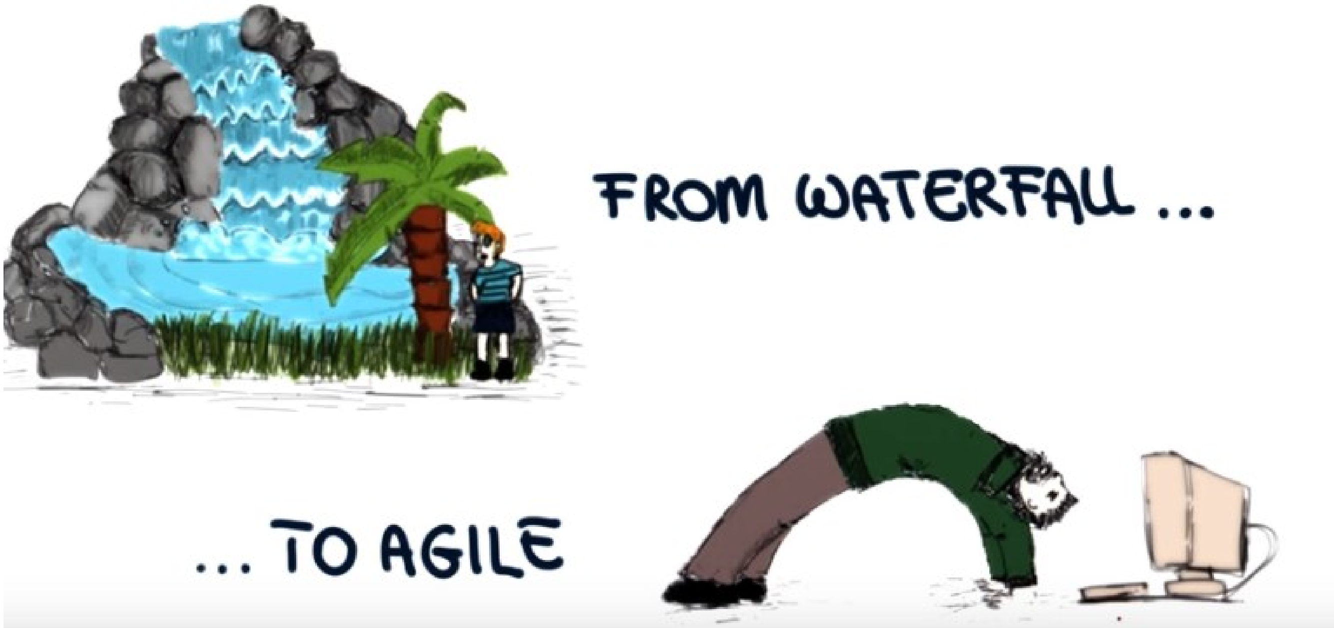
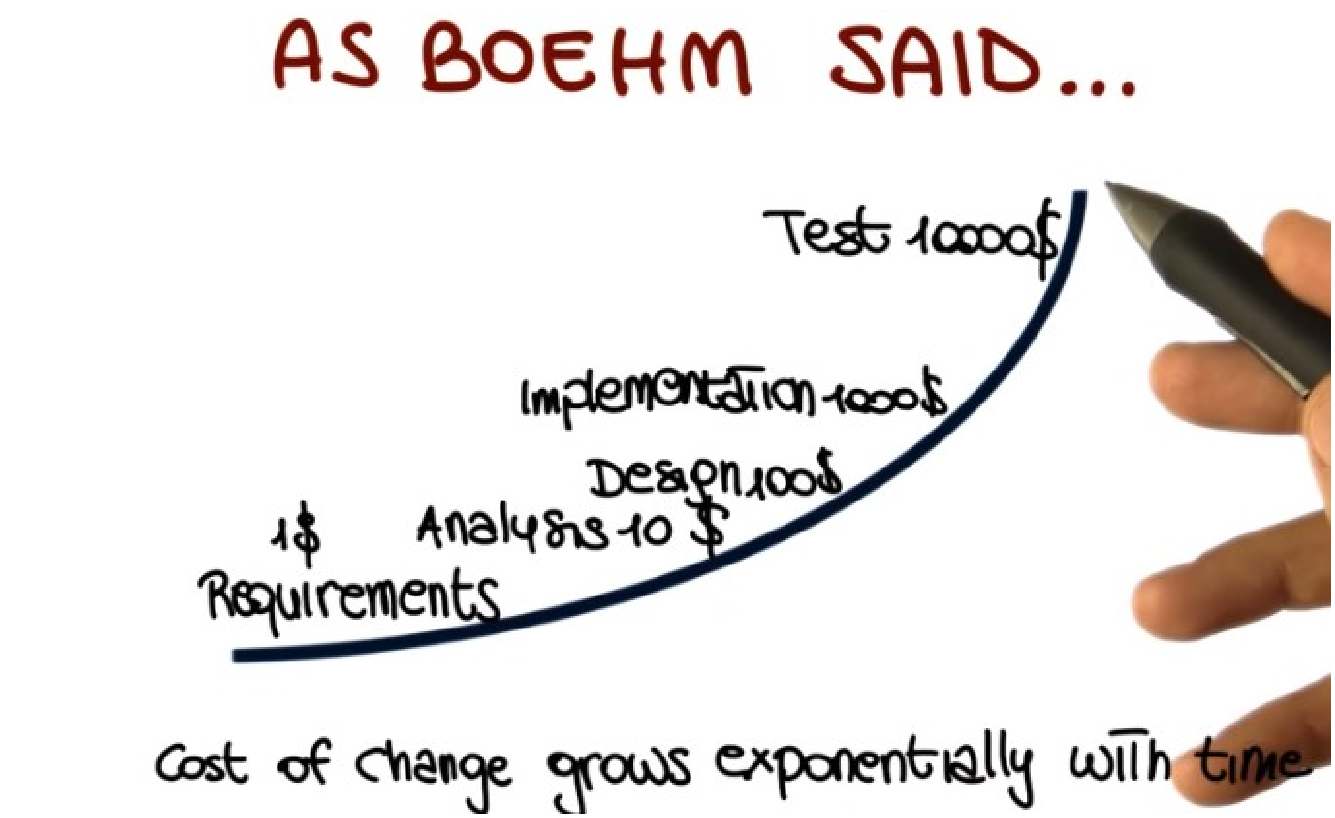
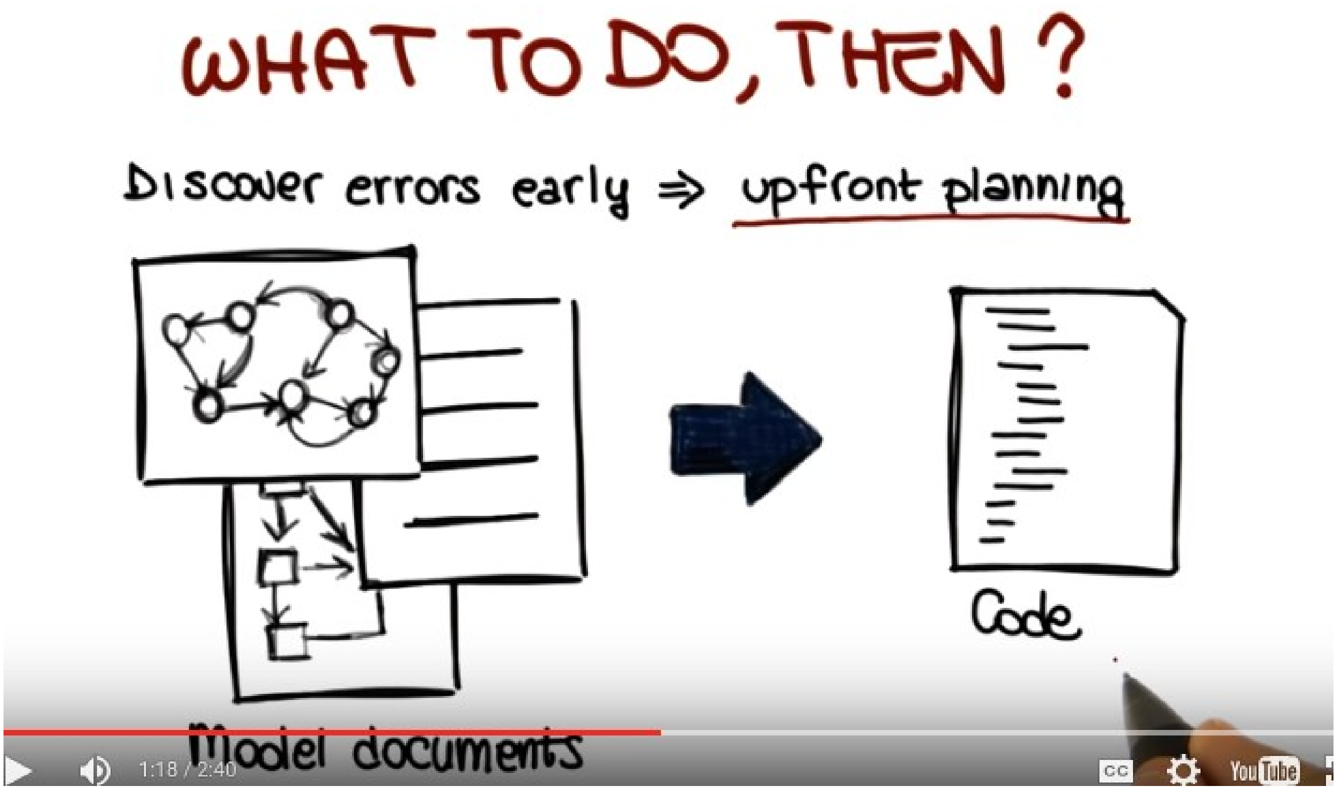
1. In previous lessons, we covered testing principles and techniques. In this lesson, we will discuss a type of software process that is heavily based on the use of testing. The agile(靈活的) development process. Also called test-driven development. To do that, we will revisit some of the assumptions that led to the definition of the more traditional software processes. The ones that we discussed so far. We will see how, when some of these assumptions are no longer valid, we can change the way in which we look at software processes. And we can change the way in which we look at software development in general. We will discuss how this changing perspective, lets us rethink software processes and make them more agile and better suited for context in which changes are the norm and we need to adapt fast. In particular, we will discuss two processes that apply the principles of agile software development and that are commonly used in industry. Extreme programming, also called XP, and Scrum. [BLANK_AUDIO]

2. We will start our discussion about test written and development by going back to a softer life cycle to examine a little bit ago which is the waterfall life cycle. And if you remember, that was a totally rigid process in which we were preparing documents and we were not studying any phase before the previous one was finished. And once a phase was finished, we were really going back to it. So today we are going to examine how it is possible to go from such a rigid process to an agile one, in which we can better deal with changes. So remember what we saw in the first lesson when Barry Boehm stated that the cost of change grows exponentially with time. So if we imagine to have time over here on the x-axis and cost on the y-axis, we can see the cost that will go up more or less this way. And what that means is finding a problem while collecting requirements will cost you much less than finding a problem in the analysis phase, which in turn, will cost you less than finding a problem during design, and so on for the subsequent phases. So if this is the case, and cost is really growing this fast as we proceed in our process, what should we do? The key thing is to discover errors early before they become expensive, which in turn means doing a lot of upfront planning. And because models are cheaper to modify than code, we're willing to make large investments in upfront analysis and design models. And only after we have built and checked these models, we're going to go ahead and build the code. In other words, we are following a waterfall mentality. However, something definitely changed in the last 30 years. For example, 30 years ago, we needed to walk down the hall, submit a deck of cards to an operator, and wait a day for our program to run and produce some results. Today we can leverage the computational power of the cloud. Computers used to be very slow and very expensive. Today, computer are a thousand times faster and a thousand times cheaper than what they used to be. In particular, if you think about the compile and test cycle, that has gone from days to seconds. Now we can change our code, compile it, run it our tests, all in a matter of instants, something that was unthinkable before. Finally, developers in the past had to do a lot of tasks manually in a very time-consuming way and often in a very painful way. Today, on the contrary, we can now automate a lot of these tasks. We have high level programming languages, version control systems, smart ideas.

Basically a whole set of tools that can help developers. And they can make them more efficient. In general, what that means is, it's easy to change, much easier than what it was before. So maybe if we take all that into account, the cost of change can be flat. So if we go back to our diagram, the one in which we showed the cost with respect to the project time, maybe instead of having this kind of curve, we might have a different kind of curve. Maybe the curve is more like this one. So maybe we can make all of this happen, as long as we use tools, practices and principles in the right way. And we're going to see what that means.

