Lecture 7 (lecture in detail)

Slide 2

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.

Slide 3

The first one, which is not universally practiced, but which you find recommended in extreme programming, is the idea that **only one pair integrates code at a time**.

So you can view it as a principle as much as a practice.

But the idea is that in this context in which we have collective code ownership and various developers or pairs of developers proceed in parallel, we want to avoid conflict.

So only one pair is permitted to integrate its change, its changes at any given time.

Slide 4

More fundamental is the idea of **continuous integration**. So it's the combination of frequent releases, which we're going to see now with relentless testing.

So an XP team tests and tests and test again all the time.

It's testing really that drives the process, as we're going to see in the next segment.

And the idea is to keep system fully integrated at all times.

Hence, very frequent, also known as continuous integrations. How frequent? That's what we're going to see next.

Slide 5

So rather than weekly or daily builds, the idea, when taken to the extreme, is to build a system several times a day with a number of benefits such as easier integration because little has changed.

The team learns more quickly.

You minimize unexpected interactions, and a problematic code is more likely to be fixed because more eyes see it sooner.

And duplication, also which is, of course, a very bad thing in software, any occurrence of copy/paste is detrimental to the future of the software.

Such duplication is going to be detected earlier, and hence, easier to eliminate.

So this whole idea is to release early and often.

So we're talking for us here about two different things.

Integration is internal to the team and should, in the extreme programming view, occur as often as possible.

Slide 6

Release is something that's going to be visible to the outside. And here, too, we want to **release early and often.**

To use the famous joke about voting in Chicago, this, of course, follows from the rejection of bigger front design and bigger front requirements.

If you are going to have a process that is continuous in which you do the requirements and the design as you go, well, the best way to see-- in fact, the only way to see that you are actually meeting the customer's needs is to produce frequent releases.

Slide 7

In XP teams, the idea is also of **small releases**. So they are small according to this discussion by Ron Jeffries, which I'm quoting pretty literally here.

There are two notions of small.

First, we release running testing software at every iteration so that a customer can use it for any purpose, including actual use.

So this is pretty risky, of course. And it's also frequent as well. For example, where projects may need to release as often as daily.

Slide 8

Deployment should be incremental too.

So the functionality should be deployed gradually, avoiding big bang deployment.

Many projects run into this typical pitfall of working very hard for weeks or months and then releasing the entire system.

And then, of course, in a complex system, there's a good likelihood that something somewhere, even a small thing, will go wrong and will hamper the entire deployment.

Well, by having incremental deployment when you release functionality, step by step, is a guard against such problems.

Of course, it's not always possible if you are launching a rocket.

Well, the software must work the day you launch it.

If you're introducing the euro in 2001, well, you'd better have everything ready the day the new ATM's go into operation.

But in many projects, especially in projects in which the software gets updated regularly, it's possible to have this incremental deployment.

Slide 9

In fact, some projects practice **daily deployment** with the idea that, of course, if anything doesn't work, the developer who is responsible for that part is going to have to come back and work until it works again.

Slide 10

Some authors, like Shore, advocate a 10-minute build. This is a kind of rule of thumb to make sure

that integration proceeds smoothly.

So the rule is that it should be possible to integrate everything and to build everything in ten minutes or less. And this includes lots of stuff, compiling, running the tests, registry settings and so on.

And the idea is that if it takes more than 10 minutes, then the project is too big and should be split into sub-projects.

Slide 11

Now, how long should be the release cycles? Well, here, as we have already seen, the methods and the individual projects vary.

Kent Beck advocates a **weekly cycle**.

Now, I'm not going to go into the details of the steps that he advocates. I'll leave them for you to read if you're interested because this is somewhat superseded by the more detailed steps and advice that we've seen in connection with scrum, with the scrum planning meeting and review meeting.

This was a more detailed and, in particular, in its distribution of roles between the product owner and the team.

What's interesting here is the emphasis that Beck makes on the period of one week.

And he says the nice thing about a week is that everyone is focused on having the test run on Friday.

So he really means a calendar week.

So if you get to Wednesday and it's clear that the test won't be running, you still have time to choose the most viable stories, remove the others, and complete those that you have chosen.

Slide 12

And Kent Beck also advocates for a quarterly cycle with a review of the high-level structure at that stage.

And as the XP authors emphasize, this often corresponds to the financial reporting practices of many companies.

It's also a period which is chosen as being large enough not to interfere with current concerns, and also short enough to allow frequent questioning of the practices in the CMMI style of a self-improving process.

Slide 13

So what we've seen in this third segment of the practices lectures is that Agile methods in many different ways promote frequent release cycles and continuous integration.

