Introduction to Week 4

Slide 2

In this fourth segment of the principles lecture, we move away from organizational and managerial principles to focus on what should be our focus, namely software development.

So we are going to see a number of technical principles coming from extreme programming and other methods, which are specific to using Agile methods to develop software.

We now move on from the organizational principles of the last two segments to technical principles.

Slide 3

There's going to be three of them.

Two with sub-principles develop iteratively, treat tests as a key resource.

This is going to be the central role of tests in Agile development and express requirements through scenarios, for which the more specific technical term will be user story.

Slide 4

First, develop iteratively-- this has two aspects, **6.1 frequent working iterations**, **6.2 the close window rule, freeze requirements during iteration**.

Slide 5

First, iterative development-- well, if we look at the notion of iterative development,

there are two possible interpretations.

And let me say right away that the Agile view of iterative development is the second one, the horizontal one, in which we have horizontally layered clusters.

But it's useful to generalize the discussion a little bit.

So one way of interpreting iterative development would be to say in any system, we have a number of technical layers, from the most technical, for example, database or networking to the most user-oriented, such as a user interface.

So one way to say we develop iteratively, which is not the Agile way, would be to say we start with the most fundamental stuff, because it's going to be the infrastructure on which everything else is built.

So we start maybe with a database.

Then we move on to the networking.

Then we move on to the business logic. Business logic means what the program actually does, whatever it is-- managing accounts for a bank, managing documents for a text processing system, and so on.

And then we move on to the user interface.

So that's one way of working iteratively.

And it's not what the Agile view of iterative development is.

Instead, we are talking about its iterations that each produce a working system.

And particularly, we're talking about frequent iterations.

But each one of these iterations is going to have, maybe in a miniature form, the entire system.

So in our example, it's going to have a little bit of database, maybe very rudimentary, or just simulated at first, a little bit of networking, a little bit of business logic, and some kind of user interface.

But it's already going to be showable to and usable by a typical customer, or a test user, or an embedded customer, if we are using the extreme programming technique.

And each one of these iterations includes the entire system—however partial the version may be.

Now, the emphasis here is on **frequent iterations.**

In the early days of Agile, people were saying something like a few weeks

to a couple of months.

A couple of months is really the upper bound.

Typically, Agile projects use two to four weeks-- I should say two to six weeks-- as their standard.

In this course, we have adopted iterative development. And a period is one month, one calendar month, because it makes it easy to remember the January release, the February release, and so on.

But in Agile projects in general, it varies sometimes down to one week or even one day for in the more extreme forums.

The emphasis is on having frequent iterations and frequent partial deliveries.

This notion of frequent working iteration is, of course, part of the mark that the Agile approach has made on the software world.

We no longer have very long release cycles, as we use to have. At least internally, we have typically a few weeks for a release cycle.

Slide 6

Now, there's a rule which goes with this, which is extremely interesting.

It's what I call **the close window rule.** This rule turns out,-- even though it's not that much emphasized typically—it turns out to be one of the major contributions of the Agile approach.

And what it says is very simple.

During an iteration, no one may add functionality. So we take a certain iteration, also known as a sprint in scrum terminology.

And it has a certain list of functionalities to be implemented.

For it could be a task list. It could be a list of user stories, which we're going to see in a few moments, which are elements of the requirements.

So this is what you're supposed to do during the iteration. And of course, we're going to try to do all of it. Maybe you'll only be able to realize part of it. The rule says that whatever happens, no one is permitted to add anything to that list. And it doesn't matter who the person is. That could be the CEO of the company or of the customer company…. If it has not been planned for that iteration, it doesn't go in.

And of course, to be meaningful, the rule has to be applied without exceptions whatsoever. If you start making exceptions, then the whole discipline breaks down.

So why is this a good idea?

Well, for several reasons-- first, it brings order into the process. It means that we know what we are working on and we're not constantly readjusting to new priorities.

Second, it only works, of course, with short iterations of, at most, a few weeks, as we just discussed. Of course, for longer iterations, it would not be sustainable. But here, let's say we have the one month iteration period.

