Lecture 9 (lecture in detail)

Slide 1

We have now seen the essentials of Agile software development.

And as you have seen in the previous lectures with the ***values, principals, roles, practices, and artifacts of Agile software development***, you will have noticed that there are some really good ideas and some ideas which-- to put it politely-- are perhaps not so good.

So we need to perform an assessment. Of course, in the end it's your assessment that matters.

You have to decide what among the Agile ideas are the ones that are right for you and which ones are not so good for you.

But there are also some objective criteria.

After all, Agile development didn't come into an empty world. We have had several decades of software engineering before. So we have some empirical studies.

And we have some analytical criteria to help us judge.

Slide 2

In this lecture which, like the previous one, is made of just one segment, we're going to perform this assessment going in the reverse order of the title of the textbook.

That is to say, we're going to start with the ***ugly, then move on to the hype, the good, and the brilliant***.

First, **the ugly**.

Unfortunately there are a number of Agile ideas which are really, really bad and which you have to stay away from. And we will analyze those.

Then we will continue with the **hype**. Those are ideas which have been hyped a lot. But they're really indifferent.

That is to say, they're not going to affect your success very much one way or the other.

And then of course, the good news is that we have the **good** and even the **brilliant**.

We have a number of ideas that are really, really useful and which have earned Agile methods their well-deserved fame.

And so in the end, it's for you to decide. But it's really important to know which ones of the ideas are really best avoided and which ones can help both the quality of your software process, and the quality of the software that you're going to produce in the end.

In this final assessment of the merits and limitations of the Agile approach, I'm going to refer to the terminology of the title of the textbook.

As you have seen, it's called ***Agile, The Good, the Hype, and the Ugly***.

So it sorts out the wheat from the chaff.

And we are indeed going to use these three categories slightly modified. But of course, we want to end on a positive note.

Obviously if Agile did not have a number of major positive contributions, we would not have been wasting our time, you and I, studying it in detail.

So we want to end with the good.

So we're going to start with **the ugly**, the ideas that really harm software projects and which we should not apply.

We're going to continue with the **hype**, or the indifferent as it's called on the other slide. Those are the ideas that don't make such of a difference, however much they might have been hyped.

And we're going to end up with the good actually split into two categories. **The simply good and the truly brilliant**. And so, we will end on a high note since it's really important to understand how Agile processes can fundamentally improve software development.

Slide 3

Well, not everything is going to improve software development. And the ideas on this slide--and you can see there are unfortunately quite a few of them, are guaranteed to harm your development.

So whatever the hype around them, whatever people say, however fashionable it may be to say that you're applying Agile methods, do not apply these ideas.

They are going to damage your software process. And sometimes it might even lead you to failure.

The first one is **the rejection of a front task**….The general idea that we should not do in particular up front requirements or up-front architecture or design.

This is really bad as we've seen. There is a grain of truth there.

There's a reason for this Agile criticism of traditional approaches which sometimes do spend too much time early on looking at a problem.

There's this expression that you may have heard….Analysis paralysis.

Well, that exists. And you also remember from the discussion of principles that it's good to have dual development, meaning that at some point, you stop looking at the problem and you jump into the solution. But only at some point…

And the idea that we should jump right away and start coding is absurd. And it is guaranteed to damage your process as I've seen again and again with failed or delayed projects simply because they had refused to take the few weeks or the couple of months necessary at the beginning to analyze the problem and define the architecture which is, of course, what every engineering project should do.

In engineering, regardless of the discipline, you first analyze a problem to understand the problem as a whole, not just pieces of it. And you define the overall structure of the solution. So that's ***requirements and design***. And you should do this in software, too.

Another really bad idea is **relying on user stories as a replacement for requirements**. User stories as we've seen have their place. But user stories are specific. They describe individual interactions with the systems. And it is the mark of the professional to be able to abstract from these individual cases and describe the purpose of the system in more general terms. User stories can serve to validate these requirements. But they are not a substitute for them!

In the same vein, **tests are useful and very important in the Agile approach**. As they should be. But they're **not a replacement for specifications**.