And there are a number of variants, such as weekly, quarterly, and in between.

And, of course, each project should choose for itself what exact period suits its needs best.

Slide 14

Agility fosters excellence. Agile processes focus on quality. And the Agile focus on quality is largely achieved through focus on testing.

In fact, in the history of software engineering, Agile processes have played a major role in rehabilitating testing and placing tests at the very center of the software process as one of the two core software artifacts.

The other one being, of course, code.

Code and tests.

These are the two core artifacts in Agile methods.

In this segment, we are going to study how Agile methods promote quality and in particular what role Agile methods give to testing and how Agile development affects the testing process as well as the other way around.

Slide 15

The Agile worldview places a very strong emphasis on technical excellence and particularly on the quality of the produced software. Much of this view of quality is based on testing.

And most of the practices that we're going to see now are focused on testing.

Most of them also originate, like those in the previous segment, with extreme programming, which is the most technically-oriented, the most software specific of all the Agile approaches.

The first practice is not specifically related to testing.

It's the idea of **coding standards**. So it simply tells us that, for each project, there should be a set of conventions, style guidelines, style rules, and such which project members should adhere to.

This is hardly revolutionary and hardly new with XP.

But it was worth repeating.

Slide 16

Now we come to testing.

And first we come to one of the principal contributions of extreme programming to software engineering, the idea that we should **never write code without associated unit tests**.

So the idea here is that whenever you write a piece of code, you should also have a piece of test that goes with it. A test or a group of tests that exercises it.

Now the question arises immediately of the order in which we are going to write the code and the test.

Let's defer that question to the following slides.

But regardless of the answer to that question, one of the main contributions of the Agile view of software development-- specifically extreme programming-- has been this idea which has really been widely adopted by many teams, whether or not they apply other Agile techniques.

Do not ever write code without immediately having the set of tests that goes with it.

It's a very sound practice.

Widely adopted today.

Deserving to be even more widely adopted.

And one of the major contributions of the Agile movement.

Slide 17

A related practice which is perhaps a bit more difficult to apply is the practice of making sure that all tests pass before the team moves on.

So this was a principle in our list of the technical principles of Agile development in here.

And it gets translated into a specific practice. Do not move on to, for example, the next user story to be taken on by the same developers or for the whole team and more ominously to the next release.

Do not move on until all the current tests pass.

Now this, of course, is hard because sometimes you might have the temptation to say well, OK.

These are almost all the tests passed.

Can't we move on?

We need new functionality.

And the strict Agile view here is to say no.

Until all the tests pass, you're not permitted to move on. Because it decreases the quality.

And if a test is not so important, then remove it. And remove the corresponding functionality. But you should not continue until you have the expected level of quality.

Slide 18

Going even further, we have the notion of test first developments which really comes from extreme programming.

And it is one possible answer-- one of the two possible answers-- to a question that I left open a moment ago.

If with every piece of code there is a test or several tests, in what order do we write them?

Well, test first development says we should write a test before we write a code.

And the test in this role becomes a replacement for the specification in the same way that we replace requirements by user stories at the level of overall system description.

Then at the level of coding, at the level of the code in the same way, we replace specifications-- traditional specifications-- by tests.

Slide 19

So here's how Ron Jeffries, one of the original main voices in extreme programming, describes this process.

Here's a really good way, he says, to develop new functionality. Find out what you want to do.

Write a unit test for it. Run the unit test.

If it succeeds, you're done for that step.

Go back to step one.

If not?   Well, fix the problem. And try again. And he says, try it.

If you like it, he cites a number of arguments for this approach.

Guaranteed flow.

It goes faster, and so on.

This is not, of course, a scientific demonstration.

But enough people have tried it. And enough of these have liked it that test first development has gained some traction.

Slide 20

It actually can be generalized into an entire development method, a unified development method known as test-driven development coming from Ken Beck who wrote a book specifically devoted to it.

And here we have the entire development method. So what Ken Beck tells you-- and it's quite a challenging view.

Of course, it's very extreme. And it doesn't necessarily apply to every kind of software development.

But it's certainly intellectually challenging. He tells you here's a way you should develop software.

Add a test. So the first time around, that's all you have.

Right? You have one test.

No program.

Nothing.

No code.

No specification, of course.

No requirements.

Just a test.

Run all tests.

The first time, of course, there's just one test.

But as you apply this process, you're going to accumulate tests.

This is a cycle, right? You are going to repeat these steps.

So run all the tests that you have so far.

And see if the new one fails. Really, it's see the new one fail. Because you expect it to fail.

After all, you have not written any code corresponding to that functionality.

