Lecture 6 (lecture in detail)

In part B, we reviewed the principles that define the fundamentals of the Agile approach.

But of course, principles are only as good as their application. And Agile methods defined beyond principles a number of practices which describe regular, almost ritual activities, that you have to practice in order to make the principles real.

These practices are the subject of the five segments of this lecture. Unlike the principles lecture, where we started with managerial principles, and then went on to technical principles. In this case, were going to go a bit back and forth between more management oriented stuff and more technical stuff.

Slide 2

So we start with **meetings**. At the core of Agile practices lie a number of regular ritual meetings, beginning with the famous daily meeting, or daily scrum.

In segment two we are going to study a number of **development practices** coming largely from extreme programming, which are also fundamental to Agile approaches.

In part three, we will focus on **release practices**. In Agile development you're supposed to release early and often, including intermediate versions. We are going to study how this works in Agile development.

In part four, we will concentrate on Agile practices that focus on **quality**. Quality is a really important part of the Agile discourse. And we will see what practices, in the Agile view, help, enforce, and ensure quality, in particular through systematic and frequent testing.

And finally, in segment five of this lecture, we're going to come back to some more **management-oriented practices**, which are also an important part of the Agile approach.

Slide 3

Agile Practices distinguish themselves in particular by prescribing a number of meetings following a quite precise, ritualized style.

And in this first segment of the Practices lecture, we are going to see what these meetings are.

The general organization of this lecture about Agile practices is to start with more technically oriented practices like development release, and quality-oriented techniques like testing, and to continue with practices that have more to do with project management.

We start, however, with meetings even though, in reality, they are more like management practice, because meetings are so important in Agile development, specific kinds of meetings which define, in particular, the daily work of a Scrum team or an extreme programming team.

These meetings are so important that sometimes they take on almost the aspect of a ritual.

The most important of these meetings, although they're not the first one chronologically, is **the daily meeting.**

We are going to see that there are meetings before, like planning meetings, meetings after like the review meeting.

But what defines the day-to-day working of an Agile team, in particular an extreme programming or Scrum team, is in part this daily meeting.

So as the name indicates, it's supposed to be held every working day. And specifically, it should be held every morning. It should be the first thing that people take part in when they arrive to work in the morning.

And the goal is very simple. Since it's a daily meeting, it should define the day's work.

But of course, in the broader context of the progress of the project, it's supposed to be a short meeting.

There is a phobia in the Agile world of very long, rambling meetings, technical meetings which can take far too much time, or organizational meetings.

And so the stand-up meeting, as it's sometimes called, is not supposed to be one of these.

And there's a strict time limit-- usually 15 minutes.

Sometimes it's called the stand-up meeting, because the idea is that if you attend the meeting standing up, you are not going to have the patience for a very long meeting.

In practice, that's not always applied the standing up idea. But the time limit is usually strictly enforced, so everyone should remain concise. And if someone starts rambling, he or she is soon going to be brought back to order by the others.

The meeting should involve all team members, but you remember from the previous lecture the distinction between committed and involved people. So the committed people are those who are supposed all to speak. So every committed member of the team is expected to speak.

And what is he or she going to do? Well, we are going to see this on the next slide.

Slide 4

It's three questions. But before I go to these three questions, which are quite important. Let's make sure we understand the purpose of this meeting is to define the commitments and to uncover impediments. That is to say, to find out what obstacles may be blocking the ideal progress of the team.

It's also important to see what the meeting is not about. Problems are going to come up during the meeting. But usually, unless the solution is simple, it is not the purpose of the meeting to go into deep discussions of how to find the best technical solution.

So even for impediments, usually the removal of impediments will take place outside of the meeting.

And if an impediment turns out to be a difficult software engineering question, it is not the purpose of the daily meaning to resolve it.

The purpose is to identify it so that, if necessary, some other discussion involving presumably fewer people can be scheduled outside of the daily meeting.

So the meeting itself is focused on three questions. This is where we have an almost ritual setup. And the three questions are codified.

What did you do yesterday?

What will you do today?

Are there any impediments in your way?