Well, on the average, what's going to happen for a great idea that has just been proposed, if it's just moved on to the next iteration, to the next print, on the average, it's going to be delayed by two weeks, which is usually not a catastrophe. And also, this has the advantage that there's a certain natural attrition of ideas.

And it's very easy for someone to say, I absolutely, I have a brilliant idea. This absolutely needs to be done right now. But maybe when you wake up the next morning and the brilliant idea doesn't look so brilliant after all, or at least so urgent, and so there's a natural phenomenon of sorting of ideas into good, less good, and not really important. So that's one of the biggest contributions of the Agile approach to software development. And I would strongly advise you to use this rule.

Now in some cases, a new functionality really is crucial to react to competition, to react to the needs of an important customer. So then there's an escape mechanism, what we would call an exception in programming.

But it's rather extreme. We cancel the sprint, and we restart a new sprint with a new task list.

So it's possible to do this. But of course, it's sufficiently drastic that people think twice about going into that solution.

The general question is here, how do we compromise? How do we integrate the two kinds of iterative development? Now, the first form where we build the infrastructure first, the essential functionality, is really scorned by Agilists, because it doesn't result in reusable systems right away. On the other hand, it is just good engineering to start by building the infrastructure, the stuff that everything else is going to need and which it may be a very, very bad idea, very dangerous to postpone to the end if it's really so critical.

And on the other hand, the Agile criticism that is justified that people often spend far too much time and energy at the beginning without producing any useful functionality.

Slide 7

So what one can recommend here, although this is not what Agilist do, is to have what may be called dual development.

Early on, despite what Agile texts say, build the infrastructure. Build this stuff that has the most risk and that has to be done really right. But don't do this forever. At some point, someone, like the project leader or the product owner in Scrum, should come in and say, OK, enough guys. We have the infrastructure now. It's good enough. So let's actually produce releases. This actually produce things that users can get their hands on and tell us whether it's what they want or not. So are we shipping yet?

This mixed Agile, non-Agile approach is, I believe, the right way to go.

Slide 8

Next principle, tests-- 7.1, do not move on until all tests pass. This is part of the Agile approach's emphasis on the importance of testing. And the idea here is that we have what is known as a regression test suite. This is a collection of tests that are essential to the project. And one reason they are essential is that all of them, or at least most of them, failed at some point. And we want to check that the failures do not reappear, that the faults do not reappear.

And so the idea of this principle here is that it is forbidden to start adding new functionality

until we are sure that all the existing tests pass. That is to say, all the existing functionality is

reasonably well-implemented as guaranteed, or at least supported, by the regression suite.

So it's a tough rule, because there might be functionality that you think is not so essential.

So we'll fix these later, even though some tests are not passing.

But it's a really good discipline to stay in consideration of practical constraints.

Don't move on until everything in the past works. Don't go to the future until the past is clean.

And that is part of the emphasis on technical excellence of the Agile approach.

Slide 9

There's another aspect to this testing principal—test first, which is also quite important.

But we are going to study it later, so I'll defer the discussion to the lecture on practices, the next lecture.

Slide 10

The final technical principle here is the notion of scenario-- more precisely in the Agile world, user stories. There are other kinds of scenarios-- for example, use cases-- but in the Agile world, it's mostly user stories.

And the idea is that user stories are the main form of requirements, or often the only form of requirements, in the Agile world.

Slide 11

What's a user story?

According to Mike Cohn, it's simply something your user wants.

Examples would be things like paginate to monthly sales report, change tax calculations on invoices, and so on. There's a standard form that has emerged in the Agile world for representing user stories, these pieces of functionality, as a something.

I want something so that something. The first something is a user role-- for example, a customer. The second something is a business functionality, such as see a list of recent orders. And then there's a business justification.

And so a typical example would be as a customer of an e-commerce site, I, the user, want to see a list of my recent orders so that I can track my purchases with a particular company, be it Amazon or anything else.

And the idea of requirements in Agile development is not that you write a requirements document at the beginning.

That was the traditional and scorned approach. It's that you pile up requirements in the form of individual user stories.

And then the accumulation of these user stories at the end is going to give you a system as you implement the user stories. But you produce the user stories as you go, not all at the beginning. And then you implement them right away in the following sprint.