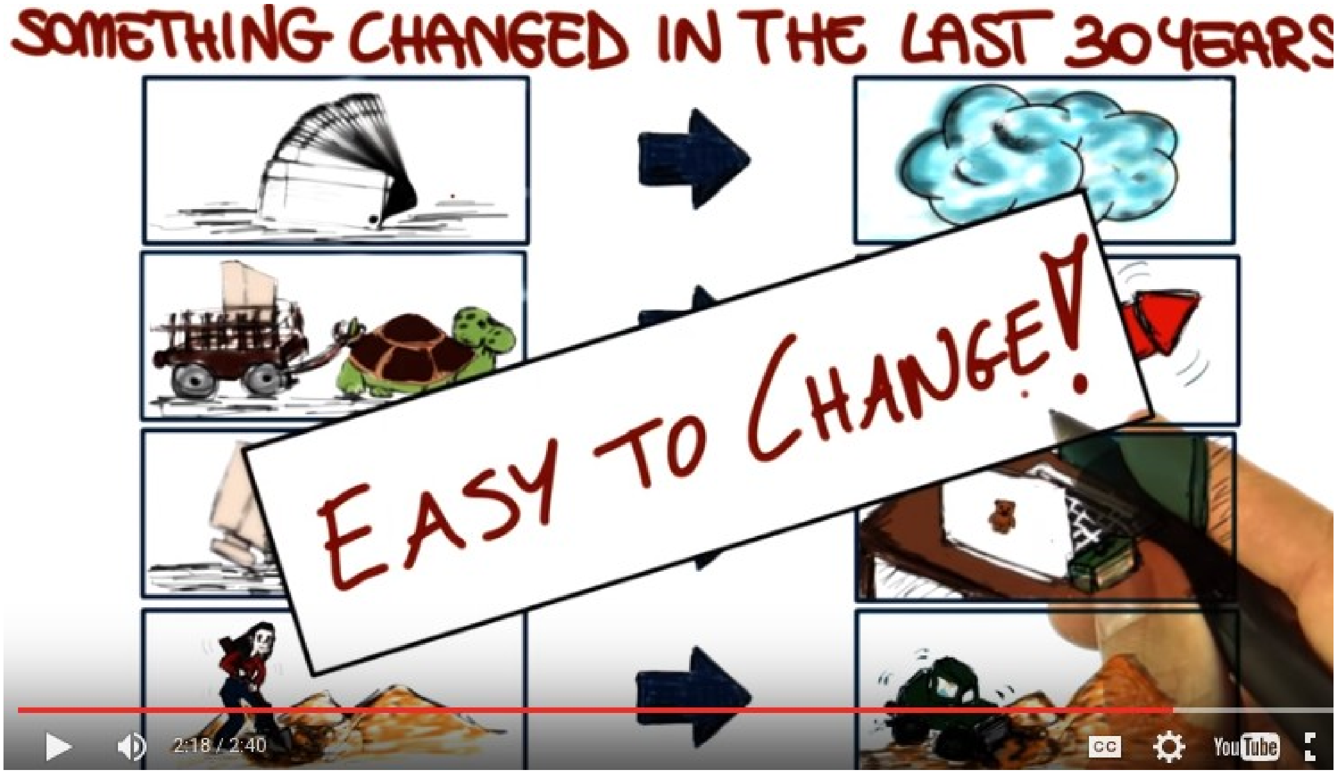






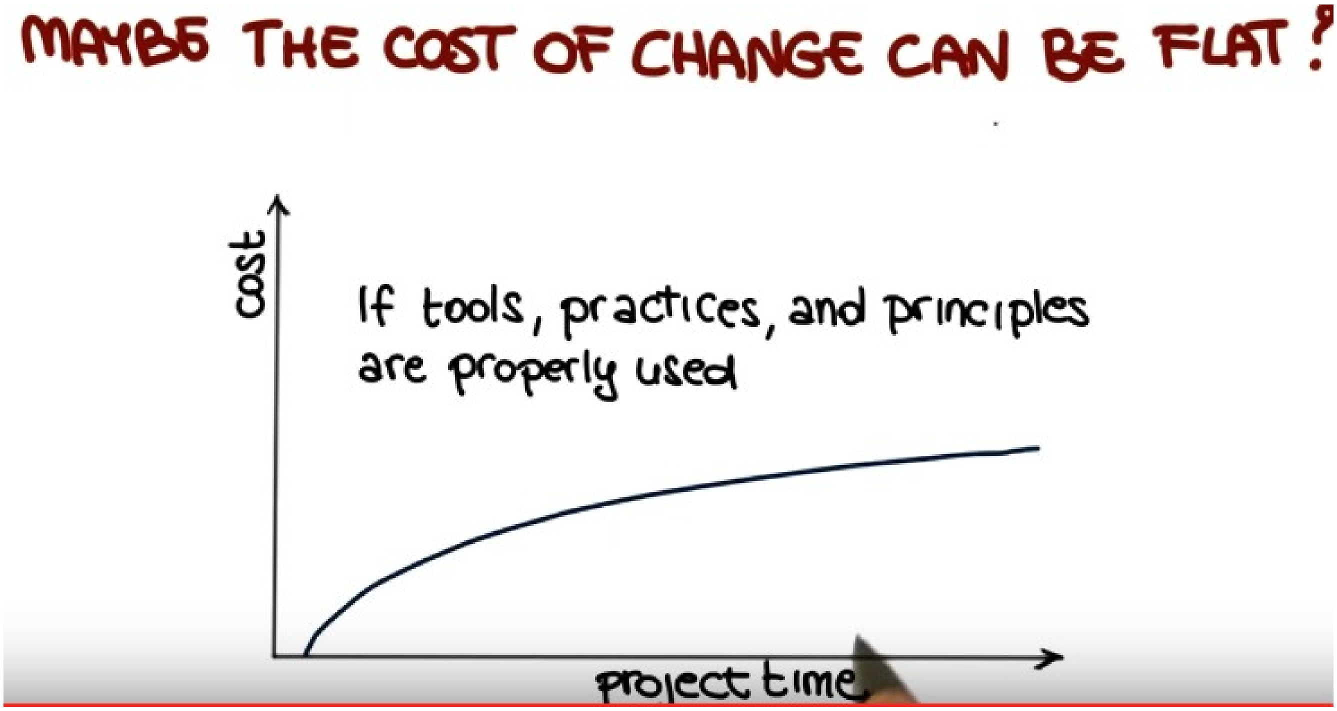
WHAT TO DO, THEN ?

Discover errors early ⇒ upfront planning

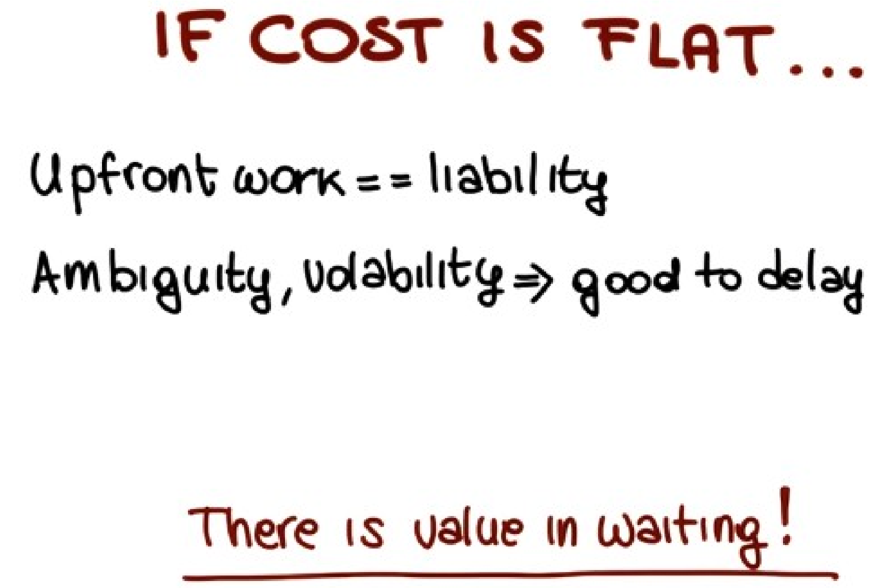


SOMETHING CHANGED IN THE LAST 30 YEARS

EASY TO CHANGE!


MAYBE THE COST OF CHANGE CAN BE FLAT ?
cost
If tools, practices, and principles are properly used
project time

3. And assuming that cost is flat that we can really lower that curve then teher are a few interesting consequences. First of all upfront work becomes a liability, we pay for speculative work some of which is likely to be wrong. Some of which we are likely to undo and the reason for ambiguity and volability for example in requirements then it's good to delay We don't want to plan for something that might never happen, to invest resources in something that we might have to throw away later on. In general, if cost is flat it is cost effective to delay all decisions until the last possible moment and only pay for what we use, so to speak. In other words, there is value in waiting, time answers questions and removes uncertainty. And we want to take advantage of that. This and other considerations led to the birth of Agile Softer Development. Specifically for those of you who are interested in a little bit of history. In February 2001 a group of software developers, 17 of them, met to discuss lightweight development methods and published Manifesto for Agile Software Developement. Which introduces and defines the concept of agile software development, or agile methods. In a nutshell, agile methods aim at flat cost and a decrease in traditional overhead by following a set of important principles. Our first principle is to focus on the code, rather than the design, to avoid unnecessary changes. Another principle is to focus on people, value people over process, and make sure to reward people. In addition agile methods are all based on iterative approaches to software development, to deliver working software quickly, and to be evolve it Just as quickly based on feedback. And feedback can come from many sources, in particular, it'll come from the customer, it'll be customer feedback. And to be able to do so, agile methods need to involve the customer throughout the development process. Finally, there are two more principles I want to mention. Which are cornerstones of agile methods. The first one is the expectation that requirements will change, and therefore, we need to be able to handle some changes. We can't count on the requirements to be still and unmutable. And the last principle is the mentality of simplicity. Simple design and simple code and so on. But be careful, because simple does not mean inadequate, but rather, as simple as possible.

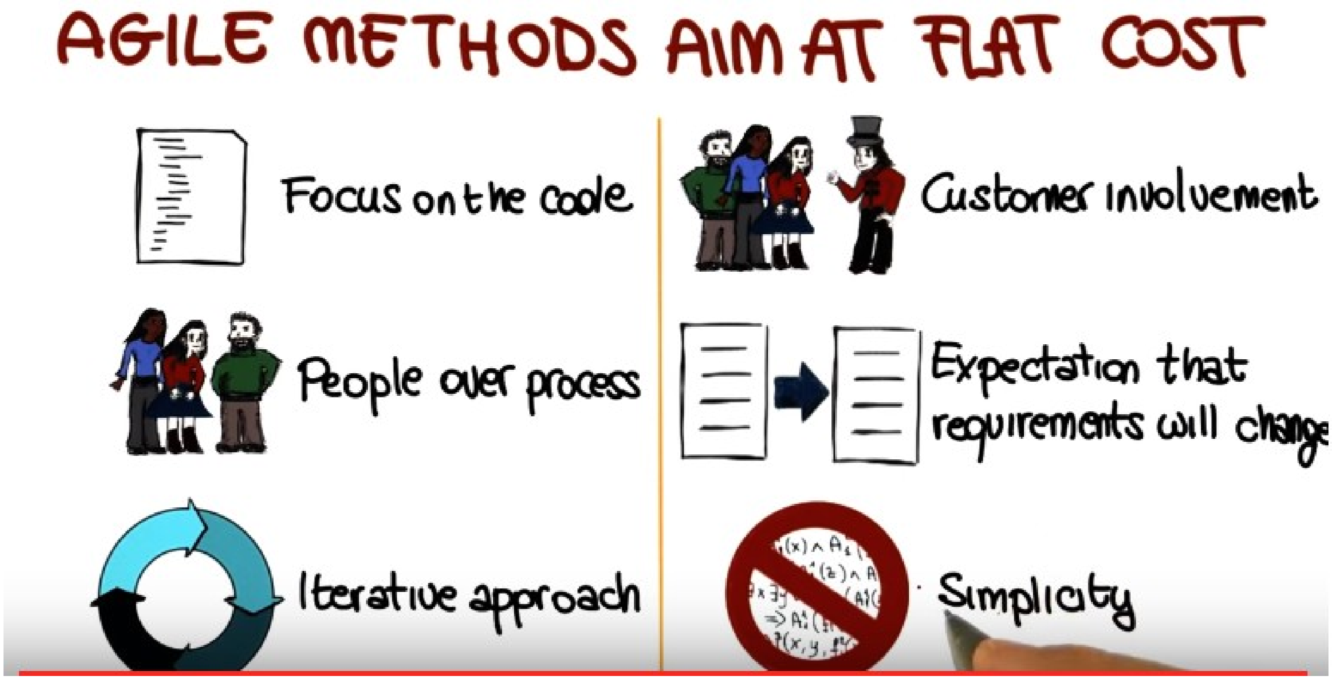


IF COST IS FLAT...