So how could it be except by magic or by miracle that the code succeeds?

So in that case you expect it to fail.

So write some code is the next step to make sure that the test now passes.

Run the automated tests.

And see them hopefully succeed.

If not, of course, fix the problem so that there might be a better loop here.

And refactor code.

So this last step, of course, is critical.

Because if you didn't refactor, you would have very piecemeal code, right?

If case number one didn't do production of result number one.

If case number two produced result number two and so on, this would not be a very good code.

So this is where refactoring as studied in the previous segment is critical to make sure that we keep the long term perspective.

Now advocates of this approach cite many benefits for it.

Catching bugs early.

Getting more tests in the end which, of course, is good.

As we're going to see, the regression test suite is one of the prime assets of the project. Obviously it is not appropriate for more complex programs which require some abstract thinking ahead of time at the beginning of the project. But it's certainly an interesting idea which has been quite influential.

So the major benefit is the regression test. What's the regression tests, or specifically the regression test suite? It's the accumulation of tests that failed earlier.

It's one of the major assets of a software project. It turns out-- and this is a phenomenon that is probably specific to software-- that all bugs have a tendency to come back.

And having a regression test suite makes sure that the hydra doesn't pop a new head all the time.

Or at least if it does, you fix it right away.

Slide 21

So the general idea is that whenever you have an important step in your project-- at least before any release but probably before every integration as we saw in the previous segment— you complete your regression test suite.

And you correct any bug that has reappeared.

There's also a variant of TDD which is a bit more specific and which I invite you to look up in the bibliography.

It's called **contract-driven Development**, which somehow reconciles test-driven development with more standard ideas of specifications.

Slide 22

A practice that goes with the previous ideas—maybe it's halfway between a practice and a principle--is the observation that a bug is not an error in logic.

It's a test that you forgot to write. So in this slogan, there's something interesting which is somehow putting bugs in a proper place in software development.

Bugs are not just something shameful that you try not to mention.

They are a part of the software development process.

And they are closely connected with tests.

So in the extreme programming view, a bug is simply a test that you forgot to write.

So it's not such a big deal.

You write the test.

You fix the code, of course.

And then the test-- the bug, rather, becomes history.

Except that it is in this regression test suite.

And I cannot enough emphasize the benefit of making sure that any bug that has ever been found in the development of a software system finds its place in the form of a test in the regression test.

Because as I emphasized, in software development all bugs have a tendency to come back and haunt you because of shared psychological modes of thinking for programmers and other reasons.

So you need to have this regression test suite.

And it's to the credit of extreme programming and Agile methods that we have put with this notion of the regression test suite at the center of our concerns.

Slide 23

They also tell you, particularly in extreme programming in Scrum, that when you have a bug, you should **not just fix it but analyze the causes**.

So see that we have now changed the viewpoint, because earlier in test-driven development, we were talking about tests that we expected to see failing.

Here, we're talking about tests or executions that we did not expect to see failing but which did fail.

So these are bugs that are found after the fact.

And they should become part of the regression test suite.

And what the promoters of this idea are telling us here is that you should not just be content with fixing the bug.

You should go deeper into the root causes to make sure that you are not just fixing the symptom but fixing the cause and, in particular, helping make sure that it doesn't happen again.

Slide 24

There's also another rule which has to do not with unit tests as the previous discussion was, but with **acceptance tests**.

An acceptance test is a test that is meaningful to the customer and which is applied typically to the system as a whole, not just to a part or unit of the system like a class, for example.

The advice here, the practice is to have these accepted tests organized carefully. Automate their running so that you can run them often.

And publish acceptance test score to the team so that everyone can know what's going on.

Slide 25

What we have seen in this segment is that in extreme programming at least, and in many applications of Agile methods, it's the tests that really drive the development.

Code follows test rather than the reverse.

All tests should pass.

And test really replace specifications.

This is, of course, a set of ideas that are somewhat extreme and controversial. What is not controversial is the central role of tests in software development, the need to have tests associated with every single piece of code, and the central role of the regression test suite.

Slide 26

We have now seen some of the most important Agile practices. But there's a few more, in particular, **managerial practices**.

And, in this fifth and last segment of the practices lecture, we are going to study a few management oriented practices which complement the set that we've seen so far.

To wrap up our study of Agile practices, we look at a number of practices which are essentially management practices. That is to say, they help us manage and organize an Agile project.

Slide 27

Perhaps the most important is the first one, **the Scrum of Scrums**, because it addresses the issue of scaling up Agile methods.

Agile methods have often been viewed as addressing only small projects.

In fact, the creators of Extreme Programming, which was one of the first Agile methods to come out, readily acknowledged that it was mostly for small projects.

