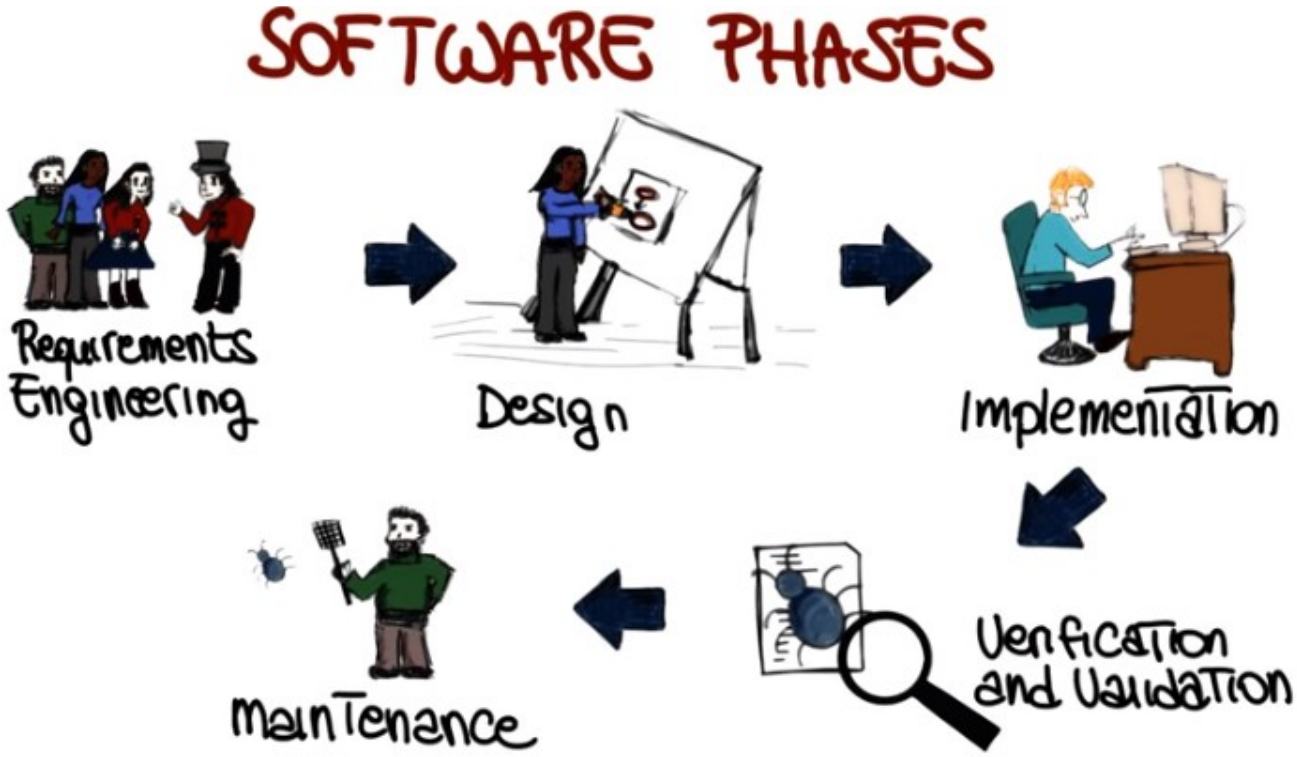
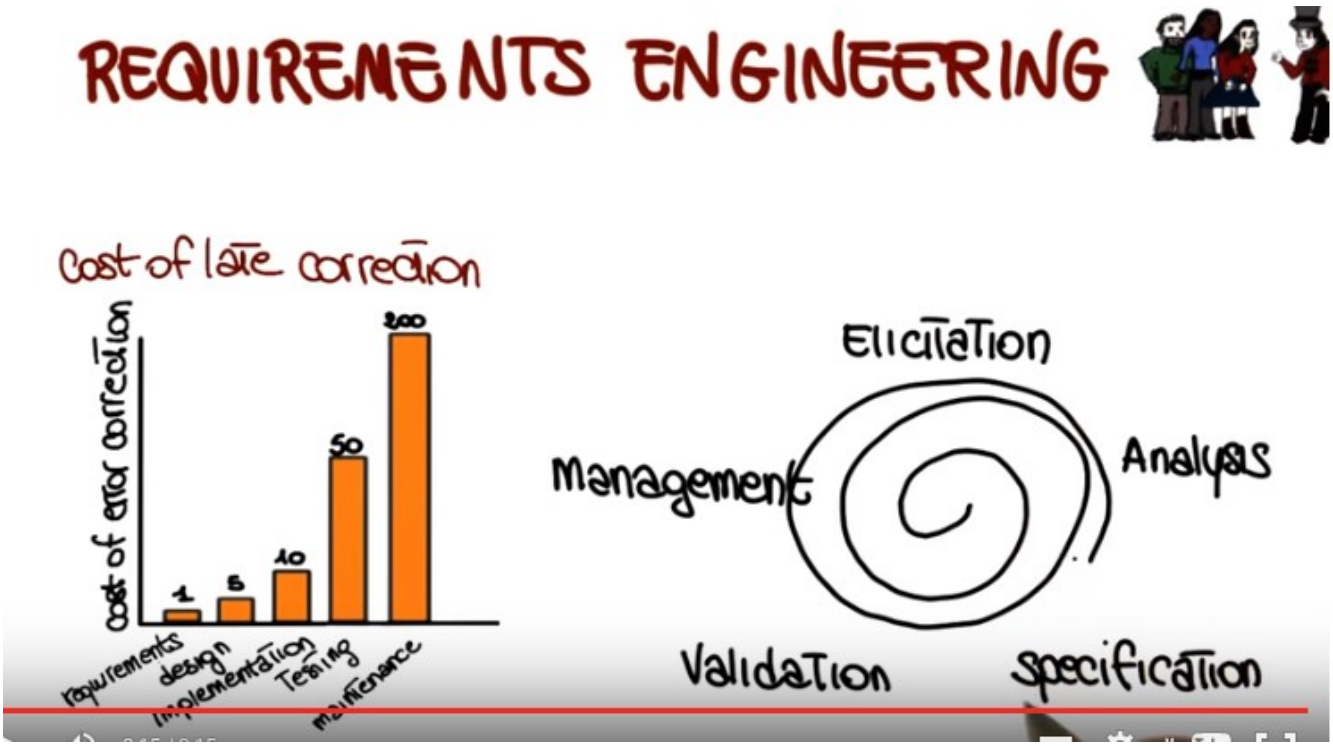
1. Hi, in the last lesson we provided an overview of the course and motivated the need for software engineering. In this lesson, we will present and start discussing several traditional software engineering life cycle models. We will talk about their main advantages, and also about their shortcomings. We will also talk about classic mistakes in software engineering that is well known in effective development practices, that when followed, tend to lead to better results. And covering those, will hopefully help us to avoid them in the future. And because in this lesson, I will discuss some fundamental aspects of software engineering, to suitably introduce these topics, I went to the University of Southern California, to interview one of the fathers of software engineering; Professor Barry Bohem. >> A well, a software life cycle is a sequence of, of decisions that you make, and it's fundamentally those decisions are going to be [INAUDIBLE] the history of the software that. You are going to build that other people are going to use, and the process model is basically answering the question of what do I do next and how long shall I do it for. And again, because there are a lot of different ways you can make that decision, you need to figure out which models are good for which particular situations. So, for example, we've, written a book that's called Balancing Agility and Discipline. It says under what conditions should you use agile methods, under which conditions should you invest more time in analyzing the situation and planning what you're going to do and the like. And so, typically if the project is, is small where it's three to ten people, agile works pretty well. If it's 300 people, then I think we don't want to go that way. If the affect of the defect is loss of comfort or limited funds, then agile is fine, but if it is a loss of life, then you don't. On the other hand if, if you have a situation where you have lot of unpredictable change, you really don't want to spend a lot of time writing plans and lots of documents. In some cases you may have a project where you want to do waterfall in some parts and agile in others. So, these are the kind of things that, that make the choice of life cycle process model very important and very interesting as a subject of concern.