A test gives you a result in one case.

A specification gives you the result for all possible cases.

And only the intellectually lazy replace this analysis by just an accumulation of tests.

Feature-based development. The myth that you can build a system by accumulating feature after feature, ignoring dependencies or denying that dependencies exist. Well, as we saw in the citation from the work of Pamela Zave, this is completely make believe. This is a Santa Claus development in practice. Any nontrivial system is characterized by dependencies between the various features which we must analyze as part of up-front requirements and up-front architecture. Ignore them at your own peril. The result is only going to be that further on in the development of the system, each time you add a new feature, you end up re-doing everything else that you had done before because of the linguini-style interaction between features.

The idea that perhaps it's not as bad but it's also pretty bad, the idea that the method keeper-- the political commissar as I called him-- the Scrum Master should not develop code. We want doers. We don't just want talkers.

Test-driven development as a replacement for analysis and design-led development. So this is not a dismissal, of course, of the Agile contribution on testing which is very important. But test-driven development in the narrow sense of the term is far too restrictive.

Dismissal of the traditional manager tasks. Many successful engineering projects are successful because of a strong manager. And to pretend that every team can be self-organized and will have better results if it's self-organized or just self-organizing than if it has a strong manager is extremely detrimental.

The dismissal of auxiliary products and non-shippable artifacts. As we saw here, too, there is a basis for this attitude. It is true that, especially in companies with very heavy processors, too much effort is sometimes spent on things that are not delivered. But the rule that we should only work on code and tests is far too extreme. It's also a somewhat Luddite rule. All engineering relies on auxiliary products.

We could, in construction, in building construction say, well, the scaffolding is not going to be delivered to the customers. So we don't need scaffolding. This would be absurd.

And the counterpart for software is just as absurd. So don't believe the narrow-minded view that intermediate artifacts are not relevant.

And the **dismissal of a priori concern for extendability and reusability**. Here we've seen that the best Agile authors have a kind of mixed attitude. But it still is a very strong philosophy that we should do just the projects that we need, just the program that the customer has requested and not think too much about future reuse and future extensions. Of course, that is bad engineering…

It's somehow compensated by the emphasis on re-factoring, which is very healthy. But you still have this idea. And of course, it is the mark of the good engineer that he or she produces stuff in particular for software engineer programs that are ready for extension and reuse.

Slide 4

Next we come to **the indifferent**. That is to say, ideas that may not harm that much but also are not worthy of the hype that they have often received.

Pair programming is an example. It's actually a good idea in some cases. It can help. And it's widely applied. But it's not justifiable as the only way to develop software.

Open space working arrangements? Then, of course, it's good to have well-organized space. But as was pointed out, many of the most successful companies started and continued for a while in a garage. And great software can be developed in not so great environments.

Self-organizing teams. Well, not all teams can be self-organized. And if you have a great manager, just take advantage of him or her.

Maintaining a sustainable base? Well, of course we all want these. But the world is what it is. And when a customer really wants a major delivery, there is really not that much one can do.

Producing minimal functionality. Also same observation. Ideally, of course we all want software systems that are simpler. But your essential feature is of no interest for me. And conversely, if a major customer absolutely wants some functionality, we have little choice but to implement it.

Planning poker. This is fun. And I described it as part of the agile folklore. --- (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.)------ It makes for a great interaction in Scrum workshops. But it really doesn't go very far. And it's not a substitute for more systematic, more scientific ways of estimating functionality.

Cross-functional teams. It's in general a good idea to have lots of knowledge shared in a team between the team members. But it's inevitable that some people will be experts in some areas. Of course, you want to avoid that a project is fundamentally dependent on one person. But you cannot ignore this phenomenon of specific expertise.

And **embedded customer?** Again this is a pleasant idea in principle as introduced by extreme programming. But it doesn't really work in practice. And the scrum product owner concept is superior.

Slide 5

And now we come to **the good**. So there is actually quite a few items again.

