a lot of people—in the worst case, right from the beginning—mainly to show the importance of the project, or of the project management. The amount of project members is not related to the amount of developers only, but also for example to the amount of customers. The more customers are involved in the project, the higher the risk of contradictory requirements.
- **Risk –** Risk is a much more complicated dimension because it can refer to almost anything. For example, team size can be a risk, but focusing on a hot technology also carries a big risk and is often followed by having to spend money to train the staff, among other things. However, risk is typically a second-order dimension.

Therefore, the two initial reasons for scaling a project are scope and people. You can definitely run a large-scope project with a small team. But large-scope projects are almost always developed by a large team—especially in large companies.

Typically, if a project is large in terms of people, all its other dimensions are probably just as large. For example, you will hardly ever find a large team working on a project with a narrow scope, a schedule of only three months, or a budget of only a few hundred thousand dollars.

The project itself might not carry any extraordinary risk, but scaling all the project's dimensions implies a risk of its own. For instance, if a lot of money is involved, there is a high risk that a lot of money will be lost. Or, if the timeframe is extremely large, the risk that the project will never be finished at all increases.

In this book, I focus on projects with large teams. However, due to the fact that large teams usually scale also the dimensions scope, time, money and risk all these other dimensions will not be ignored.

Raising Large Issues

Of course, *large* is no well-defined magnitude, and neither is the largeness of a team. Will a team considered to be large if it contains 2, 10, 100, 1000 or even more people? And what impact does every additional order of magnitude to the staff number have on the process? For example, let's look at its influence on communication:

- **2 people and more:** If a project is developed by only one person, that person has (hopefully) the big picture of the project in mind. He or she knows the whole system in terms of code and design. As soon as another person is added to the project, these two people will have to communicate with each other. Communication is the only thing that will enable both developers to understand what is going on and to further coordinate their efforts. For example, it would be annoying if they were to both work on the same programming task unknowingly, only to find out once they began to integrate the code.
- **10 people or more:** With teams of this size, you have to start coordinating members' efforts and their communication. You have to explicitly establish communication channels in order to discuss topics with the whole group.
- **100 people or more:** Even if you have an open-plan office available, teams of this size will not fit in a single room. Therefore, across the entire team, you have to strategically foster the "natural" communication, that would take place inside a single room.
- **1000 people or more:** Chances are high that this team will not only be distributed over several rooms, but also over several buildings, perhaps over several different locations. Consequently, the people on the team are unlikely to know all their teammates.

This example shows not only that large is relative, but also that scaling can lead to different consequences.

Specifying the Projects in Focus

This book is based on my experience with projects with teams ranging in size from one person to 200 people. I learned a lot about scaling agile processes while working with these different-sized teams. In my experience, you will recognize already significant consequences with a team of twenty or more. So, although this book deals mainly with issues faced by teams with over a hundred members, those projects with even more than ten people will also benefit from this book, especially if they are embedded in a large organization. Due to the lack or my own experience I do not examine the special aspects of teams with 1,000 people or more. However I assume that also in these circumstances exist issues and challenges , which are addressed by this book. This book helps to understand the agile value system and shows a way how to preserve these values even with large projects. This clarifies also the difference between the agile value system and its realization in a specific process, like for example *Extreme Programming*.

My experience was mainly with co-located teams that outsourced only minor parts of their development effort. This means that dispersed development is not a focal topic of this book, although it is discussed in Sections 4.3 and 6.9.2.
The projects I worked on were varied in their nature. I worked with teams in the financial sector, the automobile industry, telecommunications, and the software industry.
Of course I exchanged my experiences with a lot of other people, most of whom had similar experiences to mine, in terms of the largest teams we had worked with. Some of them have experience with teams of 350 people and still encountered similar challenges.

Therefore, all those issues and suggestions pointed out in this book are based on experiences with large teams and large projects—either my own or those of colleagues of mine.

Detecting the Agile Method for Scaling

This book neither presents agile processes in general, nor does it present any agile methodology in particular. (However at the beginning of the next chapter, I provide a very

brief introduction to the fundamentals of agile processes.[1]) So, although you might, for example, detect some techniques that remind you of Extreme Programming, neither the title nor the focus of this book is *Scaling Extreme Programming*. But still, it is possible to scale some of the practices of Extreme Programming, so they are therefore beneficial to large teams. And, in parallel, they support the underlying value system of agile processes.

As we shall discuss later (see Section 3) a large team is typically split into many smaller teams. Because a lot has been said already about agile processes in small teams, I do not focus on the processes these subteams are using. Instead, I concentrate on the process that brings them all together and enables them—despite the large number of people— to work together agilely. Therefore, rather than focus on every aspect of agile processes, I concentrate only on those that work differently in large projects developed by large teams.

The problem is that processes, also agile processes, do not scale linearly because, depending on the "jump" in size, completely new difficulties might occur with the increased team size. The differences are rooted in the fact that some parts of the process cannot be done well by large teams, and require a specific treatment. Other differences are the problems that arise solely in large teams, such as communication, as we have seen before.

Thus, instead of scaling a particular agile method, this book presents best practices that allow us to scale up the agile principles by respecting the agile value system.

Identifying the Reader

The book is aimed at *change agents:* the people who want to create and establish an agile process despite the difficulties of a large team. Change agents in small projects in a non-agile environment will also benefit from the practices presented in some of the chapters—Chapter 6, “Agilitiy and the Company”, in particular. I assume that the change agent already has some familiarity with agile processes in general or with a particular process (Extreme Programming, for example). Moreover, this book will be definitely of interest for people, who:

[1] If you are looking for in-depth information about agile processes, I suggest you read Alistair Cockburn’s *Agile Software Development* first. Alistair Cockburn, *Agile Software Development.* (Reading, Mass.: Addison Wesley, 2002).

- have tried and failed to use agile methodologies in large projects
- have succeeded in using agile methodologies in large projects
- have not tried agile methodologies in large projects but would like to do so.
- are firm believers of the linear (waterfall-) model and think agile processes do not work (especially on large projects).
- are firm believers of agile processes, but who think they would never work on large projects.

So, as a change agent, you are probably working on a large team as a project manager, a process coach, a consultant, or a developer. You would like to use an agile process for the large project you are working on, but are unsure how.

Revealing the Structure of the Book

The book has the following structure:

- Chapter 2, "Agility and Largeness", examines the principles and value system of agile processes, raising the difficult question of how they affect large teams.
- The focus of chapter 3, "Agility and Large Teams", is the impact on a team of a switch to an agile process. How does it affect the team members, and how can you allocate the roles and create the subteams necessary to make it all work? Also in this chapter, we look at virtual teams, using distributed teams and the open source community as examples.
- Chapter 4, "Agility and the Process", concentrates on the characteristics of the process that will allow you to coordinate several subteams without regimenting them. The goal is for all the different teams to pull together by remaining agile in their activities.
- In Chapter 5, "Agility and Technology", we look at how the size of the project and the team influence the underlying architecture. We examine the role of the architectural lead and how the architecture can provide a service for the team. We also discuss some techniques and good practices that help to define an agile system.
- Typically, large projects are run by large companies. And large companies bring their own burden to a project. Chapter 6, "Agility and the Company", deals with the problems a big enterprise loads on an agile project.
- Chapter 7, "Putting it all together: A Project Report", presents a concrete, coherent experience report, which gives an account of a large agile project.