Upfront work == liability

Ambiguity, volatility ⇒ good to delay

There is value in waiting!



AGILE METHODS AIM AT FLAT COST

Focus on the code

Customer involvement

People over process

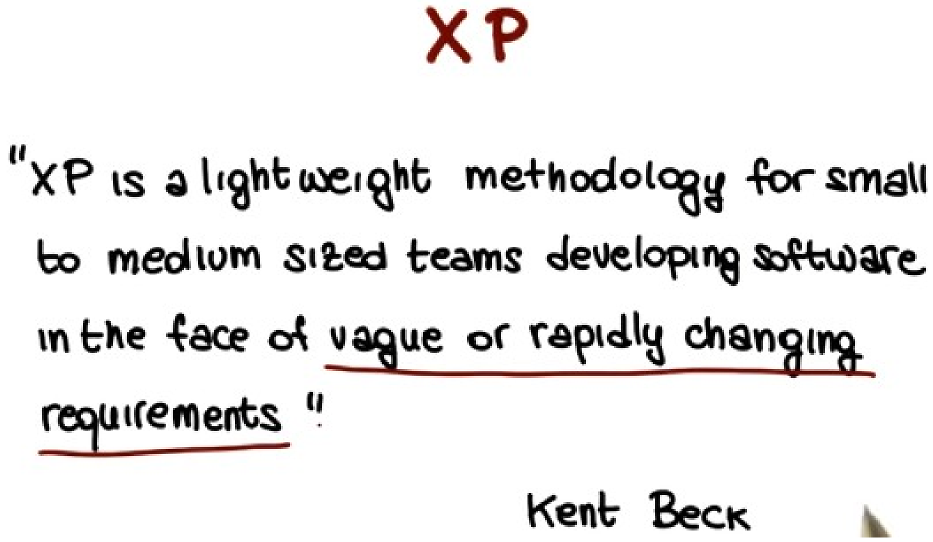
Expectation that requirements will change

Iterative approach

Simplicity

4. So now let's talk about a specific agile method, which is also one of the first ones, extreme programming, also called XP. And to introduce XP, I'm going to start with a quote. The quote says XP is a lightweight methodology for small to medium sized teams developing software in the face of vague or rapidly-changing requirements. And this is a quote from Kent Beck the American Software Engineer that created extreme programming. And by the way Beck was one of the original 17 developers who signed the manifesto in 2001. And as you can see we are still talking about the methodology. So we are

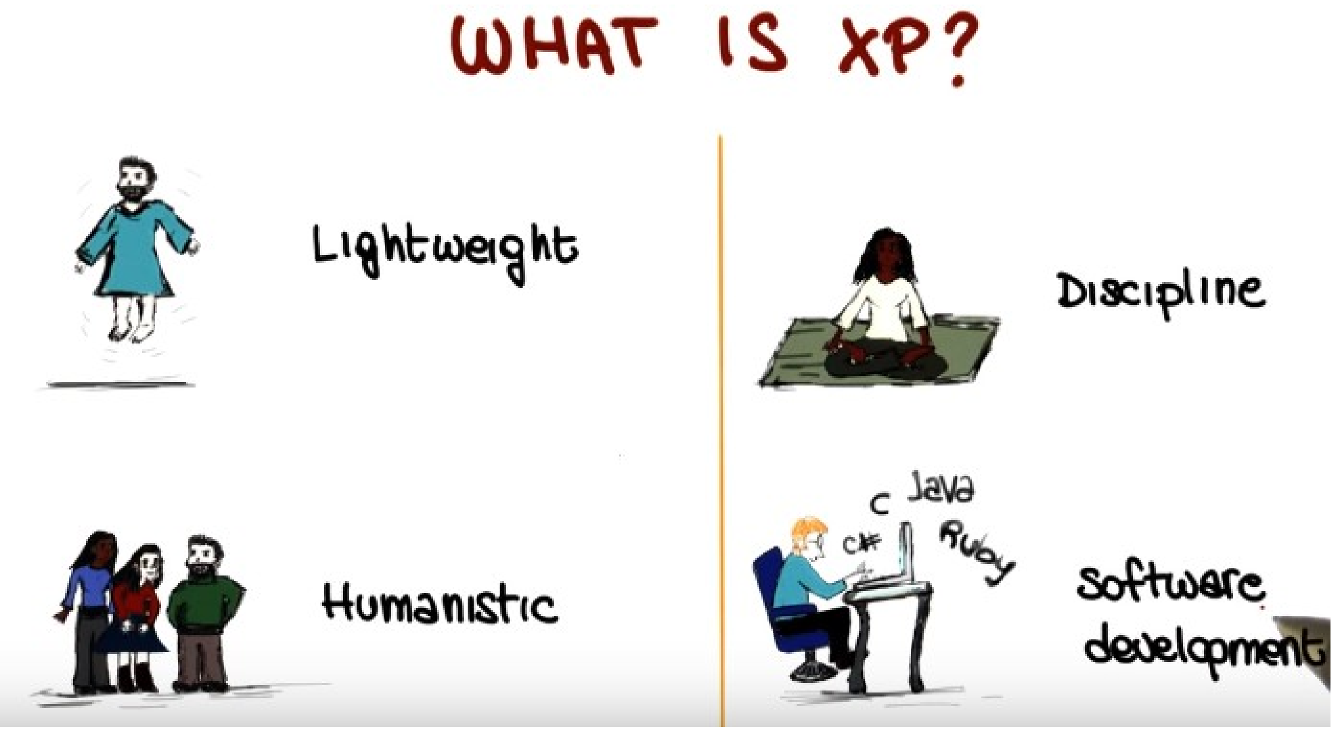
not just talking about going out and just start writing software. There are principles. And there are practices that we need to follow, but we're going to do it in a much more agile, and a much more flexible ways than we did for our software processes. And also note that the vague and rapidly changing requirements are explicitly mentioned, because this is really one of the important parts about all of this agile methodologies. so what is XP? XP is a. Lightweight. Humanistic. Discipline. Of software development. It is lightweight because it doesn't overburden the developers with an invasive process. So process is kept to a minimum. It's humanistic because as we said, it's centered on people. People, developers, customers, are at the center of the process. It's a discipline, as we said, it includes practices that we to Follow. And finally, is of course about software development. Software development is a key point of the whole method. In XP, developing is like driving, imagine having a road, a wind road, we need to able to drive our car down the road, take the abrupt turns, react promptly to changes, for example obstacles on the road. So, in a nutshell, change is the only constant. Eyes always have to be on the road and it's about steering and not pointing, and XP is trying to do the same thing, while creating softer systems. In XP we need to adopt a mentality of sufficiency, what does that mean? How would you program if you had all the time in the world? No time constraints at all, you will probably write tests instead of skipping them, because there's no more resources. You will probably restructure your code often, because you see opportunities to improve it, and you will take them. And you will probably talk to fellow programmers and with the customer, interact with them, and this is actually the kind of mentality that XP is trying to promote and agile processes in general. And we will see that the following some of the practices that XP is advocating, you can actually achieve these goals and you can actually behave in this way. And the development process is going to benefit overall.

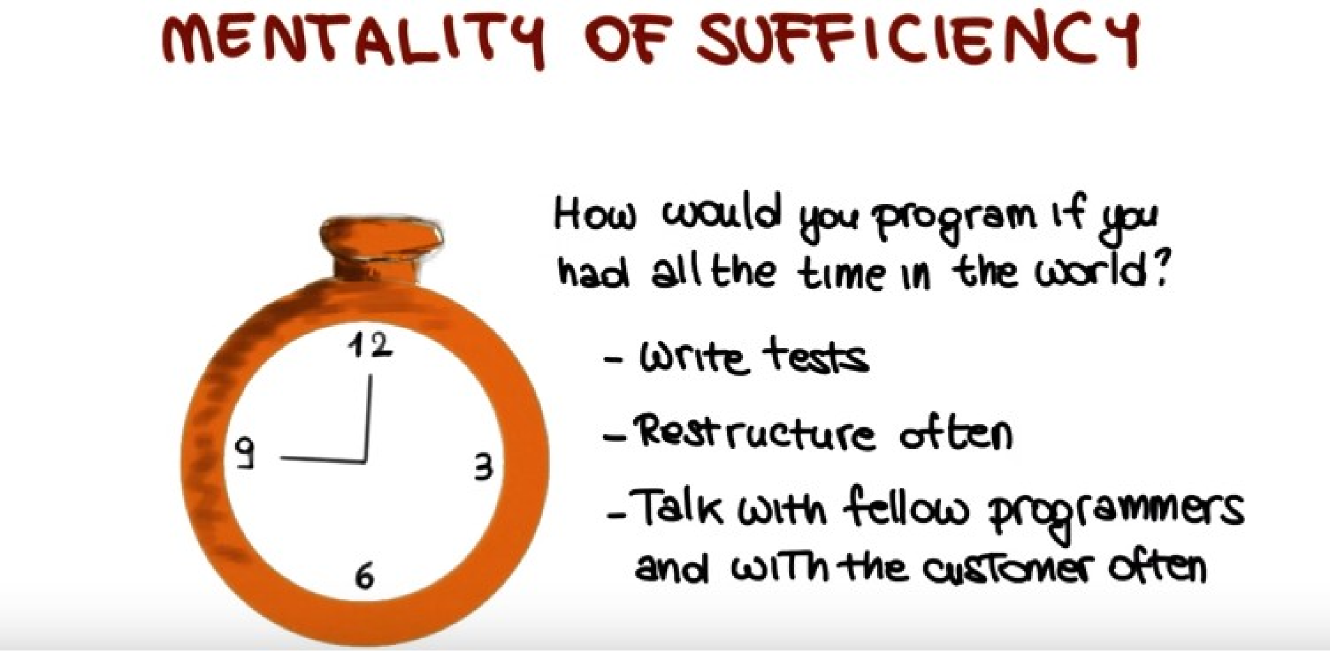


XP

"XP is a lightweight methodology for small to medium sized teams developing software in the face of vague or rapidly changing requirements"