These are the three questions that each committed team member is supposed to answer. I must say, it's a quite brilliant idea and it works well in practice, in particular, because of the connection between the first two questions in a short time frame.

Of course, if we were talking about the monthly meeting, it would be much easier to make empty promises as to the second question-- what will you do in the next month?

But of course, people have memory for one day. And if one day you promise marvels, well, two days, what will you do today becomes tomorrow's what did you do yesterday?

So the sweet talkers, the ones who promise a lot and don't deliver much are quickly uncovered and you cannot do that much hand-waving.

Also important is the focus on impediments. And this is, of course, in Scrum, connected with the idea of the Scrum master-- part of his task is to remove and to help remove impediments.

And part of the purpose of the daily meeting is to uncover these impediments.

So this daily meeting is a very good idea, at least for teams that are physically co-located. And it can be somewhat hard to organize it, of course, in the case of a distributed development team.

Also, there are more flexible work arrangements these days. People can partly work from home.

So it's not always that easy to work out a daily meeting in practice. So when it's possible, it certainly is a very efficient way to help organize the work of a team.

What I have found in my experience is that some adaptations of the daily meeting also work.

For example, in our team in IBM, we had two weekly meetings. It's a distributed team, so it could be unrealistic, if only because of time differences, to have a 15-hour daily meeting.

But we have found that two weekly one-hour meetings, one very similar to the Scrum or XP stand-up meeting, and the other more focused on technical discussion and following an agenda, works well.

Slide 5

In Scrum, there are a number of other meetings connected with individual iterations, which, as you remember, are called sprints in Scrum.

So from now on when I use the word "sprint" or the word "iteration," it means the same.

A planning meeting is held at the beginning of every sprint, and the goal of the meeting is to define the work for that particular iteration.

The outcome is a sprint backlog with a time estimate for every task. Now, you remember the distinction of roles in Scrum, so the rule in Scrum is that you have an eight-hour time limit for the planning meeting divided into two parts.

So eight hour is typically a workday. In the first half, say the morning, involves the product owner and the team now we have a product backlog, which defines all the functionalities that the entire product eventually is going to provide.

Although, I shouldn't let it sound like it's a requirements document that has been codified at the beginning of the project once and for all.

Of course, the product backlog is an evolving document in Agile methods and in Scrum in particular.

But still, at a certain point in time, there is a product backlog. And it is under the control, as you remember from the previous lecture, of the product owner.

He or she is the one who determines what functionalities are relevant for the customer.

So in the first half, the work is devoted to prioritizing the remaining elements in the product backlog.

And this, of course, has to involve the product owner. But you also remember from the previous lecture that defining the sprint backlog, the list of tasks for an individual iteration, is not the responsibility of a manager or, in Scrum, of a product owner.

It's the responsibility of the team. So in strict Scrum, the second half of the meeting is devoted to producing that backlog and is reserved for the team with no participation of anyone else, including the product owner.

Slide 6

That was for the beginning of an iteration. At the end of an iteration, there is a **retrospective meeting**, where all team members reflect on the past sprints, may continue process improvements.

And here, the two main questions are, what went right, and what could we improve?

The standard here is to have a three-hour time limit. This is an internal meeting for the team.

It does not normally involve outsiders like customers, for example.

Slide 7

There's also another meeting, which is destined for the other stakeholders as well, and of course, for the team as well. So this is where the team meets the customers.

And by the way, I'm not implying that the **review meeting** occurs after or before the retrospective.

It could be either way, but it might be that it's better to have the review meeting with the customers first and then the retrospective.

So the review meeting is devoted to analyzing what has been done. And the question is, what has been completed, what has not been completed? Of course, it should present the completed work to the stakeholders.

And what better way is there to present completed work than to demo it? Remember the Agile emphasis on producing a working system at every iteration. Well, this is your opportunity to show to the customers what has been done. And of course, remember the whole discussion of waste. Anything that is incomplete will not be demonstrated. And the emphasis is on what has been done and can be demoed. And there is, as a standard, a four-hour time limit.

Slide 8

Let me use this opportunity to mention a somewhat related idea emphasized in the Crystal method-- the notion of **reflective improvement**.