Acceptance of change. Although as we've seen, there is not that much technical support for this idea. What is important here is that Agile methods have really convinced the industry that change in software, the soft part of software was not something to be feared but something to be accepted. A normal part of the software development process and something that can even be turned into a competitive advantage.

Iterative development. Of course, the idea that we develop and deliver functionality step by step. The emphasis on working code. This is not to be understood as a rejection of other products like design or analysis or requirements. But it's is very healthy to remind the world that, in the end, it's not diagrams that are going to gain us customers and success. It's working programs that has been one of the contributions of the Agile approach.

Tests as one of the key resources of the project. Again, a major contribution of the Agile approach to rehabilitate the notion of test and convince us that our regression test suite--which we run again and again and again—is one of the key assets of the project. Essentially as important as the code base.

This is a major advance in software engineering.

The notion of velocity. Of course, it's still fairly primitive. And I'm sure better measures will appear in the future. But it's a first concerted effort to quantify the notion of progress in a software project, giving it numerical measures.

No branching. Not everyone agrees with this. There are people who claim that with modern configuration management tools, branching is OK. Here, in fact, the Agile authors are right. Branching is extremely damaging. It causes far more problems than it solves. And it's much better, at some expense of course, to retain a single branch in a software project.

The idea of **the product burn down chart.** Not so much for user stories because of the general problem with user stories but for the product as a whole. It's, of course, the way to visualize the notion of velocity. And it's extremely useful for software projects to be able to see day after day how they are progressing in a visual form.

And **the daily meeting.** As we noted, this also is not perfect. It needs to be adapted to the needs of distributed projects into modern forms of working like telecommuting, for example. But the general idea is extremely healthy, of a short meeting in which we review what is going on with the three questions that we have seen that really make a fundamental difference in the way we develop software.

Slide 6

We have **iterative development**. But it's not just iterative development. It's that **the iterations are short**. So this has been a major change in the way the industry develops software, quite differently from 20 or even 10 years ago. And that is a distinct contribution of Agile.

The close window rule. This is the idea that, during an iteration, it is not permitted to anyone regardless of status to add any functionality. This is absolutely brilliant, in connection of course with short iterations. In particular, because it helps a natural phenomenon of filtering and attrition of supposedly great ideas.

Refactoring. It should not be taken as a substitute for design. But the idea is excellent that whenever you have a first solution, it may not be the final word. And it's necessary to take a second look at it to see if it can be improved for the future.

The rule, the absolute rule that no piece of code, no piece of functionality **ever comes without a test**. This is a simple practical rule which has the potential of affecting the quality of what we do in a fundamental way.

And the notion, of course, of **continuous integration**. Integrate. Integrate. Integrate every day.

Slide 7

So as final observations, there has been a tendency in the Agile literature to promote Agile methods as a panacea, as a marvelous recipe. And some of the productivity improvements that are announced are just fairly crazy, I should say, and not necessarily born out on a large scale in practice.

It doesn't mean that we should dismiss Agile methods in general, especially as a result of the preceding review. It's just that it is not necessarily a paradigm shift. It's a major improvement.

But it doesn't displace the previous ideas. It complements them. Software development is hard. And what really counts in the end is quality. Anything that helps produce software of better quality is going to be helpful.

And Agile methods have that potential. Lots of good ideas. Agile in that can help. There is no reason to reject those from any particular style of software engineering.

Particularly since we should be quite humble there since there is not much empirical data from impeccable, unimpeachable studies that show that technique A is better than technique B. We have very little of that in our studies but not enough. So we should be very careful in our claims.

And in the end, Agile is not a replacement for 50 years of evolution of software engineering. It's a complement for those decades and all this accumulated wisdom.

Slide 8

So as a conclusion to this lecture, which is also a conclusion to the entire course, Agile is a mix of good and bad ideas, some of which are what you'd say very bad but some also absolutely brilliant.

And these are the ones that justify studying Agile methods and practicing Agile methods in depth.

And whatever its limitations, whatever the criticisms that one can make, it's a major step in the evolution of software engineering.

And I hope you can benefit from the lessons of this course to make your own software development better and to take it to the next step..