Kent Beck



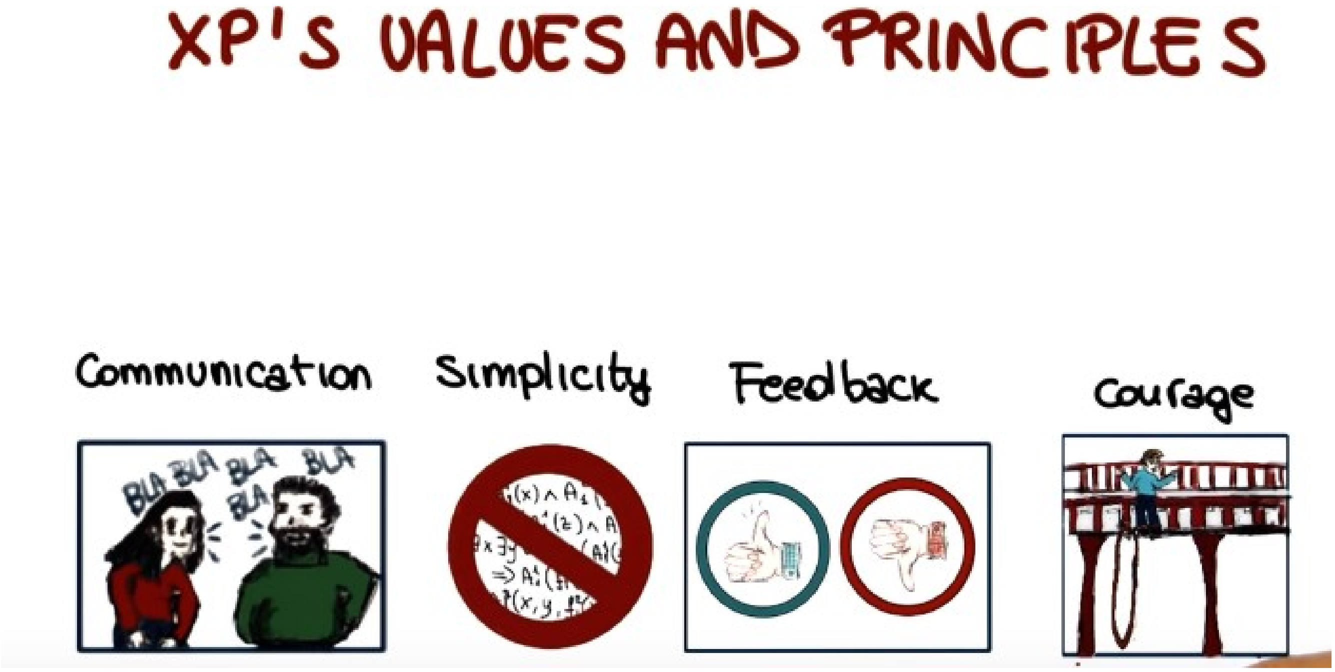

MENTALITY OF SUFFICIENCY

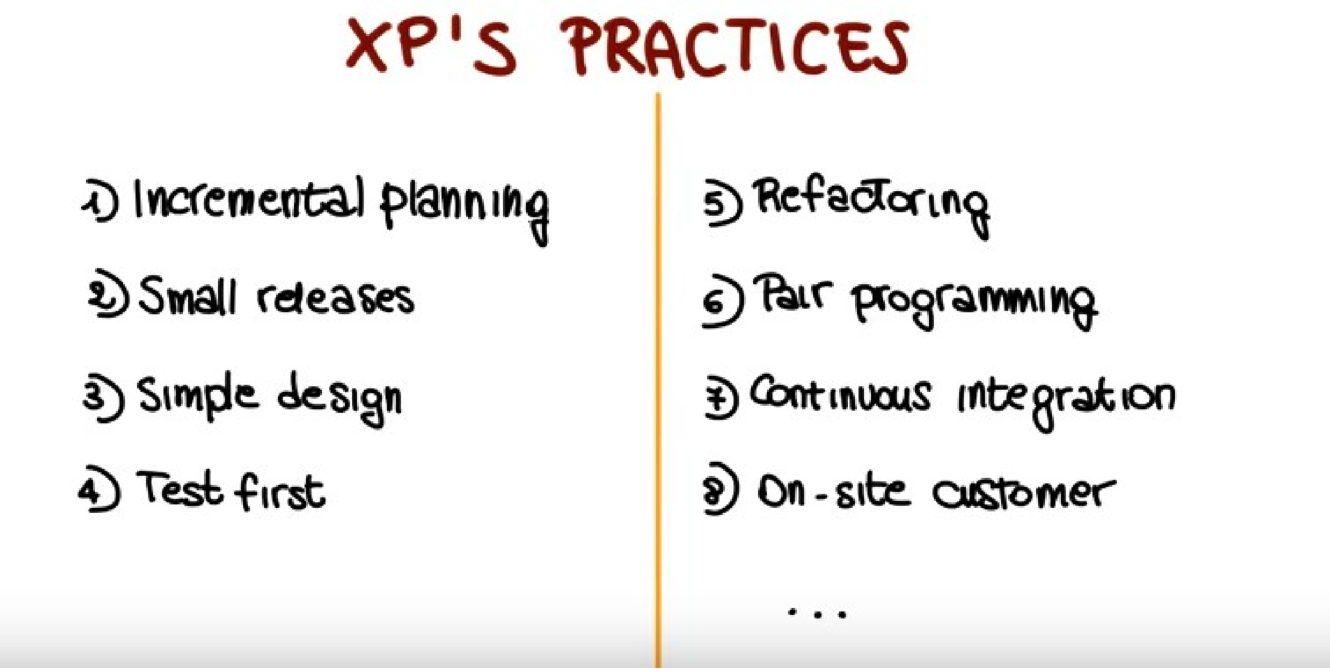
How would you program if you had all the time in the world?

- Write tests
- Restructure often
- Talk with fellow programmers and with the customer often

5. Next I'm going to go over some of the XP's values and principles that I just hinted on earlier in the lesson. The first one, the first important value is communication. There is no good project without good communication. And XP tries to keep the right communication flowing and it uses several practices to do that. It's got practices based on and require information and in general share the information. For example, pair programming. And we're going to say more about that. User stories, customer involvement, all activities that help communication, that faster communication. Another important principle that we already saw, is simplicity. And the motto here is live for today without worrying too much about the future. When you have to do something, look for the simplest thing that

works. And the emphasis here is on, that works. We want to build something simple, but not something stupid. Feedback. That's extremely important in XP, and it occurs at different levels, and it is used to drive changes. For example, developers write test cases. And that's immediate feedback. If your test cases fail, there's something wrong with the code. Or there's something that you still haven't developed. Developers also estimate new stories right away as soon as they get them from the customers and that's immediate feedback to the customer. And finally, on a slightly longer time frame, customers and tester develop together functional system test cases to assess the overall system. And also in this case, that's a great way to provide feedback and by the way, also to help communication. And finally, courage, the courage to throw away code if it doesn't work. To change it if you find a way to improve it. To fix it if you find a problem. To try out new things if you think that they might work better than what you have right now. Now, that we can build and test systems very quickly, we can be much braver than what we were before. So how do we accomplish all that? And what are XP's practices that are going to help us follow these principles and adhere to those values? These are some of XP practices. There are more, but those are the ones that I'd like to discuss in a little more depth, individually. Incremental planning, small releases, simple design, test first, refactoring. We will actually have a whole lesson next on refactoring. Pair programming, continuous integration, and on-site customer. So let's start with incremental planning.





XP'S PRACTICES

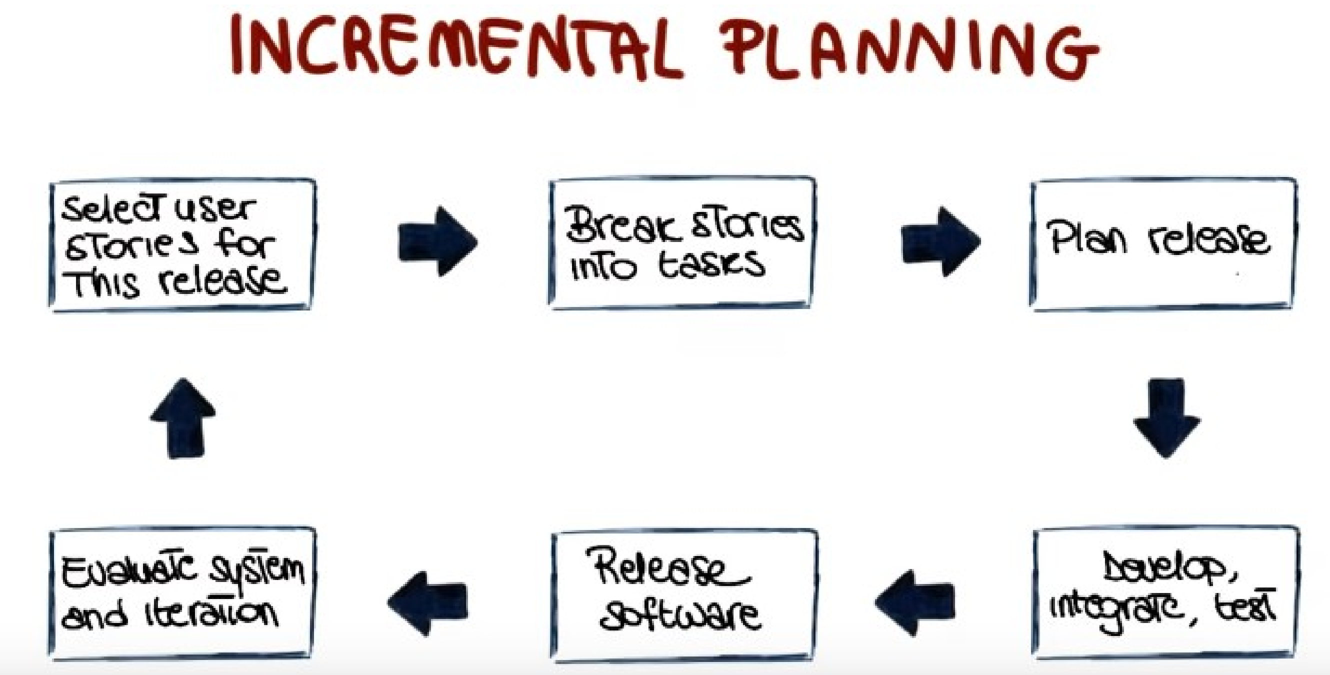
1) Incremental planning
2) Small releases
3) Simple design
4) Test first

5) Refactoring
6) Pair programming
7) Continuous integration
8) On-site customer

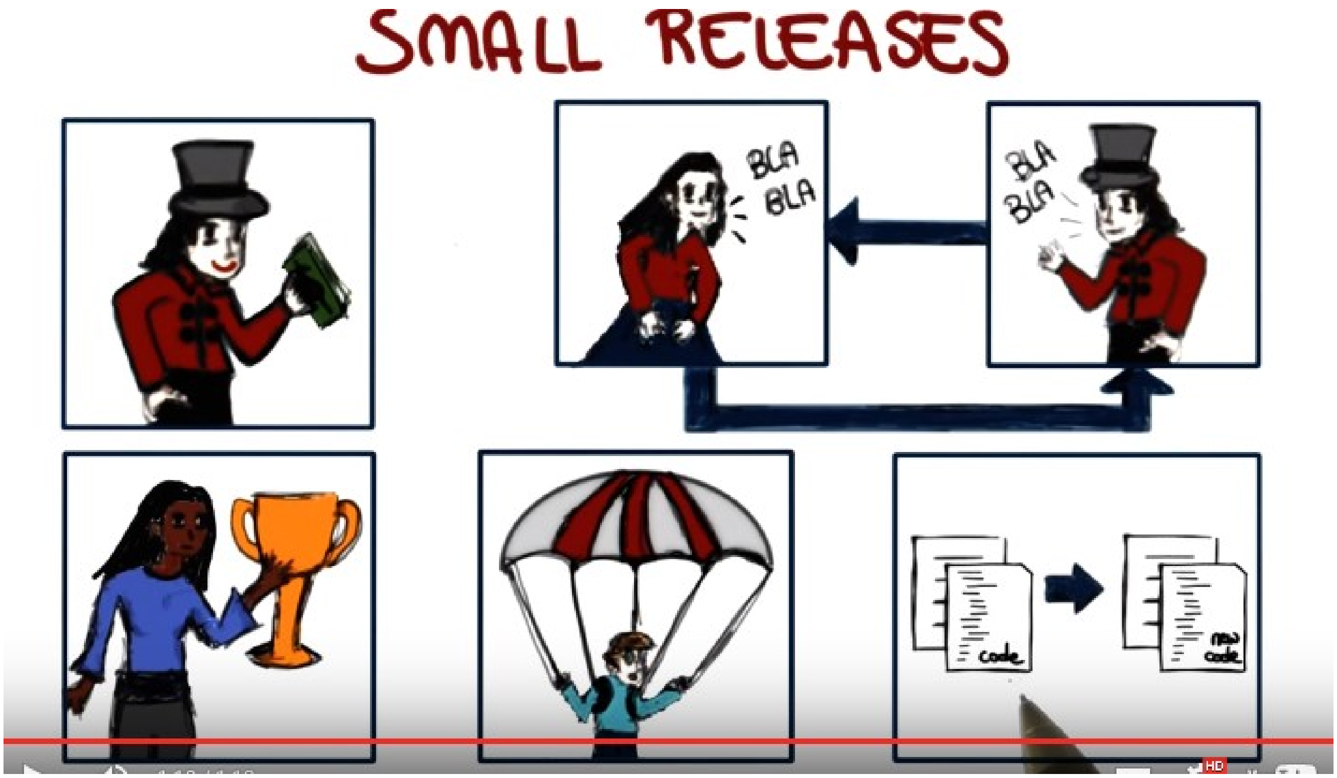
...