Slide 12

Here's another example of a user story that you can look at in more detail from an article by Waters.

As a registered user, I want to log in so that I can access subscriber content.

And he writes, I would certainly argue it's more easily digestible, these kind of the requirements is more easily digestible, than a lengthy specification, especially for business colleagues.

Well, sure-- does it, on the other hand, make for the best possible requirements to produce a system that satisfies the user needs?

Slide 13

Well, the problem with user stories is that they are examples. So they are to requirements what tests are to specifications, individual examples.

To have a little fun, you can read the following example in this blog post here.

See https://bertrandmeyer.com/2012/10/14/a-fundamental-duality-of-software-engineering/

or http://tinyurl.com/qzyxlal

And the question is, I'm giving you a function with some examples. And I'd like you to tell me what the function is.

So the value for zero is zero.

The value for one is one.

The value for two is four.

The value for three is nine.

The value for four is 16.

So what is that function?

OK, I'll be kind. I'll give you the value for five. It's 25.

And I had some fun in this blog article to plug these values into curve fitting programs.

And sure enough, I find the square function, X square.

But we can also get this kind of function here. Or we can get this really interesting function that fits very well.

So the point of this little fun example is simply to show the obvious. That is to say, examples do not replace a general abstraction. If I tell you this is a square function, I've given you all the possible answers.

If I give you a million values, I've still left a lot of ambiguity. And when we do requirements, we need more than examples. We need the abstraction.

And the task of the requirements engineer is to go beyond the examples and derive the abstraction.

Slide 14

Does it mean that user stories are useless? Certainly not.

They're essentially a tool for validating requirements. If you have more general, more abstract requirements, then user stories enable you like a kind of a smoke test, a litmus test, to check that. Besides having a good general description, the system will perform right in typical scenario. So that is extremely useful, but it's not a substitute for requirements.

Slide 15

What we have seen this segment is a number of principles of Agile development.

Again, the technical principles-- iterative development, the fundamental role of tests-- and there's more on this to come-- and user stories as the Agile source of requirements, also some of the limits of this reliance, this exclusive reliance on user stories for requirements.

Slide 16

In the previous segments of this principles lecture, we studied principles which although they might originally have come from this or that method are pretty much nowadays common to and accepted by all Agile methods.

In the last segment, we are going to take a look at a few principles which are specific to a particular method, in particular Scrum or Extreme Programming.

Slide 17

Complementing the general principles that we've seen, organizational principles and technical principles which are common to all or most Agile methods are a number of principles introduced by specific methods, although they carry a general value and are worth studying.

We will take examples from Scrum, from Crystal, and several from the Lean software approach.

Slide 18

The first one indeed comes from Lean, Mary Poppendieck.

We have already seen that a notion of waste and in particular the notion of fighting waste is central to Lean software. And we have already seen the seven wastes of manufacturing and their counterparts for software, as seen by Poppendieck.

To complement this analysis, here is a more detailed checklist of stuff that constitutes waste in Poppendieck's eyes. And this is the kind of thing to be on the lookout for in software development.

Unnecessary code; unnecessary functionality; delay in processes; unclear requirements that are going to delay you and hence cause waste; insufficient testing, which is going to lead to bugs that surface far too late in the process; repetition in the software process if it could be avoided; bureaucracy that stands in the way of effective development; internal communication that is too slow; coding that is not finished. A piece of code that is almost done is almost as bad as not done at all, because it doesn't produce any working functionality.

Waiting, waiting for processes to finish, waiting for some other team or some other team member to complete his or her work, waiting for other activities, waiting for customers to tell you what you need to know from them.

Bugs, of course, defects, lower quality, and anything managers do that instead of helping actually harms the process.

So this is a quite useful checklist that you can keep to monitor your projects and make sure that they don't produce waste.

Now the reverse of waste is learning. We are going to have after this negative list of things that we don't want, that is to say waste, the other side of the coin, the positive side of the coin, what we try to maximize rather than minimize like waste, and it's learning, learning also meaning information.

Slide 19

Many of the mistakes we make which cause waste are due to insufficient information.