We are getting very close here to ideas emphasized in a completely different part of the software engineering world, something which is often considered the other extreme on the spectrum as compared to Agile Processes, **CMMI, the Capability Maturity Model Integration.**

This is the more process-oriented approach. And there is, at level five, the top level of the CMMI, a notion of self-improvement, where the process is organized so that it includes mechanisms to include itself, to include the process itself.

And here, this is an area where some very different software engineering techniques, such as Agile on one hand and CMMI on the other hand, really meet.

And in fact, a number of people have attempted to combine CMMI and Scrum or other Agile Methods.

So here, it's kind of the same idea. Developers, according to the Crystal rules, should take breaks from regular development to look for ways to improve the process.

So it's not necessarily a specific meeting at a specific time. It's regular meetings to improve the process. And iteration is helpful with this in the Crystal view by providing feedback on whether or not the current process is working.

So we get an idea that is very similar to the Scrum review meeting.

Slide 9

What we have seen in this first segment is that a major part of Agile development is

a set of well-defined meetings taking a kind of ritual role that define the day-to-day life and the iteration-to-iteration progress of an Agile project with well-defined goals, outcomes, and participants.

The most important is the daily meeting. In Scrum, we also have planning, review, and retrospective meetings.

And we have also hinted at the fact that these meetings were really designed for the ideal case of a team, a classical team that is all together under one roof.

And for distributed teams and modern forms of work, these ideas may require adaptations.

But the're still based on very sound ideas.

Proponents of Agile methods, in particular Scrum, like to say that these methods can be applied to a broad range of project kinds far beyond software.

But the original and still the most important area of application of Agile methods remains software.

Slide 10

And in this second segment of the practices lecture, we are going to study a number of technical software-oriented practices, coming largely from Extreme Programming, which are central to the application of Agile methods to software development.

Development-oriented Agile practices affect the way we develop software, that is to say in particular how we design it and how we implement it.

Most of these techniques actually come from XP, Extreme Programming, because of all the Agile methods, this is the one that is the most software-specific, less focused on management and particularly focused on development.

Slide 11

The best known and most controversial of these techniques is **pair programming**. It's controversial because Kent Beck, the originator of Extreme Programming, told everyone, in particular in his books on Extreme Programming, that this is the way everyone should develop software, always in pairs.

Well, in fact, very few teams do this all the time. However, pair programming remains an interesting technique to be applied occasionally.

So what's the idea?

Well, pair programming means that you have two people developing software together.

So instead of having one programmer, you have two writing a single piece of code. They are sitting together at one machine. One of them, of course, at any time has the keyboard and mouse, but they could change. And they think out loud, so they are not going to enter anything into the editor before explaining what the person at the keyboard is planning to enter, or if it's the other person taking the lead what he or she suggests entering.

So everything is discussed and everything is explicit. And that's one of the main advantages advocated for this process. Instead of keeping everything for yourself in your head and sometimes being vague, in particular the rationale for what you are doing, you make it explicit.

And often-- and that's one of the immediate advantages-- you are trying to explain your way of thinking, your rationale to someone else.

Means that you're going to realize that you cannot express it or you can find an inconsistency or a contradiction right away, and so you'll be stopped in your tracks and this will be a bug that does not happen.

Another advantage, of course, is that the two partners can keep each other on task. At least that's the idea.

They are not going to start talking about their latest vacation, but they focus and they keep each other focused on their programming task at hand.

They can brainstorm together.

They can clarify their ideas.

What happens also in software development, of course, is that once in a while you get stuck. Well, there's a chance that not both of them will get stuck at the same time, so when one is stuck, the other can take the initiative.

And of course, in a team that has practices, the best practices, the method guidelines or the company guidelines or the project-style rules, then they can hold each other accountable and kill any attempt to deviate from or violate the rules.

And this is actually a very good technique.

What is really not desirable is to impose it as the sole technique for developing software, as the most extreme of the Extreme Programmers would have us do.

