Introduction to Week 3

At the very core of the Agile approach lie a number of principles which we are going to study in the five segments of this lecture. Some of these principles are organizational. We could also say managerial, and they have to do with how we organize our software projects and how we manage them.

Some of the other principles are technical. They are directly software related, and they have to do with how we write our programs in an Agile approach.

Many of the organizational principles come from Scrum. Many of the technical principles come from Extreme Programming, although there is a bit of both on both sides.

In the first segment, we're going to review non-Agile ideas. They're important, however, to understanding the Agile approach because they are the kind of things that Agile lists like to rant against, like to differentiate themselves from. These are ideas from traditional software engineering.

In segments 2 and 3, we will review organizational principles.

In segment 4, we will review technical principles.

And finally, in segment 5, we're going to review a number of principles that are not universally accepted by all Agile methods but are specific to this or that Agile method, like for example, Scrum, or Extreme Programming, or Crystal.

Proponents of Agile methods like to present their ideas against the established order of software engineering. So to understand better what agile is about, we need to take a quick look at some of the ideas that came before, the kind of ideas that advocates of Agile approaches like to criticize and want to replace.

Slide 3

Before we start looking into the details of Agile principles, remember how we are studying Agile ideas. We have classified them into six categories-- values, which we saw in the last segment of the previous lecture; **principles, which we're going to see now**; roles; practices; and artifacts.

Slide 4

And the principles themselves are going to be eight of them divided into two categories, organizational principles and technical principles.

And let me read these principles now, and then we're going to study them in detail.

The organizational principles are, **one,** put the customer at the center.

Two, accept change. Remember that the Agile manifesto said welcome change. Accept change is perhaps a bit more realistic.

Let the team self-organize, number **three.**

Number **four**, maintain a sustainable pace.

And number **five,** produce minimal software, in three forms of minimalities, so we have three subprinciples here : Produce minimal functionality, produce only the product requested, 5.2 and 5.3, develop only code and tests.

All these ideas have to do more with the management aspect of software. And then we have the more technical aspects with three principles, two of them with subprinciples.

Six, develop iteratively in two aspects of iteration. First, produce frequent working iterations. Second, freeze requirements during iterations.

Seven, treat tests as a key resource. And in particular, do not start any new development until all tests pass, and test first. There is this notion of test first development, TFD, or test-driven development, TDD.

And finally, number **eight,** express requirements through scenarios.

Slide 5

Before we start going into these principles, it's useful to understand exactly what kind of things the creators of Agile methods were reacting against. Agile is very much a rejection of some ideas, and it's important to see some of the basic elements of these ideas.

The origin of much of modern work on software processors is an article by Royce from the US Air Force going back to 1970, which introduced the waterfall model.

And the scope and the goal of this article, which has been extremely influential, was to describe the set of processes involved in the production of systems and how we sequence, how we organize in sequence, the succession of these steps.

A life cycle model is a model in both meanings of this term in ordinary language.

A model is an idealized description of the reality, but it's also prescriptive. It's also an ideal to be followed and a life cycle models, such as the waterfall are both.

Slide 6

This is a little screen shot from original page shot from the original Royce article explaining that we need to put some order in the process of developing software.

Slide 7

And the famous diagram, known as the waterfall diagram because of its shape, distinguishes between a number of steps which are actually fairly close to what we still do today 46 years later. **System requirements, software requirements, analysis, design, coding, testing, and operations.**

Well, it's perhaps not exactly what we do, but it's still pretty close.

And there's also these arrows, which as the legend of the figure says (suggests), that hopefully we should limit the interactions between steps to neighbouring steps.

Slide 8

One interesting aspect of the waterfall article, of this original Royce article, is that even though it was the first published description of this model, the waterfall model, it was already criticizing it.

And in fact, this is one of the major uses of the waterfall model, is to serve as a foil for all kinds of shots taken at it.

But in fact, the waterfall model, of which here we have a somewhat more modern version, is something else. It's really a pedagogical device. It explains a kind of idealized process at one extreme of the spectrum. The other extreme of the spectrum would be a total absence of any discipline, any order, whatsoever. And the practical processes that companies and development teams apply are somewhere in between those two extremes.