以下就是分別講上圖中的八點

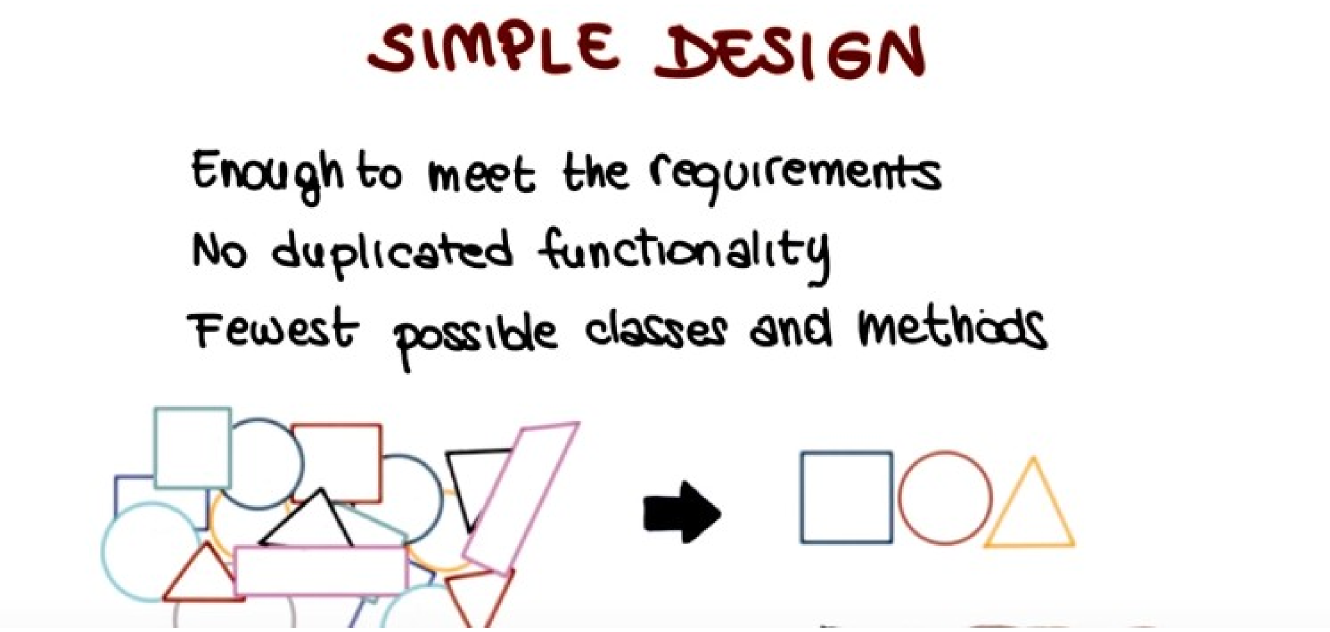
6. Incremental planning is based on the idea that requirements are recorded on story cards, sort of use cases or scenarios that the customer provides. So the first step in incremental planning is to select user stories for a given release, and which stories exactly to include depends on the time available and on the priority. For example, scenarios that we might want to realize right away because they are particular important for the customer. After we select user stories for the release, we break stories i to tasks. So what we do, we take the user stories and we identify specific development tasks that we need to perform in order to realize these stories. Once we know our tasks, we can plan our release. And at that point, we can develop, integrate and test our code. And of course, this is more kind of an iterative process right here, so we do this many times. When we're ready, when we accomplish all of our tasks, all of our tests pass and we're happy, we release this software. At the point, the released software is evaluated by us, by the customer, and we can reiterate. We can continue the process, select more stories and continue it this way.



7. The first practice that we just saw goes together with small releases practice. This idea that instead of having a big release at the end of a long development cycle, we try to release very often. And there are many advantages to small releases and to releasing often. The first one is that we deliver real business value on a very short cycle. And what that means is that we get business value sooner, and that in turn increase our customer confidence and makes the customer more happy. So more releases also mean rapid feedback. We release the software soon, we get feedback from the customer soon, and we can in this way do exactly what we were saying before, steer instead of driving, adapt weekly to possible changes in the requirements. We avoid working for six months on a project and find out six months later that the customer wanted something else and we got the wrong requirements. In addition having small releases, so seeing your product deployed and released soon produces a sense of accomplishment for the developers. And in addition, it also reduces risk because again, if we're going down the wrong path, we will know right away. If we're late, we will know right away. So these are just additional advantages of having this quick cycle and more releases. And finally, as we also said before, we can quickly adapt in the case our requirements change our code to the new requirements.



8. The next practice is simple design. We want to avoid creating a huge complicated possibly cumbersome design at the beginning of the project. What we want to have instead is just enough design to meet the requirements, so no duplicated functionality, fewest possible classes and methods in general, just the amount of design that we need to get our system to work. So one might object that to for designing that way we will have to change the code a lot, we will need to adapt the design as the code evolves, and that's exactly the point. That's what we will do. XP is all about changing and adapting, changing your design, changing your code, refactoring. And with the fact that we have test cases throughout the development process, we can do that with confidence. Because if we break something we will know right away, which leads us to the next practice.



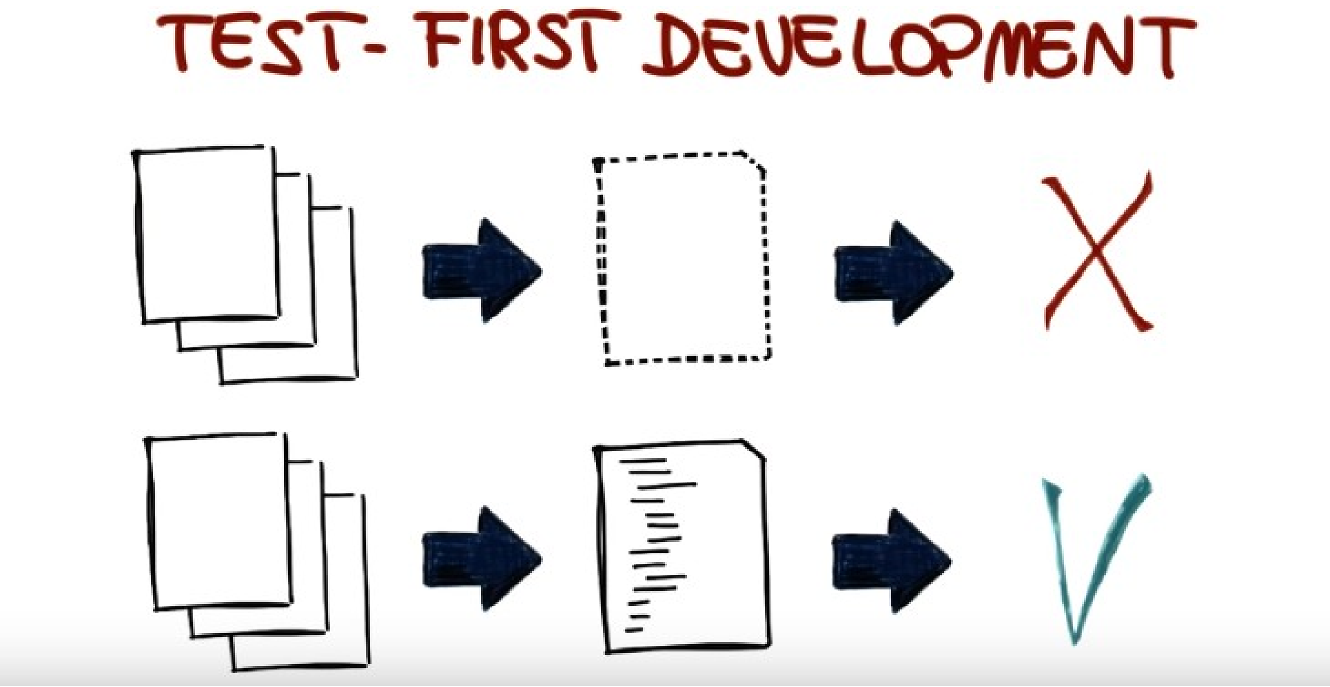
SIMPLE DESIGN

Enough to meet the requirements

No duplicated functionality

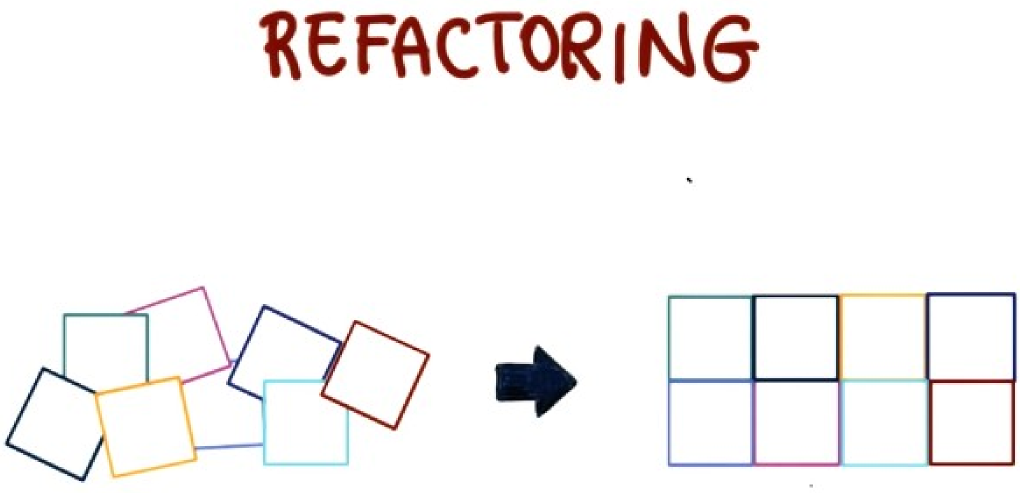
Fewest possible classes and methods

9. Which is test-first development. The key idea is that any program feature that doesn't have an automatic test simply does not exist. If there is a feature, you need to write a test for the feature before. So, what developers do is to create unit tests for each such piece of functionality even before the functionality is implemented. And of course, when you run this test, they will fail. But the beauty of it is that, as you write your code and you add more and more fractionality to the feature that you're developing, these test cases are going to start to pass. And that's extremely rewarding because it gives you immediate feedback, again feedback on the fact that you're developing the code in the right way. As soon as you write it, you will know. And if you write a piece of code and the test says still fail, that means that the code is not doing what it's supposed to do.