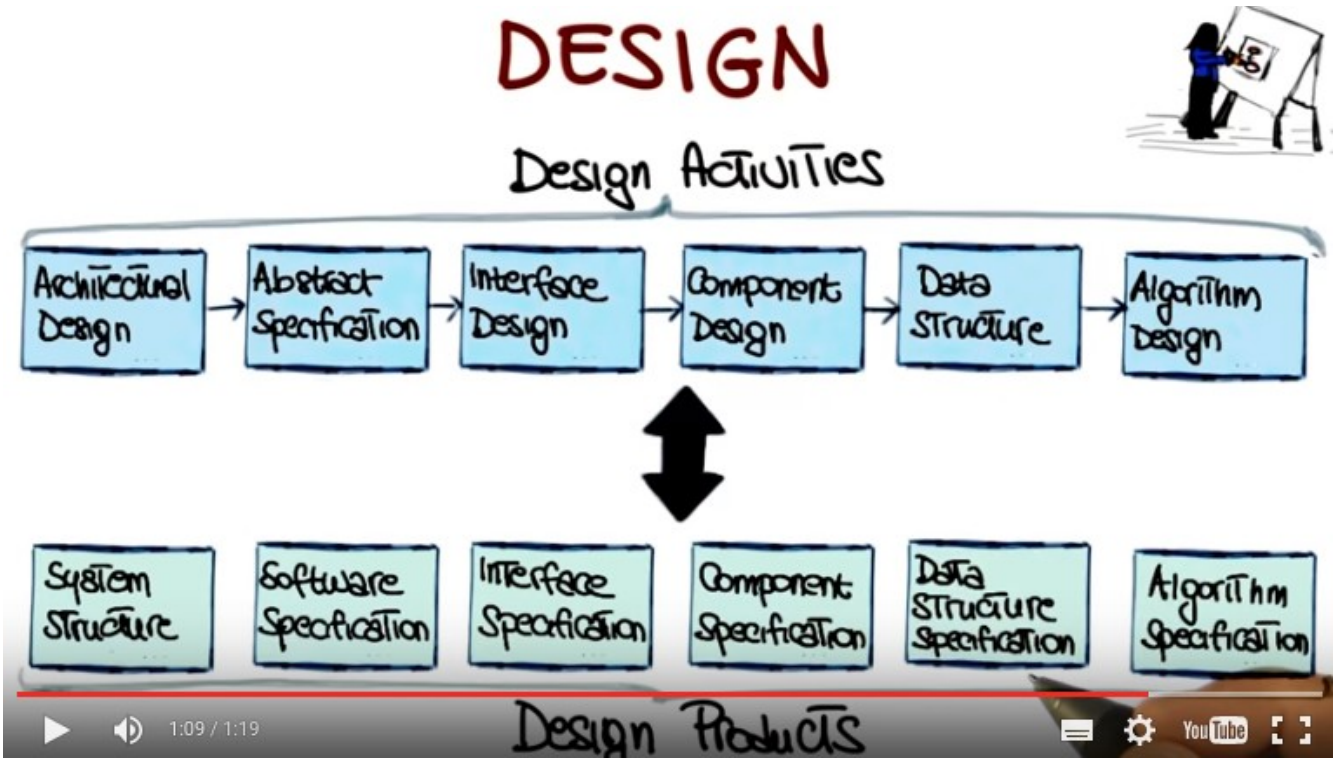
2. As we just heard from Professor Bohem, software engineering is an important and critical discipline, concerned with cost effective software development. We also heard that this is based on a systematic approach that uses appropriate tools and techniques, operates under specific development constraints. And most importantly, follows a process. As we discussed in the previous lesson, the software development process contains fundamental activities, or phases. Since we will discuss several processes, I'm going to remind you what these phases are. We start with requirements engineering, followed by design, implementation, verification and validation, and finally maintenance. Note that we will revisit each of these phases and devote an entire lesson or more to each phase. So what I want to do next is simply to give you a quick overview of what these phases are. Note also that for now I will follow a very traditional take on these topics. Later on in the class we will see how things can change and did change over the years.