But as Agile methods have been more widely adopted, many people have tried to see how they could be applied to bigger projects beyond the seven plus or minus two ideal size that we mentioned in an earlier lecture.

So Scrum in particular came up with the idea of Scrum of Scrums. ***Scrum of Scrums, as the name suggests, is simply a bigger project organized as a collection of ordinary Scrum projects.***

And what happens is that there is an extra meeting to coordinate between these various Scrums and these various projects each day after the daily meeting, also known as the daily Scrum.

So in that meeting, clusters of teams discuss areas of overlap and integration.

A designated person for every Scrum project attends. And the agenda is the same as the daily Scrum.

Remember the three questions.

What did we do yesterday?

What are we going to do today?

And what are the impediments?

But here, there are four additional questions, which are the ones relevant to the coordination between individual projects.

What has your team done since we last met?

What will your team do before we meet again?

Is anything slowing your team down?

So again, emphasis on impediments.

And are you about to put something in another team's way?

So the idea here is to remove impediments even before they arise by making sure that if there's a dependency between the work of a team and another-- and in particular, if there's a potential for trouble because of what one of the teams is going to do-- it's known in advance so that it can be handled properly.

Slide 28

The next practice coming from Extreme Programming but also widely adopted in Scrum is what we may call **Whole Team.**

And we've seen some of this already when talking about cross-functional projects.

All contributors sit together. It's this practice of XP as members of a team.

So there is always a business representative. This is somehow the XP idea of the embedded customer, but programmers are there, possibly testers, analysts, and a coach or a manager.

So here, we're kind of in a pre-Scrum world in which the roles of Scrum Master, Product Owner, and so on have not been that precisely defined.

We are in a broader context.

But of course, this is easy to transpose to the exact context of a Scrum project with the Scrum roles that we saw in the last lecture.

The important point here, regardless of the method details and the role details, is to make sure that we bring everyone under one roof, in particular representatives of the various stakeholder categories, developers, and people representing customers.

Slide 29

There's a technique here which really belongs more to the folklore, but some people take it seriously. To me, it's really part of the folklore of Agile methods is **Planning Poker**.

So it's a technique which is used in Scrum in particular to estimate the duration, meaning the effort and the cost, of any particular task.

Of course, various people may have widely differing estimates in the Planning Poker, which is a variant of a technique known as Wideband Delphi.

It is based on cards. So you can buy these cards from various providers. And each card represents a duration.

So typically, it's in days. It could also be in half days or possibly in hours. But usually it's days, with perhaps one in half days. So it could be like zero, one half, one, two, and so on, except that we don't want to have people haggle over this task is going to take six days.

No, it's going to take seven days.

These are too small differences.

So typically, people use a sequence such as the Fibonacci sequence, which has the advantage of growing much faster after the first few values so that we don't argue or haggle over small differences.

And so the idea works like this.

We are taking a certain task.

We are a group of people.

And each one of us is going to have his idea of how long the task is going to take-- a half day, two days, three days, whatever.

So we think very hard to determine that idea without immediately revealing our choice to the others.

We may talk, of course. But we each decide for ourselves.

And then at some point, we reveal our cards. We show our estimate.

If the estimates of everyone in the team are the same, this ends the process.

That is going to add value, is going to be the estimate.

Otherwise, we talk.

We try to justify our respective assessments. And then we play again until we all agree.

This is rather simplistic because task estimation, software cost estimation, is a difficult area.

And there is a really strong chance in a process like this that the loud talkers, not necessarily the most experienced people, are going to influence the others who just want to move on, especially if they do not have themselves to implement a task.

But I give you this technique for what it's worth since it is widely applied and shown as an example of Agile, and specifically Scrum practice.

Slide 30

We come now to **workplace practices**. Agile methods have put a lot of emphasis on the role of the workplace.

And basically, they want to have workspace that enables everyone to communicate with everyone else.

The emphasis is on communication.

Now, one thing that should be said here at the outset is that workspace is important. But after all, some of the most successful tech companies were started in a garage.

And sometimes, there's a little bit too much of emphasis on this aspect in the Agile method.

So Extreme Programming and Crystal and others emphasize the need for ***an open workspace which***

is organized around pairing stations.

If we do pair programming, as in XP, with lots of whiteboard space, this is certainly a very good idea.

We need to have ways to communicate, to discuss our designs.

And people are locating according to conversations they should overhear.

That is to say, is it good or bad that team A will be able to overhear what team B, two or three people are discussing?

This is going to be the primary criteria.

And then, of course, there should be a place for private conversations.