TEST- FIRST DEVELOPMENT

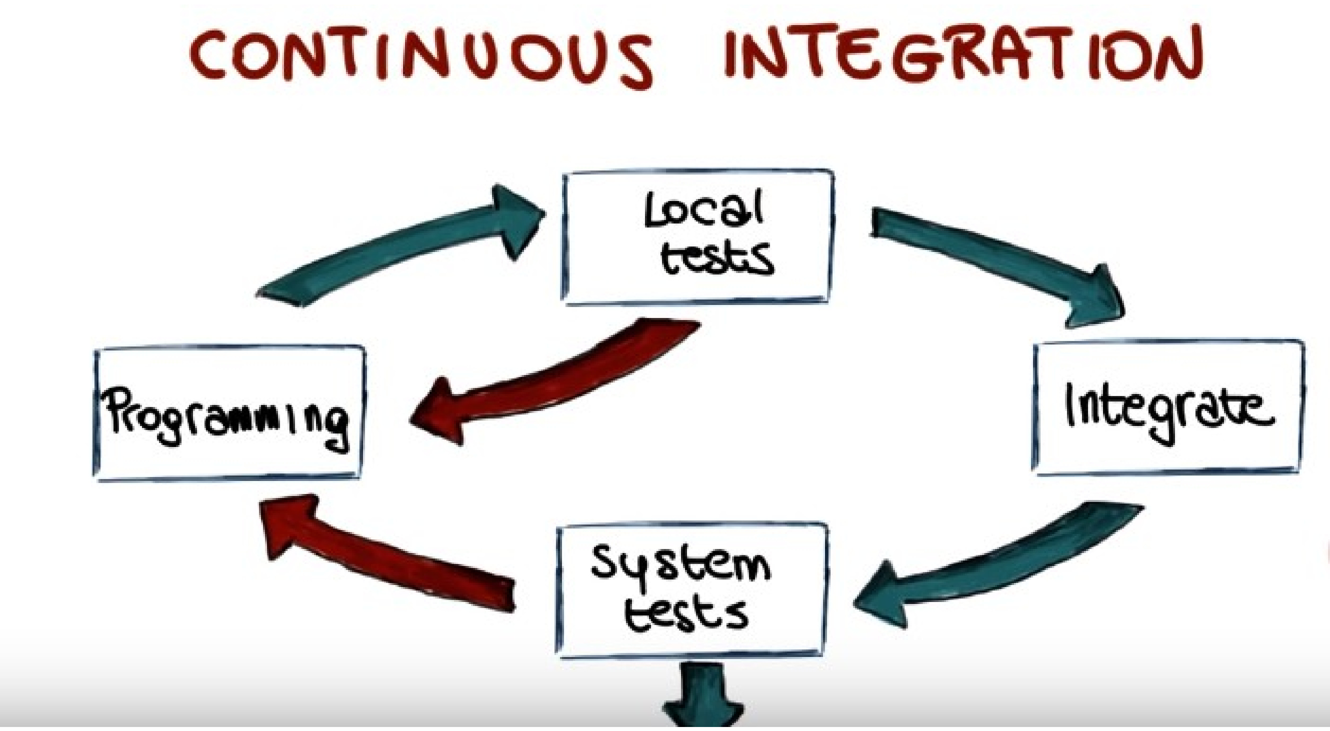
10. A couple of minutes ago we talked about the fact that well, we might need to change our design a lot, so how we going to do that, that's going to be expensive. Well it's not very expensive, if we can do efficient refactoring. Which is another one of the important xp practices. And what does it mean to refactor? It means to take a piece of code who's design might be suboptimal, because for example, we evolved it, we didn't take into account that from the beginning some of the features that had to be added later, probably because we didn't even know about this feature, because the requirements evolved. So we're going to take this piece of code and we're going to restructure it, so that it becomes simple and maintainable. Developers are expected to refactor as soon as opportunities for improvement, are found. And that happens for example, before adding some code. You might look at the code that you're about to modify, or to which you are about to add parts, and say can we change the program to make the addition simple, that has maintainability or we can do it after adding some code to our code base. We might look at the code, the resulting code, and say well can we make the program simpler? Was the running all the tests and the key point here is that we don't want to refactor on speculation, but we want to refactor on demand, on the system, and the process needed. Again the goal is just to keep the code simple and maintainable, not to over do it. And as I mentioned before we're going to have a whole lesson, the next lesson on refactoring. So we're going to go in more depth in the discussion of this topic.



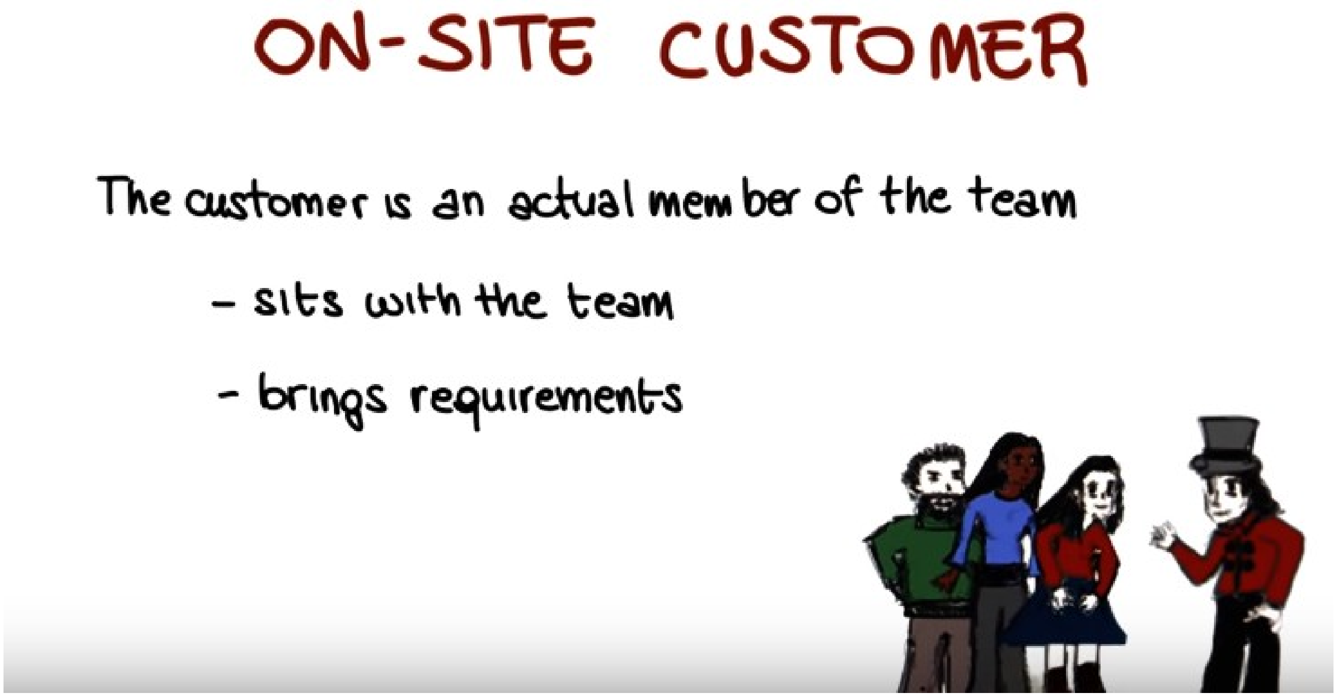
11. The next practice I want to discuss is a very important one in XP, and also one of the scandal, controversial, and it's the practice of pair programming. What does it mean? It means that all production code is written with two people looking at one machine. And not that they're, they're working with one keyboard and one mouse or they're not just interfering and writing on each other's code. And the way in which that happens is by playing different roles at different times. So the two developers alternate between the role of programming and strategizing, where strategizing means, for example, looking at the code that has been written and thinking whether that would work. Or what other tests that are not there might not work, given the way the code is being written. Or maybe looking at the code from a, you know, slightly detached perspective and trying to figure out whether the code can be made simpler, more maintainable, more efficient. And interestingly, there are measurements, there are studies that suggest that development productivity with pair programming is similar to that of two people working independently. And that answers one of the main objections against pair programming, which is why should I put two developers together, which is going to cut their productivity in half. It is not. Studies shows that that does not happen. And that the resulting code

can actually benefit from the fact that two developers are working together.

12. An important practice to get all of this to work is continuous integration, which means integrating and testing every few hours, or a day at most, because we don't want problems to pile up and to be discovered too late when there are too many of them to fix. So what goes on here is a cycle. And the cycle starts with the developer's programming, as soon as the developers are done modifying the code and they have a stable version they will run the local tests. If the local tests fail, the developers will go back to programming to fix their code and possibly add new code as needed, and this cycle, mini cycle will continue until all the local tests pass. At that point the developers can integrate their code with the code of other developers. And they can run test for the integrated system, and when they run this test again there are two possibilities. The test might fail, and if the test fails you broke it, and therefore you'll have to fix it. So developers will have to go back and modify the system and again going through the cycle of running the local tests, integrating, and running the systems tests. Conversely, if all the systems tests pass, then at that point the code is good to go and it is integrated into the system. And it will be the problem of some other developers if something breaks because at the time you integrated your code, the code was compiling, running and passing the tests successfully. So again, if we do this every few hours or every day, we can find problems very early, and we can avoid the situations in which we have many different changes coming from many different developers in a integration nightmare as a result.



13. The last practice I want to mention is on-site customer, and what that means is that literally the customer is an actual member of the team. So the customer will sit with the team and will bring requirements to the team and discuss the requirements with them. So the typical objection to this practice is the fact that it's just impossible in the real world. There is no way that the customer can have one person staying with the team all the time, and the answer to that objection is that if the system is not worth the time of one customer then maybe the system is not worth building. In other words, if you're investing tons of dollars, tons of money in building a system, you might just as well invest a little more and have one of the people in the customer's organization stay with the team and be involved in the whole process.



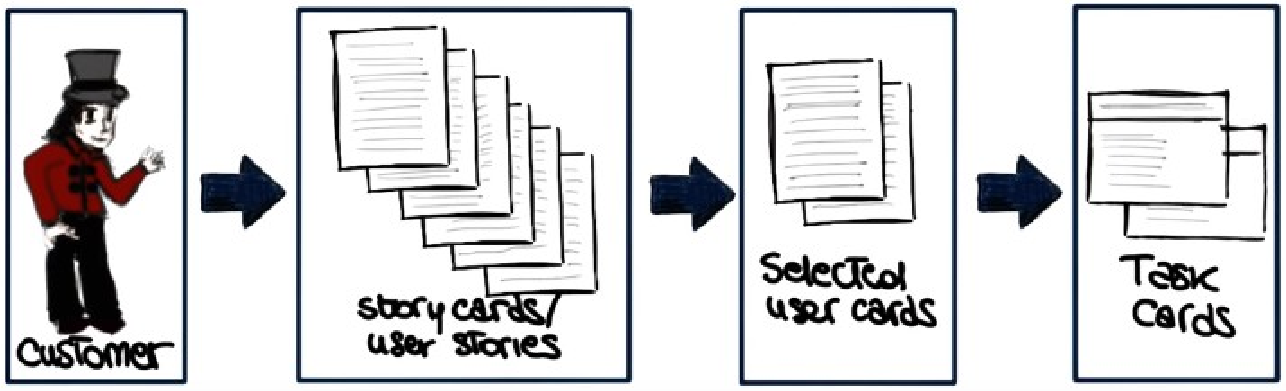
ON-SITE CUSTOMER

The customer is an actual member of the team

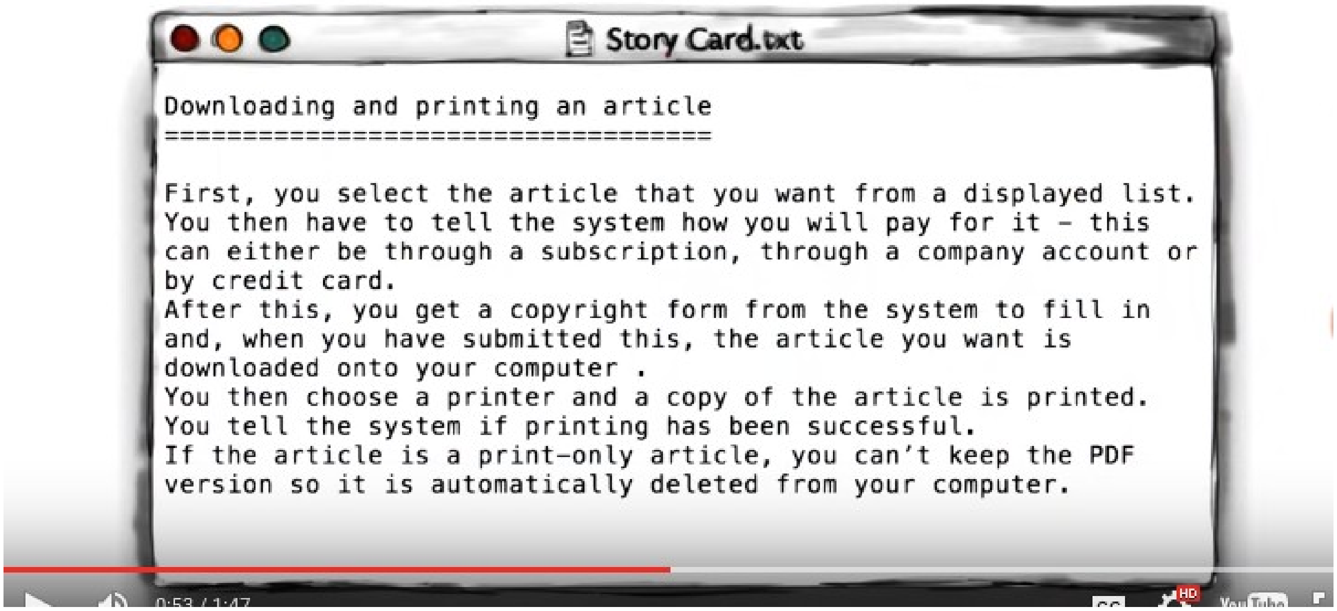
- sits with the team
- brings requirements

14. Now that we saw what the main values and practices of XP are, I want to go back for a minute to discussion of requirements engineering in XP. In XP, user requirements are expressed as scenarios or user stories, as we already discussed. These are written by customers on cards (這不就是 MadGraph 裡的 cards 嗎), and what the development team does is to take these cards, take these users stories and break them down into implementation tasks. And those implementation tasks are then used as a basis for scheduling cost estimates. So given these estimates, and based on their priorities, the customer will choose the stories that will be included in the next release, in the next iteration. And at this point, the corresponding cards will be taken by the developers and the, the task will be performed, and the relative, and the corresponding card will be developed. And just to give an idea of the order of magnitude, if you consider a few months project, there might be 50 to 100 user stories for a project of that duration. So, now let me give you an example of what the story card might look like, and I'm going to do it using a story card for document downloading and you can really do all of this, basically as seeing what the scenario is, downloading and printing an article. And it describes basically what happens when you do that, what is the scenario. First, you select the article that you want from a displayed list. You then have to tell the system how you will pay for it. This can either be through a subscription, through a company account or by credit card, and so on. So what developers do, they take this story card, and they break it down in to development tasks. So, here I'm showing you some examples of task cards for the user story that we just saw. In particular I'm showing three task cards and if we look at the third one, there is a name for the task, which is implement payment collection. So this is the development task that we have the perform and here, there's a description of what that developed code should do. And notice that, you know, the task card can even be more. explicit than this, more specific than this, and talk about actual development tasks.