3. So, let's start with requirements engineering, which is the field within software engineering that deals with establishing the needs of stakeholders that are to be solved by the software. So why is this phase so important? In general, the cost of correcting an error depends on the number of subsequent decisions that are based on it. Therefore, errors made in understanding requirements have the potential for greatest cost (好像跟下圖是相反的) because many other design decisions depend on them and many other follow up decisions depend on them. In fact, if we look at this diagram, which is again a qualitative diagram, where we have the cost of error correction over the phase in which the error is discovered. (下圖)We can see that if we discover an error in requirements it's going to cost us one. If we find it in in design it's going to cost us five and so on and so forth. And the cost grows dramatically as we go from the requirements phase to the maintenance phase. Why? Because of course if we discover a problem here (maintance) we're left to undo a lot of the decision that we had made before to correct the error. Whereas if we find an error here (requirments) we can correct it right away and we don't affect the subsequent phases. So how can we collect the right requirements. Traditional requirements in engineering does so through a set of steps. The first step is elicitation(引出) which is the collection of requirements from stake holders and other sources and can be done in a variety of ways, we will discuss some of them. The second is requirement analysis which involved the study and deeper understanding of the collective requirements. The third step is this specification of requirements, in which the collective requirements are suitably represented, organized and save so that they can be shared. Also in his case, there are many ways to do this, and we will see some of this ways when we talk about the requirements engineering in the dedicated lesson. Once the requirements have been specified, they can be validated to make sure that they're complete, consistent, no redundant and so on. So that they've satisfied a set of importance properties, for requirements. Finally, the fifth step is requirements management which accounts for changes to requirements during the lifetime of the project. And here I talked about steps, kind of giving the impression that we're just going from the first step to the fifth one and that this is sort of a linear process. In reality, as we will see, this is more of an iterative process in which will go and cover the different phases in an iterative fashion. We will discuss extensively requirements engineering in our second mini-course (即 P2L1 等).

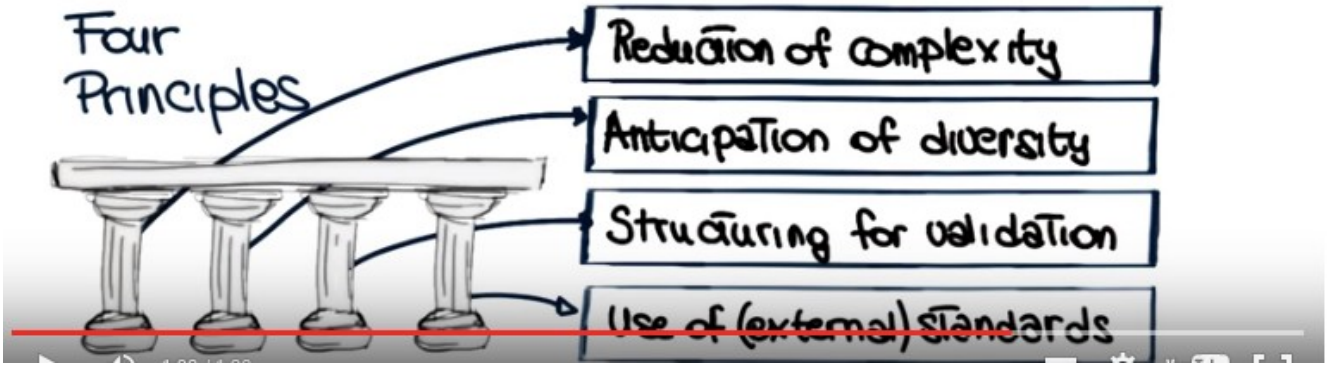


4. Now let's discuss the next phase of software development, which is software design. Software design is the phase where software requirements are analyzed in order to produce a description of the internal structure and organization of the system. And this description will serve as the basis for the construction of the actual system. Traditionally, the software design phase consists of a series of design activities(見下圖). Which normally consists of the architectural design phase, the abstract specification, interface design, component design, data structure and algorithm design. And notice that this is just a possible list of activities. But you can also characterize design activities in many different ways. And if you're looking at different books, and different sources, you might find different activities described. But the core idea, the important point is that we go from sort of a high-level view of the system, which is the architectural design, to a low-level view, which is the algorithm design. And these activities result in a set of design products, which describe various characteristics of the system. For example, they describe the architecture of the system, so how the system is decomposed and organized into components, the interfaces between these components. They also describe these components into a level of details that is suitable for allowing their construction. We will discuss the details of software design and talk extensively about these different actives and these different products in the third mini course of this class.