It's important to mention that because in the Agile literature, you'll find a lot of criticism of the waterfall. But in fact, that is not new. Everyone criticizes the waterfall, and very few

companies, if any, have ever applied a straight waterfall because it's impossible. It's more like a pedagogical notion used to explain concepts.

Slide 9

There are some arguments for the waterfall though in principle. And this I'm taking from a famous book, Software Engineering Economics, by Barry Boehm. He points out in that book-- an old book, but still quite useful-- that the activities are necessary.

We need to do some kind of requirement, some kind of specification, and so on. And in the abstract in principle, the order of these activities is the right one. We should do requirements before we do a design, and so on.

So those are arguments in favor of the waterfall showing that it's a serious idea to be taken with consideration.

Slide 10

On the other hand, there are lots of things which we can criticize with the waterfall, such as the late appearance of actual code. This is perhaps the major problem.

In software we know whether something is going to be satisfactory or not when we see the actual code. And a process in which the code only appears very, very late is fraught with risk.

There's a lack of support for the requirements change. There's this myth that the requirements are set at the beginning of the project and don't change. Well, of course that's not how things happen in practice. Requirements do change.

There's little recognition for the maintenance activity, which is often considered to account for 70% of software costs.

There's a division of labour hampering total quality management. That is to say, you tend to have people who are more in charge of analysis and requirements people who are more in charge of design and so on.

And of course, in order to get a satisfactory result, we need to have people who feel responsible for the whole thing.

Then we have impedance mismatches, if we consider requirements, design, and implementation as different steps, then there is a problem of translating between the results of these steps.

Slide 11

There is an old cartoon which describes this quite amusingly. It's from the same time as the Royce article or even before. No one knows who the author was, but it's still quite appropriate.

Slide 12

Now, this is the kind of thing that Agile methods reject. They also reject the upfront requirements phase, even though it is well-known from many studies in the field cited here-- and I'm not going to go over the details-- that requirements errors are one of the primary sources of bugs and catastrophes in software development.

So it's important to get requirements right. Of course, the agile literature doesn't tell you that you should not do requirements. That would be absurd. But it rails against up front requirements.

Slide 13

The general idea is that requirements done upfront are bad, and there are two criticisms there, which are sometimes merged into one. But it's really two different arguments.

One argument is that requirements are going to change the very moment the ink is dry on the document that describes the requirements.

And the other argument is that requirements are not delivered, and they are what the lean method calls waste.

So these are two different criticisms. And in particular, the change criticism doesn't mean that requirements are useless. Of course requirements change like everything in software, like code, like designs… It doesn't mean we don't do code, or we don't do design.

But still, in a typical text here by Kent Beck, the two criticisms are merged. Software development is full of the waste of overproduction, such as requirements documents that rapidly grow obsolete.

Slide 14

Here are some Agile views on requirements.

Kent Beck: Requirements gathering is not a phase that produces a static document,

but an activity which produces detail just before it's needed throughout development.

Mike Cohn, one of the well-known proponents of Scrum: Scrum projects do not have an upfront analysis or design phase. All work occurs within the repeated cycle of sprints. So the idea is that you start coding and doing requirements at the same time in a stepwise fashion, and each phase brings a new round of requirements.

And **Mary Poppendieck** from the lean approach: and those things called requirements, they're really candidate solutions. Separating requirements from implementation is just another form of handover.

Handover is a term of the lean approach. It's, of course, a negative term. It's any situation in which there is what I called a moment ago an impedance mismatch, that is to say, two different people have to hand over one to the other a certain result to be consumed by the other person or the other group with, of course, the difficulties of translation and a problem of dealing with change.

Now, this Poppendieck criticism is partly justified. It is true that sometimes people put into requirements what are really sneaky ways of making early implementation choices.

But, it's not a reason to throw away the baby with the bathwater…. Not all requirements are early implementation or design decisions, and any good engineering has to define what the problem is before solving it.