REQUIREMENTS ENGINEERING IN XP

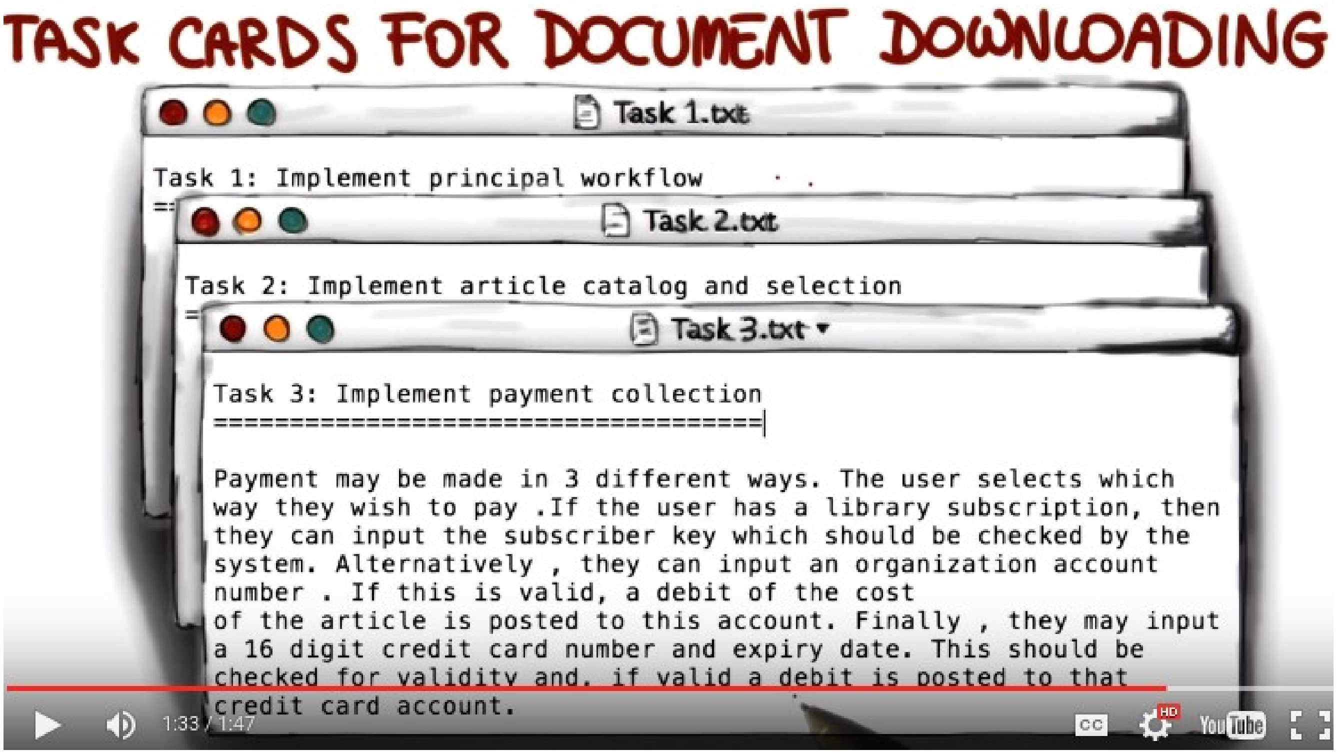


STORY CARD FOR DOCUMENT DOWNLOADING



```
Downloading and printing an article
===================================

First, you select the article that you want from a displayed list.
You then have to tell the system how you will pay for it - this
can either be through a subscription, through a company account or
by credit card.
After this, you get a copyright form from the system to fill in
and, when you have submitted this, the article you want is
downloaded onto your computer .
You then choose a printer and a copy of the article is printed.
You tell the system if printing has been successful.
If the article is a print-only article, you can’t keep the PDF
version so it is automatically deleted from your computer.
```



TASK CARDS FOR DOCUMENT DOWNLOADING

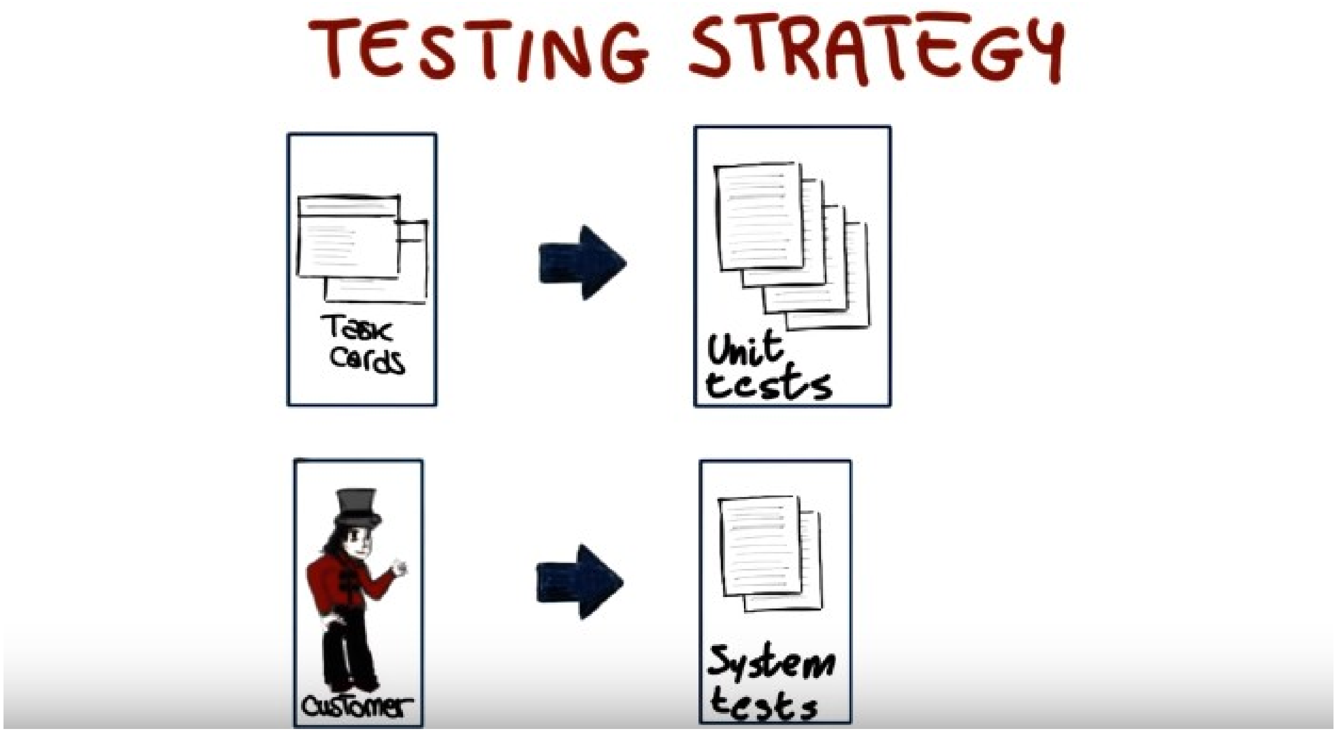
Task 1: Implement principal workflow

Task 2: Implement article catalog and selection

Task 3: Implement payment collection
====================================

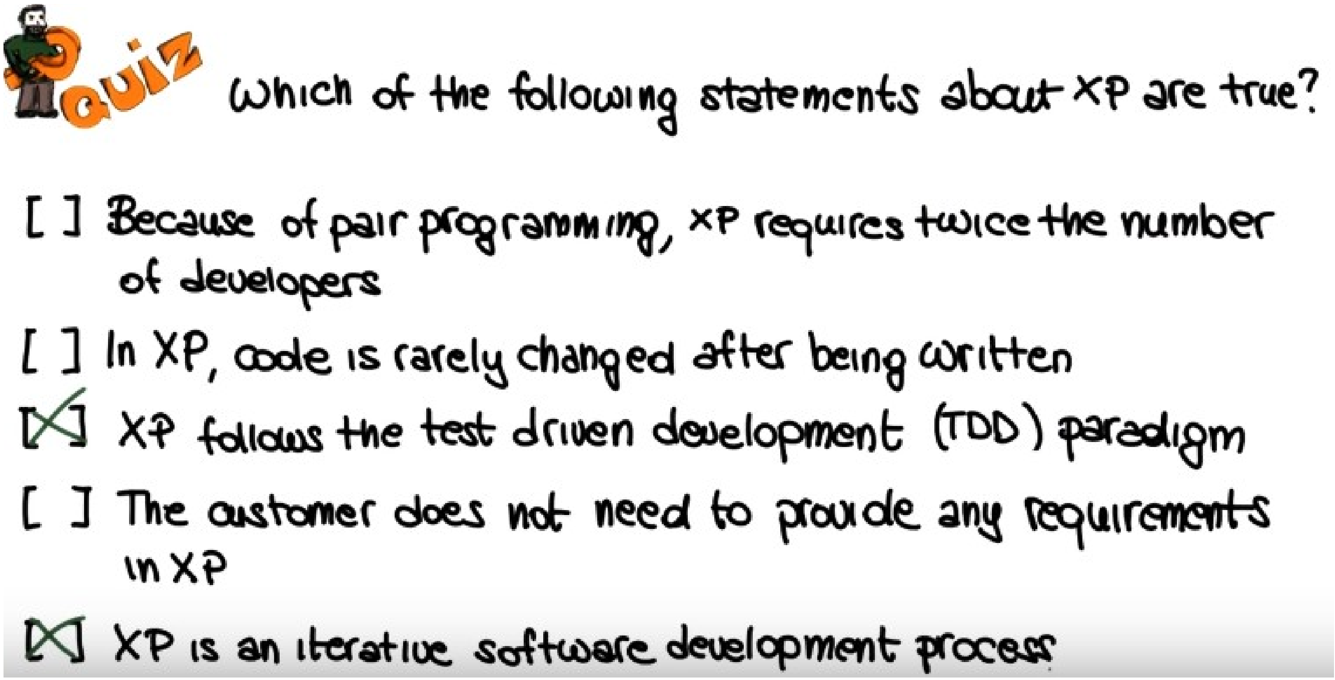
Payment may be made in 3 different ways. The user selects which way they wish to pay .If the user has a library subscription, then they can input the subscriber key which should be checked by the system. Alternatively , they can input an organization account number . If this is valid, a debit of the cost of the article is posted to this account. Finally , they may input a 16 digit credit card number and expiry date. This should be checked for validity and, if valid a debit is posted to that credit card account.

15. As you probably realized by now, at job development, it's a lot about testing. So there is a lot of emphasis on testing. Testing first, te, testing early. So that's the reason why I also want to discuss what is the testing strategy in XP. So first of all what is the basic principle? The basic principle is that testing is Coded confidence. You write your test cases and then you can run them anytime you want. And if they pass, they'll give you confidence that your code is behaving the way it's expected. If they don't pass on the other hand, you'll know that there's something to fix. Another important concept is that test might be isolated and automated. So both the running and the checking of the tests has to be automated for all of this to work. And there are two types of tests. The first type of test is unit tests, that are created by the programmers, and they're created by looking at the task cards. The task cards describe what they implemented, functionality should do, and therefore allows the developers. The right test that can test this functionality. That can check that the code's correctly implemented functionality. And as we said, you should really test every meaninful feature. So, for example, you should test every meaningful method in your classes. You should put specific attention to possibly complex implementations, special cases or specific problems that you might think of. while reading the task cards. In some cases, when you do refactoring, you might also want to write test cases specific to that refactoring. But we'll say more about that. So this was for the first kind of tests that are involved in the, in the XP process. The second kind of tests are the system tests, also called acceptance tests. And those tests involve the customer. So basically what happens is that the customer provides the test cases for their stores and then the development team transforms those into actual automated tests. So these (unit tests) are tests created by the developers. They run very quickly and they run very frequently. These (system tests) are tests developed with the help, with the involvement of the customer they run longer. And run less frequently, they run every time the system is integrated. According to the cycle we saw a few minutes ago.