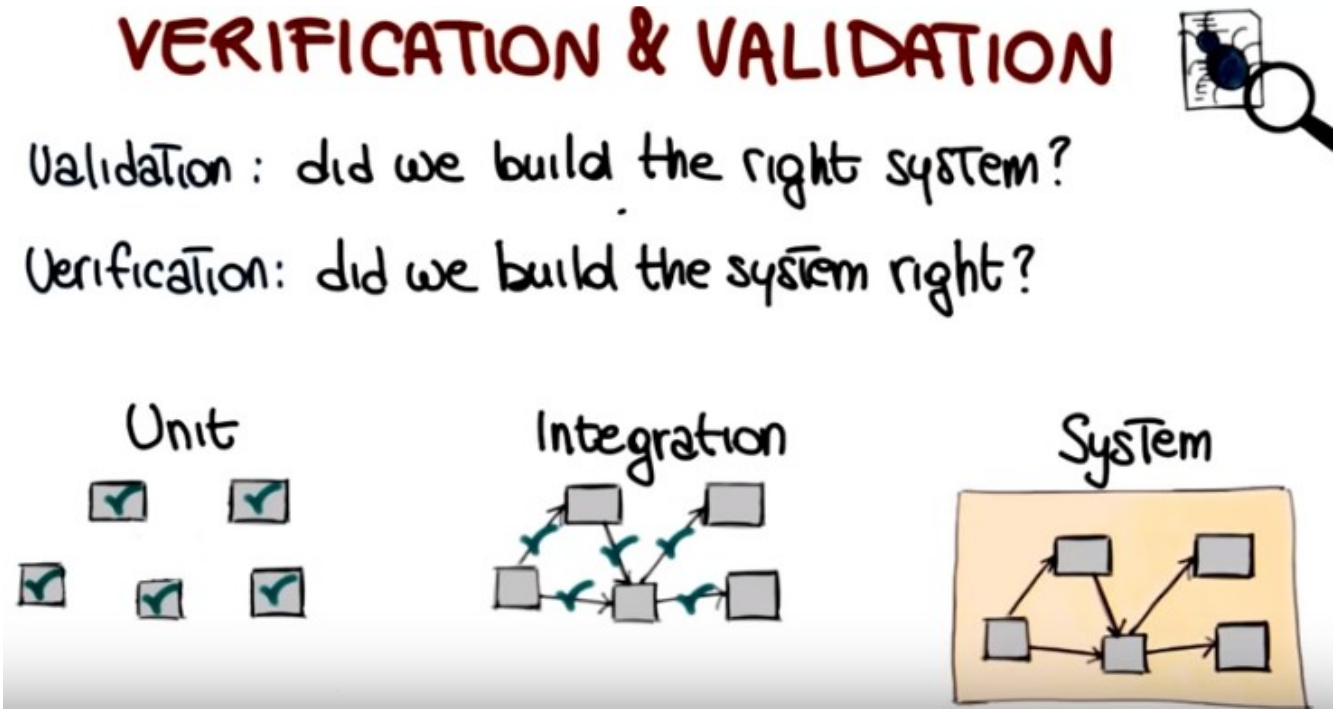


5. After we have designed our system we can implement it. In the implementation phase what we do is basically taking care of realizing the design of the system that we just created and create a natural softer system. There are four fundamental principles, four pillars that can affect the way in which software is constructed. The first one is the reduction of complexity. This aims to build software that is easier to understand and use. The second pillar is the anticipation of diversity. Which takes into account that software construction might change in various way over time. That is that software evolves. In many cases, it evolves in unexpected ways. And therefore, we have to be able to anticipate some of these changes. The third pillar is the structuring for validation. Also called design for testability. And what this means is that we want to build software so that it is easily testable during the subsequent validation and verification activities. Finally, and this is especially true within specific organizations and or domains. It is important that the software conforms to a set of internal or external standards. And some examples of this might be, for example, for internal standards, coding standards within an organization, or naming standards within an organization. As for external standards, if for example you are developing some medical software. There are some regulations and some standards that you have to adhere to in order for your software to be valid in that domain.