And we are going to see later on in this lecture the Agile replacement for requirements in the form of user stories. Here I would say that one of the worst pieces of advice that I've seen applied by people taking the Agile credo literally has indeed been **the rejection upfront of upfront requirements.**

So as usual, Agile authors have a solid point in criticizing the over lengthy periods that some projects take just to do requirements before actually building a solution that can be exaggerated. But all engineering requires planning. All engineering requires a definition of the problem before building a solution. And certainly, this is a place where as I'm presenting the Agile approaches, it's my duty to warn you of some of the risks of applying Agile admonitions too literally :D…

Some of the worst catastrophes that I've seen in software projects applying Agile methods have been due to the refusal to spend a little time at the beginning of the project, maybe just a

few weeks, just to sit down and decide overall up front what the system should do before actually starting to build it.

Slide 15

So what we've seen in this first segment of our lecture on principles of Agile development is (1) some of what Agile advocates reject, the waterfall model, which is useful as a foil, as a pedagogical device, and which no one really applies 100% in practice.

(2)The role of requirements, both in the traditional world and in the Agile world, and (3) the risk of applying extreme precepts literally.

Slide 16

In this second segment, we review a first set of organizational principles coming mostly from Scrum, a few of them from extreme programming, which tell us how to organize and in particular how to manage our software projects.

Slide 17

The first set of principles that we are going to see (remember our list of Agile principles) are all organizational.

We'll cover four of them in this segment, (1)put the customer at the center, (2) accept change, (3) let the team self-organize, and (4) maintain a sustainable pace.

Slide 18

The first one is to put the customer at the center.

There's a very strong emphasis in Agile message on making sure that customers are involved, not just at the beginning and at the end as in some traditional developments where you'll write requirements with users, with customers, at the beginning, and then you'll present the result to them.

In Agile methods, you're supposed to involve customers throughout to make sure at every instant that you're building the system that they actually want, not just building the system right, but building the right system.

Here, for example, is **Kent Beck:** *You will get better results with real customers. They are those whom you're trying to please. No customer at all or a proxy for a real customer, by which he means for example a requirements document, leads to waste, as you develop features that are not going to be used, specify that do not reflect the real acceptance criteria important to customers, and you lose the chance to build real relationships between the people with the most diverse perspective of the project.*

And so there is two solutions to address this issue, the extreme programming solution and the Scrum solution.

In XP, the idea is to have an embedded customer who is a member of the team. So you take a customer away from his or her organization. He or she is an expert in the application domain, and you put them in the project along with the developers. I must say that in practice, that doesn't work very well because it's difficult to find such people. And even if you find one, he is probably not going to be representative of all users. Typically, what characterizes customers is that they have diverse perspectives, and various people involved in the customer organization have a diverse perspective. So if you find just one, there is a serious risk that you are going to be biased.

Scrum replaces this idea with a different one, the notion of product owner. So this is different because a product owner is officially authorized to commit the organization. The role of the product owner takes away part of the traditional role of the manager. We've seen that in Agile methods, many of the roles of a manager are given to other actors, and this is one of them. One of the things that managers traditionally do is to decide after an iteration of the development whether it's acceptable or not. Well, this is the task of the product owner, who is entitled to commit the customer organization for this kind of decision. And I would say in practice this works better than the embedded customer of XP because of his official responsibility and commitment.

What is common to both approaches, extreme programming and Scrum, is the importance of involving customer representatives at every step. Now, it's important to note to take a bit of distance from these Agile techniques that customer involvement is a useful complement to requirements, but not a replacement. You still need to express, and preferably in writing, what the system should do and of course to involve customers in the definition of whether the description of the requirements is correct or not.

Slide 19

Second principle, accept change.

As we've seen in the Agile manifesto, the formula is that we welcome change, but in fact this is a bit of an exaggeration…

The notion of supporting change is not new in software engineering. In fact, every software engineering textbook talks about extendibility, which is one of the important qualities of software systems, the ability to be changed easily in response to changing user requirements.

One of the best techniques that is known to achieve extendibility is object technology, particularly object-oriented programming. And in fact, that is one of the main claims to fame of object technology and one of the reasons for its success.