We are in a tunnel looking for the light, looking for the end of the tunnel, and anything that sheds a little light is going to help us. So maybe the appropriate metaphor is a maze with not much light, and anything that enables us to find our way through the maze of developing a software system is going to be welcome.

And how do we amplify and speed up learning? Well, we know that in Agile methods, it's important to run tests all the time.

We're going to come back to this in the study of test-first development as part of practices.

The justification for this in the Lean view is that it helps prevent the accumulation of defects.

Tests give you learning. Tests give you information.

Instead of documentation, which would be one way of producing learning, the Agile view is-- in particular the Lean view is that we should try different ideas by writing perhaps different variants of the code and it may be more effective a use of our time than to write documentation.

Just try different solutions and see which one works best.

Another way of gaining information is to present some results to our users and get their input.

Also, short iteration cycles are justified here by the goal of amplifying learning. That is to say they give you immediate feedback and they provide you some light, helping you to get out of the maze.

Another example is feedback sessions with customers.

So the important notion here is learning. The more we learn, the less waste we are going to produce.

Slide 20

Yet another idea, perhaps more controversial, from Lean is ***decide as late as possible***. So the idea is to delay decisions as much as possible in order to avoid making decisions for which we do not have enough information.

The danger here is to decide, because we have to decide, but we don't have the information that enables us to make the right decision, and then we either have to live with the wrong decision or at a high cost correct the decision, and scale back.

So the justification here, for example, for the iterative approach is that it enables us to postpone some decisions and make sure that when we make the decision, we have the right information.

Now this is in principle a good idea, of course, but we may also note that sometimes it's better to decide something than not to decide at all.

Procrastinating is not a good way of managing a project and the idea of plans, which is much criticized by Agile methods, is in part to help us assess the various elements of information early on so that we can make the decisions and if possible make them early.

Slide 21

An idea from the Crystal method is ***focus***.

Alistair Cockburn tells us that it's critical both for the team as a whole and for individuals in that team to keep focus, meaning not to disperse themselves too much.

So for example, he criticizes the common practice of having developers active on several tasks at once, which decreases focus. So it's better to focus on the individual task of the moment, rather than to do everything at the same time. This ensures flow of progress. And one precise kind of advice that he gives is to deal properly with interruptions.

One of the banes of software development in his view is the constant occurrence of interruptions for developers. And so he suggests, for example, having a rule that there are two-hour periods without any interruption whatsoever. And he also criticizes the current practice of switching developers from project to project or from task to task too often.

So for example, assign a developer to a given project for at least two days before switching. Actually two days doesn't seem that much to me. And here I would go further than him and say make sure that people are not switched from project to project more often than every few weeks.

The focus is also for the project as a whole, and ensuring that everyone has a clear and shared vision of what the project is about and where it's going, so that everyone is so to speak battling in the same direction.

Slide 22

Another idea from Lean and Scrum, ***deliver as fast as possible***.

The view here of Mary Poppendieck is that it's not the biggest that survives, but the fastest.

So the sooner the end product is delivered, the sooner feedback can be received. And she transposes to software the just-in-time production ideology.

Once again, these are ideas that have to be considered with some distance, because the history of software development is also littered with companies that went too fast to market,

like VisiCalc, for example, that opened a market for spreadsheets which was then taken over by much bigger fish.

So be careful here of not just believing a single anecdote like the one that I just gave or any which would go in the other direction. One has in good engineering to weigh the pros and cons, rather than apply simplistic recipes.

Slide 23

Another advice that Poppendieck gives is to ***try multiple designs***. That is to say to have several teams compete.

Well, this is something that has been proposed many times before Agile in software development. And the industry doesn't do it very much, mostly for cost reasons and also because developers tend to fall into the same pitfalls, so you would have to guarantee that the teams are really different. But still it's an idea to consider.

Slide 24

Something more fundamental and highly subject to discussion, going back to our discussion of user stories, is the implicit idea behind the use of user stories as the basis for requirements, the implicit idea that you can actually pilot user stories and obtain requirements.