Now, this is, again, part of what we could characterize as the folklore.

It could even be detrimental because not all programmers like to communicate.

And you have excellent programmers who work best by concentrating in a quiet space.

Of course, this would be anathema to the Agile view of pair programming all the time.

But it is more important to recognize the diversity of personalities and to avoid imposing a standard communicate, communicate, communicate pattern on everyone.

Slide 31

Crystal emphasizes, in the same spirit, the notion of what Alistair Cockburn calls **smarty communication**, where the team is together in a room and listens to each other.

Information can flow around.

A developer must be forced, if necessary, to break his or her concentration.

Questions are answered rapidly.

This is all in line with, as you may remember, what the lean method called amplifying learning.

This was one of the lean principles.

And of course, the faster you hear about what's going on, the faster you get information, the more effective your software development is going to be in this view.

On the other hand, again, this should be taken with a grain of salt because software development is a highly demanding intellectual activity, close in many respects to mathematics.

And sometimes, you just want to shut yourself off and think, even if this is not viewed as the optimal in communication according to the Agile view.

Slide 32

The workspace should not only be open and osmotic. It should also be **informative.**

And here, we have a set of practical very simple but very useful ideas.

Make sure that communication is facilitated through techniques such as a **storyboard**. We're going to study the storyboard in connection with the artifacts the next lecture.

A storyboard is a place where you record the progress of ongoing tasks and ongoing user stories.

Release charts, iteration burndown charts. We will also study the burndown charts in the next lectures, various indicators showing the status of the latest test, the latest release.

This is all quite important because it boosts team morale.

Slide 33

When you see what's going on, even if you're stuck yourself, if you see the progress that has been made, it really can help.

And this part of the advice and this part of the practices is very good.

Also, a piece of advice which should be obvious, but which for many nontechnical managers is not, is given in Crystal.

Give developers the tools that they need. Don't skimp on things like memory.

Give them the access to all the tools that they need to be effective software developers.

Slide 34

Extreme Programming emphasizes the need for **team continuity**.

Try to avoid moving people around teams all the time. Keep the team together and stable.

This is, again, halfway between a principle and a practice.

Slide 35

Perhaps a more controversial principle propounded by XP is **shrinking teams**.

The recommendation here is that as a team grows in capability, keep its workload constant but gradually reduce its size so that you can have a free forming set of teams.

And when the team has too few members, merge it with another too small team.

It's interesting to pictures this kind of game of life organization of teams.

I'm not sure how many companies can really practice this.

Slide 36

Another practice which really follows from previous principles is **to maintain only code and tests as permanent artifacts**.

This is, again, somewhat extreme. But it's quite in line with the lean view that we've seen, that in the end, only deliverables matter.

Slide 37

In line with the Extreme Programming emphasis on **having an embedded customer**, there is this general recommendation in the Agile world of involving customers throughout.

So in the XP style, which is representing here, this is embedded customer.

The idea is more general.

It can be applied also in the context of Scrum with the product owner.

The role of customers can be to help write user stories, to help make sure the design functionality is there during the planning meeting, to negotiate the selection of user stories for release.

In Scrum, this is not going to be any customer. It's going to be customers represented by the product owner.

The customers, of course, should have their say, again, represented by the product owner in Scrum in the definition of the release timing, which should be a negotiation.

And they should be involved in anything that has consequences on business goals, which is actually a lot.

And Wells justifies this by saying that this may seem a lot of the customer's time.

But it's going to be saved by all the mistakes that we avoid making by having a well-specified and cooperative system.

Slide 38

Finally, there is a principle that we find in Extreme Programming, which is **Slack**.

So again, one can be a bit doubtful as to how many companies will be willing to apply this.

It comes from a book with the same title by Tom DeMarco, the same author as Peopleware, which was also very influential on the Agile school.

And DeMarco advocates for not being too tight, not filling every second of your workday, but leaving some room.

And according to Extreme Programming, this is an important to practice not to feel everyone's agenda, but in any plan including some minor tasks that would really not be essential and can be dropped if you get behind.

And the goal is to establish trust in the team's ability to deliver because it's always dangerous to have a plan that is too strong, too ambitious, since the risk is high that you will not entirely fulfil it.

So the team will get frustrated and morale will go down.

If, on the contrary, you put in little stuff here and there that is not essential, well, the essential goals will be achieved.

And this is good for the morale. And of course, we want to reduce waste.

Slide 39

What we have seen in this final segment of the practices lecture is a number of management practices that all are there to support the Agile principles of the previous lectures, in particular by respecting developers, by ensuring that measurement gives developers the support and confidence they need to apply the principles and practices of Agile development.