Interestingly, while the Agile literature promotes change, that is to say extendibility, it tends to take a somewhat contemptuous view of object technology. For example, here in this citation **by Mary Poppendieck:**

"While in theory object-oriented development produces code that is easy to change, in practice, object-oriented systems can be as difficult to change as any other, especially when information hiding is not deeply understood and effectively used."

Well, yes, certainly. Any approach which is applied by people who do not understand it and do not use it effectively is not going to be indeed very effective.

But that doesn't say very much, and it's somewhat surprising to see that while on the one hand Agile methods promote the abstract idea of extendibility, of welcoming change, in practice they fall completely short of proposing actual techniques for achieving it.

And I would say that in practice, object technology, in spite of all the unfounded criticism, unjustified criticism, is still the best known technique for achieving extendibility.

Slide 20

The third principle (again an organizational principle): **is let the team self-organize.**

So the traditional view here is that you have a manager, and one of the main tasks of the manager is to tell workers to do their job, to assign tasks on a day-to-day basis.

In the Agile view, this is no longer the responsibility of the manager, to the extent there even is a manager.

Instead, a team is self-organized, and the members of the team collectively decide on task assignments. And the role of the manager there is to listen to the developers, to explain what can be done, to coach, to mentor, to encourage progress, to help catch errors, to remove impediments.

We'll come back to this notion of **impediment**. An impediment is an obstacle, and it's a notion that plays an important role in Scrum. It's an important part of the tasks of the Scrum Master, the coach to remove impediments. We'll come back to that…

The manager, the leader, is here to provide support and help in difficult situations and to make sure that scepticism does not ruin the team spirit. But the team chooses its own commitments and has permission to talk to customers.

This idea is quite central in Agile methods.

Slide 21

It reminds us perhaps of some orchestras that are self-organized, the most famous of which is the "I musici".

But we can also note, if we take this analogy, that such ensembles are really a minority, and that most orchestras still have a manager.

Slide 22

Now, the goal of self-organization was well captured by this blog poster poster cited here.

The most important aspect is to put the management of the project where it belongs, on the backs of the people doing the work.

On the other hand, if you look through the small print, you'll realize that someone like Ken Schwaber, for example, ---one of the creators of Scrum,--- explains that it's not completely self organization, that is still a lot of control, and in particular **subtle control**.

And I must say, this kind of comment makes me cringe a little bit because if I'm controlled, I'd rather be controlled **explicitly rather than subtly.**

And perhaps it's better to have a manager than to have impressions of self-organization and someone pulling the strings in the backstage.

So here, this is one of the areas where Agile message provides some important ideas, but we shouldn't be too strict and too dogmatic. In some cases, you will have teams that can self-organize, like "I Musici". In others, a traditional manager is required.

Slide 23

The fourth principle (the last one we'll see in this segment) **is maintain a sustainable pace.**

Here, the idea is to go back to these two books shown here, **one by Ed Yourdon, criticizing the notion of death march.**

You know the **death march** is the **kind of extreme pressure put on programmers typically to meet a deadline.**

And there's also another famous book **by DeMarco and Lister, Peopleware**, with two editions **explaining how programmers actually work and explaining in particular that programmers need to have some peace, some quiet, and some respect in order to do a good job.**

Slide 24

So here, as explained for example in XP, in Scrum, in Crystal, the emphasis is that people perform best if they are not overstressed.

So there is this emphasis on not forcing people to work more than 40 hours or whatever the legal limit is in a given country. And it varies over time…

We can work, which can always happen. It should not happen regularly. It should be compensated by some free time in the next week.

So this is part of a general approach which promotes (1)**frequent code merge**, (2) **always maintaining an executable test covered high quality code so that programmers can be proud of their achievements.**

(3) **Constant refactoring.** We'll talk about refactoring in more detail…Refactoring means improving the code and the design.

(4) **And a collaborative style of working in constant testing.**

Slide 25

So what we've seen in this second segment of our principles lecture is a number of principles.

Put the customer at the center, accept change, let the team self organize, maintain a sustainable pace.

We've also seen some criticism or limitations that can be expressed about these principles.

And of course, we have a number of other principles complementing these, which we're going to cover in the next segment of this lecture…

Slide 26

The third segment of this lecture about principles introduces a few more organizational principles to organize and manage our software projects.