Now, the promoters of Scrum in particular, Jeff Sutherland, go quite far in this direction and assert it is possible to remove dependencies between user stories so that at any point, any team can select any user story from a kind of queue of user stories, because there are no dependencies between them.

So this if, if you think about it, an essential assumption behind the use of user stories as a source of requirements and this idea that every morning the team is going to-- the first person available in the team is going to pick the next user story.

And this is really, to be quite open about it, more like a fairy tale. So to be more precise, it depends on the kind of system that you have.

Slide 25

Here we can talk of two kinds of complexity, ***additive complexity*** and ***multiplicative complexity***.

We can also talk in more culinary terms of the lasagna kind of complexity and the linguine kind of complexity. These, of course, are only metaphors, but I think they carry the idea well.

So there's a kind of complexity that is quite susceptible to working by subsequent user stories that you just pile up.

This is the lasagna kind of complexity, where the various pieces of functionality are independent from each other. Assume that you have a system that supports a version in English, a version in French, and a version in Spanish. If you're asked to produce an Italian version of it, it's probably not going to be too hard, because it doesn't interfere that much with the existing functionalities.

On the other hand, much of the interesting and difficult stuff in software happens when we have linguine style complexity. That is to say, the various features are so intricately intertwined that it's difficult to touch anything without pulling off everything else.

And this is what makes software difficult as well as interesting. And if you have that kind of feature interaction, the idea that you can just take the next feature, the next user story and work on it without interacting with what already has been done just belongs to the realm of fairy tales.

Slide 26

And we have a very good source for this observation, Pamela Zave, who has been all her career at AT&T studying and building telecommunications software.

Telecommunications software is often feature-based. And she shows the limits of this approach. She says in telecommunication software, feature interactions are a notorious source of complexity, of bugs, of overruns, and bad user experiences.

And she gives many examples.

I'm only including two here, but I strongly urge you to read her article, starting with her web page, which already has a lot of material.

So call forwarding and do not disturb, well, one person has call forwarding and his assistant has do not disturb. And the two features have not been planned together, so someone calls Bob, the call is forwarded to Carol, and Carol's phone, which shouldn't ring does, because do not disturb is not applied to a forwarded call and everyone had missed that.

Or the case of a sales group where the feature is select anyone from the group. But one of the members of the group has forwarded the calls to his personal cell phone and sends a great greeting to the customer which is probably not what the customer was expecting when calling to buy a laptop.

So this is typical of what happens when you try to mix features, just a pile of features which do not pile up naturally.

And this shows that just relying on the succession of stories implementation is just not going to work.

Slide 27

If we had been asked to write the user stories corresponding to this telecommunication example, we might have had a user story which gives the redirection specification and the next one which gives the mechanism, the procedure for busy actions, then one which gives the salesperson interruption mechanism for important customers, and so on.

So each one of these by itself looks great. But then if you just put them together, they just don't make sense anyway and there's no substitute for abstracting from individual user stories and looking for the requirements of the system as a whole.

Slide 28

A couple more principles coming from Crystal, more on the personal sides. Crystal emphasizes the human nature of software development very much.

Personal safety, encourage free expression of ideas, Alistair Cockburn tells us.

Do not ridicule anyone. This is more like good practices in a team.

Slide 29

Humanity, this is in fact from Extreme Programming. It goes in the same general direction.

It's from Kent Beck's book on Extreme Programming.

Recognize that software is developed by people-- this also comes very much from the influence of the Peopleware work by Lister DeMarco, which I mentioned in a preceding segment--so offer developers what they expect, safety, as also pointed out by Crystal, accomplishment, belonging, growth, and intimacy.

So that's an important aspect of the Agile methods, the emphasis on the personal side and the respect due to programmers.

Slide 30

What we have seen in this segment is a set of important ideas from all four example methods of this course, Scrum, Lean, Crystal, and Extreme Programming.

And some of them are minimize waste and maximize learning. The two go together.

Accumulate user stories, again as a mechanism for requirements gathering. And we've seen the really strict limits of that approach, which is highly subject to criticism. It's not among the best contributions of the Agile methods.

And ensure personal safety and humanity, this entire focus of the Agile approach on providing programmers with a safe environment and respecting the personality of individual members of the team.