It turns out that it's one of the Agile techniques for which we can test the effects on productivity and quality, because of course on the one hand you have classical managers, maybe the MBA types who say, well, this is going to decrease productivity because for the same number of programmers you are going to have twice as few programs being developed.

And then you have the XP advocates, which say, well, any of this is going to be more than compensated by all the bugs that we have avoided by finding them before they even have had a chance of becoming bugs, simply bad ideas that are discarded right at the root.

So we can test that.

And it's one of the areas that has been the most thoroughly studied in experimental studies of Agile programming. So you take two groups of programmers, one using pair programming, the other not using it, using for example more traditional quality techniques, such as code reviews, code inspections, and you look at the results.

And these results, as confirmed by many studies, do not show any major advantage one way or the other.

It turns out that pair programming has about the same results in productivity and in quality than more traditional quality-oriented techniques like code reviews.

It doesn't mean that pair programming is a bad idea. In fact, it's a very good idea, especially for tricky cases.

And this has become part of the modern battery of tools of the programmer as one of the tools.

Of course, it's a mistake to impose it as the only tool. But for delicate cases, it's quite natural for programmers to want to study and implement a particular piece of the project in collaboration with someone else.

One mistake that is sometimes made in practice is to confuse pair programming with mentoring, that is to pair a senior person with a more junior person with the hope that it's going to give the advantages of pair programming as well as help train the junior person.

Well, mentoring is a good technique, but it's different from pair programming and you cannot do the two together and expect the same results.

So you can do mentoring, you can do pair programming, but for strict pair programming, the two members of the pair should be of approximately equal levels of competence.

Slide 12

Another Agile technique, specifically Extreme Programming, is the idea of a **single code base**.

And there's a strong emphasis, and I would say quite justified emphasis, in the XP community against branching.

Branching, which is in principle easy—you start developing two parallel versions of the software-- causes lots of problems, because at some point you have to reconcile these branches.

And a rather simple view that you should have just one branch is quite healthy.

Slide 13

Another XP emphasis is on **shared code.** Agile methods reject code ownership, that is to say the situation in which one person is responsible for a particular part of the code and is the only one or a small group is the only one permitted to change a certain part of the code.

Now, we're not talking here about intellectual property. This is another matter.

We are talking about permission to change. And in many companies, some parts of the software are the exclusive province of someone or a group of people.

And in Agile methods, particularly in XP, the idea is that this should not be the case and that anyone in the project should be permitted to modify anything.

This is more debatable. Of course, the arguments are clear.

Code ownership creates unnecessary dependencies between team members and creates delays.

It also gives the opportunity for blame game, because my stuff is not working but it's not my fault.

I can do nothing about it. I depend on my colleague's code, which I cannot change.

So all this is good.

On the other hand, it's also true that you need some discipline in a software project and that sometimes some people are much more competent than others at modifying delicate parts of the code.

So this is a piece of advice which is good for reflection, but it should be applied with caution.

Slide 14

Another standard Extreme Programming piece of advice, which of course is not used with XP, is to **leave optimization until last**.

Wait until you have finished a user story and run all your tests before you try to optimize your work.

This is usually good advice.

But of course, sometimes optimization cannot be completely done afterwards, especially big optimization having to do with algorithm complexity and the choice of the right algorithms.

Slide 15

Continuing with design, there's this idea that we have already seen of **simple design** in Agile methods in general and XP in particular.

Produce the simple design that works—remember the slogan, the simplest thing that could possibly

work-- and then refactor.

Refactor means improve the design efforts and we are going to come back to refactor in just a minute.

It's not only simple design, but it's incremental design.

Slide 16

Now, you might think that you're practicing **incremental design** even if you're not using Extreme Programming.

But here it goes further than the basic idea of doing things one at a time, which is rather straightforward.

Here, incremental design as seen in Extreme Programming really means don't abstract right away.

Look at individual cases and solve these individual cases, as explained in some detail in this article by Shore here.

Start by creating the simplest thing that could possibly work.

Then incrementally add to it as the needs evolve, rather than looking for abstractions.

And this article actually criticizes politely the usual practice of abstracting first.

It recognizes, it admits that the ability to think in abstractions is often a sign of a good programmer.