Slide 27

The final set of organizational principles is actually one principle but it is divided into three subprinciples.

It's all about minimality.

The Agile approach wants you to produce as little software as you can in order to produce what you do produce right.

And minimality or minimalism has three aspects which are distinct, **produce minimal functionality, 5.1**; **produce only the product requested, 5.2**; **and develop only code and test 5.3 .**

Slide 28

So the first is about developing minimal software.

Slide 29

This is a principle which was introduced I belief by extreme programming and which is commonly referred to in the Agile world.

It goes by the slogan YAGNI for You Ain't Gonna Need It…

That is to say, before you build a certain functionality into your system, make sure that someone somewhere really is going to need it.

So here's what Ron Jeffries has to say, this principle, the YAGNI principle, *reminds us always to work on the **story** we have, not something we think we are going to need, even if we know we are going to need it…..* which is perhaps a bit extreme.

When he uses the word "**story"** he is referring to something that we're going to study in more detail, the notion of user story, which is the unit of functionality and also the unit of requirements.

And so what he's saying is you get stories from users and don't try to do more than what the users have requested. They are going to give you individual stories, meaning individual pieces of functionality, and don't try to be too general.

Here's another view on the same topic by Mary Poppendieck, "*our software systems contain far more features than are ever going to be used.*"

So she's criticizing many existing systems. Extra features increase the complexity of the code, driving up costs non-linearly.

If even half of our code is unnecessary, a conservative estimate, the cost is not just doubled.

It's perhaps 10 times more expensive than it needs to be.

So as always, there's a grain of truth here.

And certainly we have all grappled with software systems that we think have far too many functionalities, including things that we are never going to need.

And when we're never going to need them, it's easy to conclude that no one is going to need them.

But of course, the reality is usually more complex than that.

If a functionality is there in a software system, it's not just because some crazy programmer has decided to put it in, but usually it's because some customer, some perhaps big money paying customer, wants it.

So it's easy, of course, to dismiss systems as being too complex and having too many functionalities.

But your useless bell and glitch may be my essential functionality. So these things are not-- this kind of criticism is partly founded, of course. It is true that people sometimes build in too much functionality. But there is only so much you can do to remove functionality. And of course, the quantitative elements in this criticism, non-linearly, if even half of our code is unnecessary, conservative estimate, it's perhaps 10 times more expensive than it needs to be, all this is very informal.

I don't know of any studies that actually back these claims. So this is the kind of Agile advice that doesn't really go very far and should be taken with a grain of salt.

Slide 30

So, in fact, if you're listening to Mary and Tom Poppendieck, it's more interesting to look at their definition of waste, because this is much more constructive and practical.

So in the Lean approach, as we have already noted, there's an emphasis on avoiding waste.

And this comes from industrial engineering. As I mentioned, it's from car production initially. And it's transposed here to software.

So she talks about seven wastes of software development and gives examples for software.

And it's actually quite interesting to look at this list, because it gives us some practical things to look out for, a checklist of things to check. And it's true that many software projects sin in some or sometimes all of these dimensions.

So extra, **unused features, that's over-production in classical industrial engineering**, so don't build in any features that you're not sure someone, somewhere is going to need.

Partially developed work which is not released. In classical industrial engineering, this corresponds, of course, to **inventory**, which people try to minimize. And in software, it's anything that the programmer tells you excitedly about, it's being developed but it's almost done, in the famous sense of almost in software engineering and it's never going to be released. Well, this is waste.

Intermediate unused artifacts, so things that were, for example, preprocessors or various tools that had been built in order to help the project but in fact do not help it and just take time away.

Seeking information, in industrial engineering this would be motion, too much time and money spent on moving products around. And Poppendieck's analogy for this in software is

information that we don't have. We cannot do something because we are missing information about the customer needs or we're missing information about algorithmic techniques, and we waste time because of that.

Escape defects not caught by test reviews. So in general this is defects, and of course defects are very costly.

Waiting, including waiting for customers to tell us what they need or to give us some critical information about the business.