16. Now that we are done discussing XP Extreme Programming, I would like to have a quiz, in which I make sure that some of the concepts behind XP are well understood. So, I'm going to ask you which of the following statements about Extreme Programming are true, and here are the statements. Because of pair programming, XP requires twice the number of developers. In XP, code is rarely changed after being written. XP follows the test driven development, or TDD, paradigm. The customer does not need to provide any requirements in XP. XP is an iterative software development process. So I would like

for you to mark all of the statements that you think are true about XP.



Quiz

Which of the following statements about XP are true?

☐ Because of pair programming, XP requires twice the number of developers

☐ In XP, code is rarely changed after being written

☒ XP follows the test driven development (TDD) paradigm

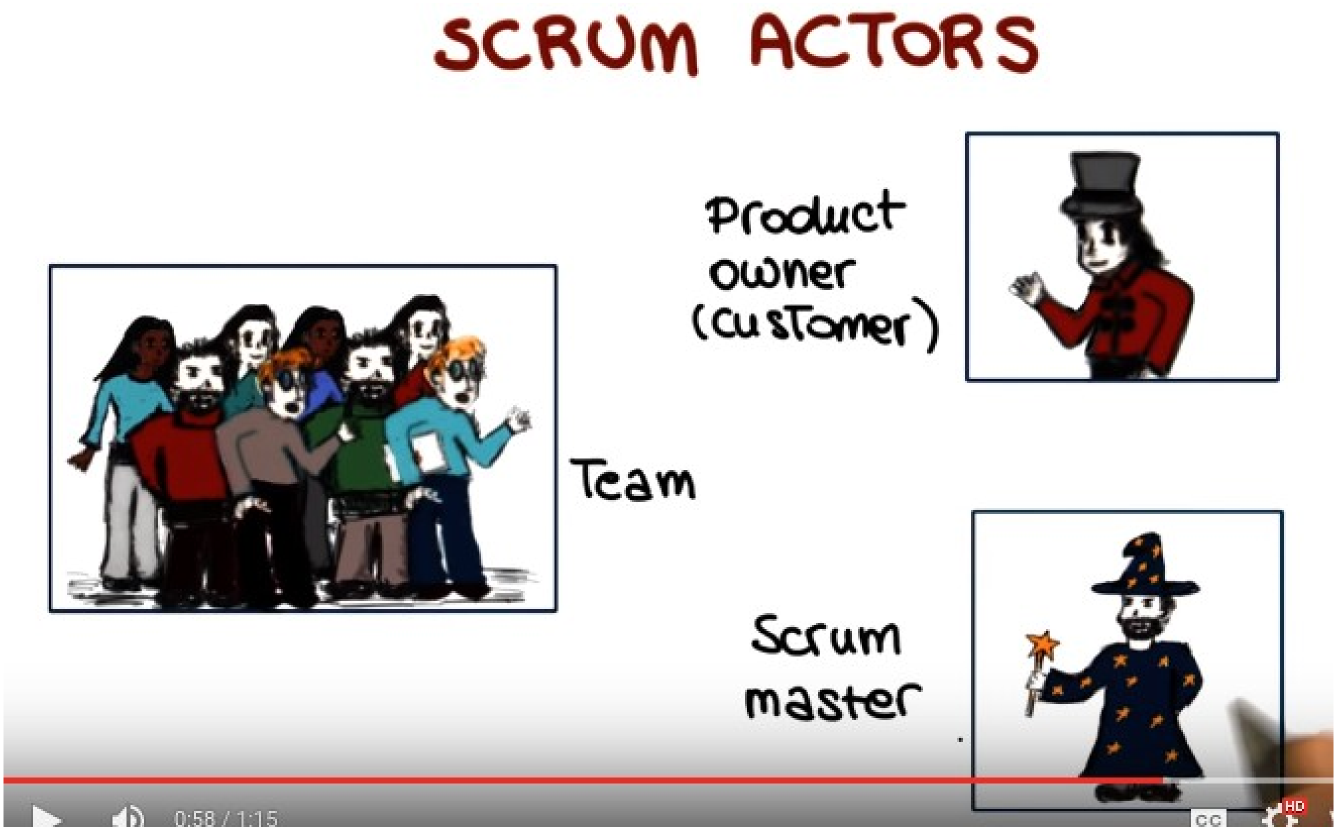
☐ The customer does not need to provide any requirements in XP

☒ XP is an iterative software development process

17. The first statement is false. It is not true that because of pair programming, we need twice as many developers. In fact that there is some evidence that even though in pair programming we have two developers working together, that the overall efficiency of the programmers is not really affected by use of this practice. In XP, code is rarely changed after being written. This is also definitely false. In fact in XP there is a lot of emphasis on change on the fact that the code can be changed, it can be resigned, because of the fact when we do that, we have a set of these cases that we can use to check right away that the code still works as expected. So again in XP, it's all about steering rather than just driving down one fixed direction. And therefore, the code can be changed. So this statement is false. It is definitely true that XP follows the test driven development paradigm. In XP we first write tests, and then we write the code, which is exactly what TDD is about. It is not true that the customer does not need to provide requirements in XP. The customer does provide requirements in the form of user stories, and the user stories are the starting point of the development process. Finally, XP is definitely an iterative software development process. In fact, we saw that XP is based on subsequent iterations of the same cycle, in which we select from a set of story cards, or user stories, the stories that we want to implement in the next iteration. Based on that we develop task cards, and then we use the task cards to write this case and then to write code. And we continue this cycle in an iterative way until we are done with all the story cards, and all the user stories, so definitely XP is an iterative software development process.

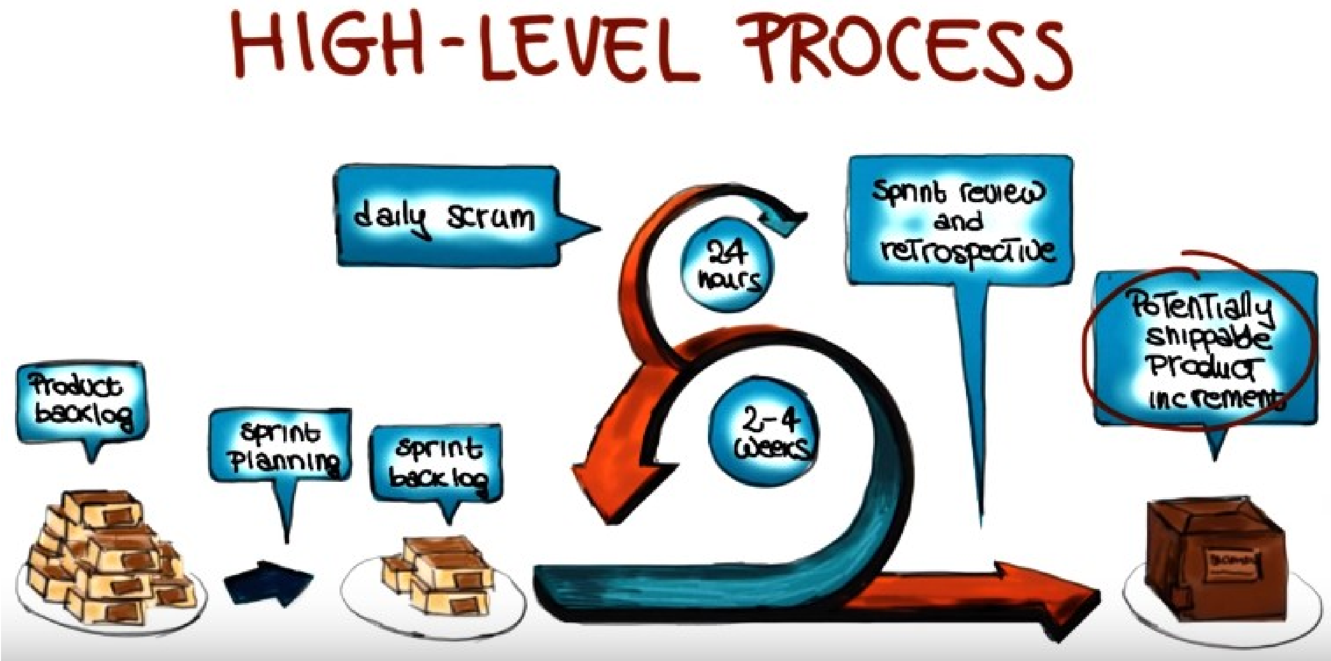
18. Before concluding this class on java development, I want to talk about another process that is very popular these days, and it's used in many companies, which is called Scrum. Which similar to XP is another agile development process, and I'm going to start by discussing what the Scrum actors are. There's three main kinds of actors. The first one is the product owner, which means the customer. The product owner is mainly responsible for the product back log, where the product back log is basically the list of things that have to be done, the back log in fact for the project. And that is analogous to the user stories to be realized in XP, that we just saw. So what the product owner does is to clearly express these back log items, and to also order them by value, so they can be prioritized. The second actor is the

team. The team is responsible for delivering shippable increments to estimate the back log items. It's normally self-organized, consists of four to nine people, and it's what you would consider normally as the main development team in a project. And finally we have the Scrum master. The Scrum master is the person who's responsible for the overall Scrum process, so he or she has to remove obstacles, facilitate events, helps communications, and so on. So you you can see the Scrum master as sort of a manager or the person who's got oversight, or the supervisor of the Scrum process.



19. So I want to conclude this lesson by providing a high level view of this scrum process. The process is represented here, and as you can see it has several components. We're going to go through all of them one at a time. We're going to start with a product backlog. Product backlog is the single source of requirements, for the process. They're order by value raised priority necessity, so that all of this characteristics can be taken into account when selecting which backlog items to consider for the next iteration. It's a living list in the sense that backlog items can be added or removed. And it's not really defined as we just said, by the product owner. In the sprint planning, what happens is that the next increment or the next sprint is defined. So basically, the backlog items of interest are selected based on the characteristics we just mentioned: value, [UNKNOWN], priority, and necessity. And the items are converted into tasks and estimated. So the result is this sprint backlog, which is the set of backlog items that will be completed during the next sprint. The sprint is an actual iteration of this scrum process. It's got a main part that lasts two to four weeks, and within this main part, there are many daily scrums that last 24 hours. So let's see how this work. A daily scrum is typically characterized by a 50-minute meeting at the beginning of the day for the team to sync, and what happens during the meeting is that there is a discussion of the accomplishments since the last meeting. A to do list for the next meeting is produced, and there is also an obstacle analysis. So if some problem appear, they're discussing the daily scrum, and possible solutions are proposed. At the end of the two four-week cycle, there is a

sprint review and retrospective. The sprint review normally consists of a four hour meeting. In the meeting, the product owner assesses the accomplishment for the specific sprint, and the team discusses issues that were encountered and solved. There is typically a demo of the deliverable for that sprint. And at that point, the product owner will also discuss the backlogs. And together with the team they will decide what to do next. In the retrospective conversely what happens is there is more focus on the process. So the goal of that part of the meeting is discussing possible process improvements. To identify them and if promising improvements are identified try to plan how to implement those improvements and use them in the next iterations. And something else that might happen at the end of a sprint is that if the product increment is good enough as it reach the state in which it can be actually shipped that will result in a release that is not just internal. To show the product owner the progress that can also be deployed and actually used in production. So one final consideration is that as you can see, XP and scrum are fairly similar, and that's because they're both agile development processes. So the main thing to keep in mind is that they both implement and enforce those ideas, values, practices, and characteristics that we saw when we discussed agile development process in general.


HIGH-LEVEL PROCESS
daily scrum
24 hours
Sprint review and retrospective
Potentially shippable Product increment
Product backlog
sprint Planning
sprint backlog
2-4 weeks
DONE