Yes. And that is definitely true.

But still, this article argues against it, XP argues against it by saying, don't try to be all things to all people.

This is also part of the general push against too much a priori reuse and too much a priori design for extension, for extendibility.

Well, just do what you are asked to do and then generalize. Well, as often, there's a mix of good and bad here.

It's true that sometimes it's better to solve the problem at hand.

But it's also a mark of the good engineer that he or she designs not just for the problem of the moment but for future extensions.

And abstractions indeed are signs of good programmers.

So this is kind of the more Luddite aspect of Extreme Programming, to be viewed with caution.

Slide 17

Now, when we look for abstractions, one technique that some Extreme Programming literature recommends is the **system metaphor**.

Now, this is somehow gone out of fashion, so I'm not going to go into it in detail.

I'll just mention the idea.

How do you actually abstract right?

Well, one of the elements of advice here is the system metaphor, which is meant to be agreed upon by members of the project to explain the purpose of the project and guide the structure of the architecture.

Slide 18

So as example from that same paper by Tomaiko, let's assume we have a wrist camera.

And then a metaphor that could help drive the project is to consider this wrist camera as cities and towns.

Or a variable transparency window, well, we think of a chameleon if the window uses variable opaqueness to vary according to a sensor reading.

Well, I'll leave it to you to decide how useful that is, but it's important to mention it.

Slide 19

More important and much more widely used and influential is the notion of r**efactoring**, which was largely developed in a book by Martin Fowler, again the reference is in the bibliography.

And the idea of refactoring is that when you have something that works, when you have a design that you have implemented that satisfactorily addresses the user stories that you have or the tests that you have, you shouldn't stop there, because it may be good enough for the moment, but it may not be good enough for the long term.

So this is where abstraction comes back, and in particular where the distrust of extendibility and reusability stops, because we do want to have good designs in the end, not just designs that work, but designs that will stand the test of time.

So the idea of refactoring is that once you have something that works, you sit back and take a look at how good it is from the viewpoint of design.

And then you apply refactoring techniques from a catalog published in particular by Fowler.

There are other additions to this catalog, of course, from later on. So examples are to encapsulate a field or an attribute into a function, to replace a conditional with a more object-oriented technique of polymorphism and dynamic binding, to extract a routine or method from some code to abstract it into a routine if that code is important or if it's used several times, to remove duplication, to rename, give a better name to a routine or an attribute, to move it to another class, to move it up or down in the inheritance hierarchy.

This is, of course, a very good idea.

In Agile methods, it's widely used as a substitute for up-front design. We've seen the rejection of it from tasks, in particular from design, so the mantra goes that you don't need to do up-front design, you should just do the simplest thing that works and then refactor.

And of course, this is completely wrong. The idea that you can take anything and turn it into a good design is absurd. Junk in, junk out, or garbage in, garbage out.

Now, the refactoring is best applied to good designs, good but not perfect designs.

And so here, as in other cases like pair programming for example, we have a case of an Agile technique which is often presented in the Extreme form as a replacement for well-established software engineering techniques and is much better applied as a complement to them. And in fact, refactoring in this specific case works best with good but of course not perfect up-front design.

Slide 20

What we've seen in this segment is a number of practices that have exerted significant influence on the way we develop software, in particular, two of them, pair programming, the first one, and refactoring, the last one.

And it's important to note that they're extremely interesting ideas, but they are best viewed as complements and not replacements for more up-front style techniques, which we have learned to apply and perfect over four decades of history of software engineering.

Slide 21

One thing that any software project has to do is once in a while to **release software**.

In traditional software engineering, releases were few and far between.

In theory, you can have a project, even a successful project, with just one release at the end.

That's the minimum.

Well, welcome to the new world of Agile software development where releases come early and often.

It is one of the central tenets of all Agile methods, that you should have a frequent release policy.

And in this segment of the practices lecture devoted to releases, we are going to study exactly what the policy is for releases in Agile software development.

Agile methods, as we know, emphasize frequent integration, frequent deployment, frequent releases.

And the practices that we are going to see in this segment in each—each in its own way emphasize these goals.