And finally, **handoffs.** We've also encountered that the handover is the same thing. So in industrial engineering, it would be transportation, money and time lost by moving products or materials from plant to plant. In software, it's anytime there are several notations and tools used to work on the same thing, for example, an analysis and design notation like UML and then a programming language where you have to do translations and communication between various formalisms. It's much better to work in a single framework if you can.

So this checklist of sources of waste to look for is a very useful contribution of the Lean approach.

Slide 31

The second aspect of minimalism is **to build a product only.**

And this is a reaction against too much pressure to build extendable software, that is to say software that will be changeable for new purposes, new requirements, and also to build reusable software.

So here's a text by Ward Cunningham, one of the very top names in Agile, in particular in extreme programming, which talks about this.

"You're always taught do as much as you can."

And he means of course in traditional software engineering.

"Always put checks in.

Always look for exceptions.

Always handle the most general case.

Always give the user the best advice.

Always print a meaningful error message.

Always this, always that.

You have so many things in the background that you're supposed to do, there is no room left to think.

I say forget all that and ask yourself, what's the simplest thing that could possibly work?"

This last sentence, what's the simplest thing that could possibly work, is also a well-known Agile, in particularly extreme programming, slogan.

Well, this is really exaggerated of course….

It's hard to criticize Walt Cunningham, who has many achievements to his name, in particular the invention of the wiki.

But he is really going too far here and he is denying the effects of 40, 50 years of software engineering and good software development practice.

And indeed, the difference between good engineers and bad is partly that a good engineer is going to look beyond the immediate goals, beyond the immediate requirements that he or she has been given and instead try to build a product that is going to stand the test of time and the test of change.

So this kind of advice really is somewhat Luddite and is best disregarded.

And same thing about reuse.

Slide 32

Here is another top name in extreme programming, Ron Jeffries.

So basically, I'll let you read the details of the text.

But what he's telling you is that he doesn't try to do more than what he's asked to do.

And it's a waste of time most of the time to think about possible reuse.

Just do what you're asked to do.

And of course, this is not what good engineering is about, in particular good software engineering.

In good software engineering, we try to think of not just today's project, but the next project.

As always, there's a grain of truth.

It's true that some programmers may indulge in too much perfectionism and see the long term at the expense of the short term.

In the short term we have a product to deliver, but of course good software engineering means that you build stuff that is going to be useful again for your next project and the one after that.

And really, this kind of advice is best ignored.

Slide 33

There's a third aspect to minimalism. It's the emphasis **on building only code and tests**,… so

in a somewhat view of Agile method, but it's really what Agilists recommend.

All that really matters and all that you should build is code and tests.

So Alistair Cockburn writes, *"you get no credit for any item that does not result in running tested code."*

And then, as I pointed out, he is somewhat more modern than some of the other Agilists so he adds as a qualification, OK, you also get credit for final deliverables, such as training materials and delivery documentation.

Well, thanks for the permission.

But my customers would probably be very upset at me if I did not deliver training materials and everything that they expect for practical use of the system.

Poppendieck, "*the documents, diagrams, and models*"-- She doesn't like the documents. "*The diagrams, the models produced as part of a software development project are often consumables, aids used to produce the system but not necessarily a part of the final product. Once a working system is delivered, the user may care little about intermediate consumables."*

Well, sure. But this is true of any engineering, right? If you are walking over a bridge, you're a user of the bridge and you certainly don't pay much attention, you care little about the plans and diagrams that were used by the bridge builders.

On the other hand, it doesn't mean they were useless. So again, this kind of advice saying reject any intermediate product, just focus on code and tests, has a grain of truth.

It is true that people in projects sometimes pay too much attention to intermediate products. But in the end, you do need, like in any good engineering, plans and documents and diagrams. Otherwise, you're not going to get very solid results!!

Slide 34

So what we've seen in this segment is the Agile insistence on minimality in three ways, minimal functionality, build a product only today's product only, not tomorrow's, and only code and tests.

And we have also seen that there is a criticism of approaches that put too much emphasis on plans and intermediate documents. The criticism is partly justified, but it really goes too far.

And these are not the best parts of Agile message.

We should not hear some of the more extreme advice and we should practice good engineering.