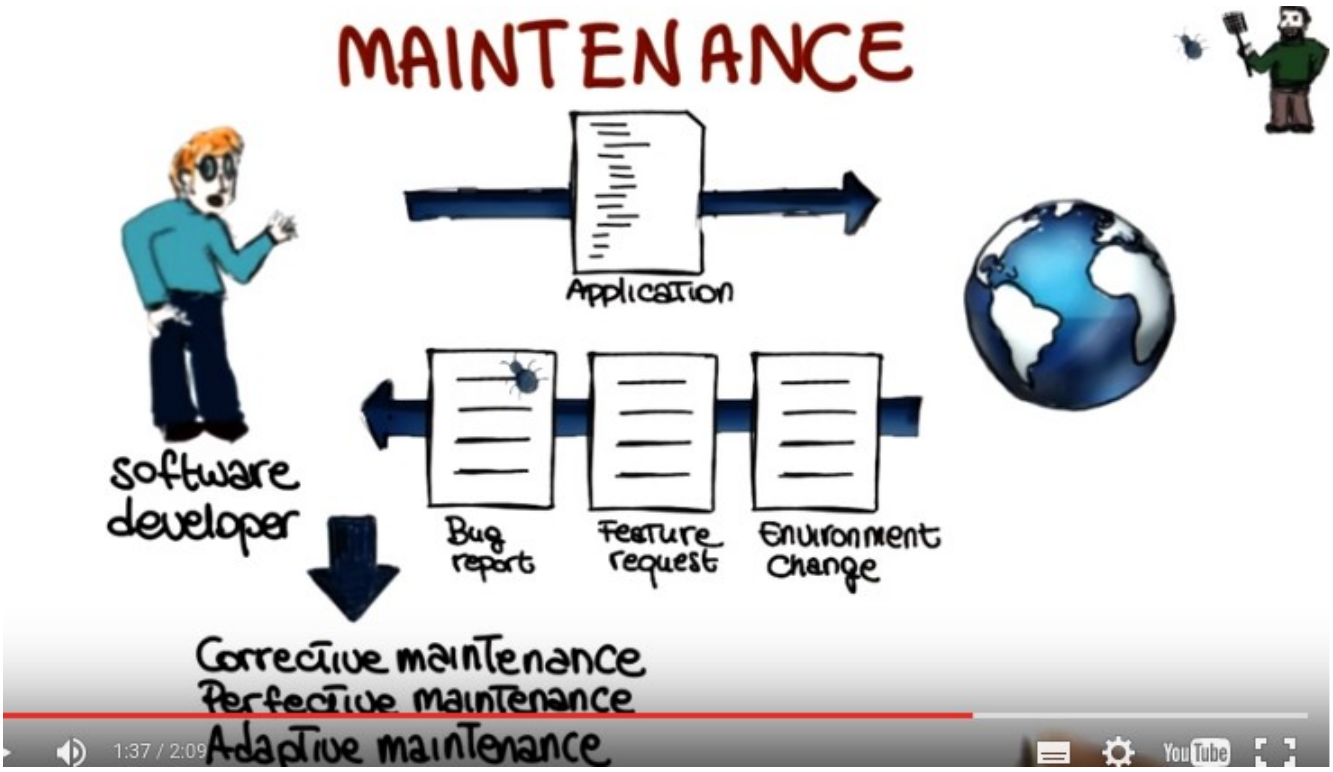
IMPLEMENTATION



6. After we have built our system, verification and validation is that phase of software development that aims to check that the software system meets its specification and fulfills its intended purpose. More precisely, we can look at verification and validation independently. And validation is the activity that answers the question did we build the right system. Did we build the system that the customer wants? That will make the customer happy. Whereas verification answers a different question which is did we build the system right. So given a description of the system that is the one that we derived from the customer through the collection of requirements and then design and so on, did we build a system that actually implements the specification that we defined? And when we look at verification there's many, many ways of doing verification and in fact in the mini course number four we will cover verification extensively. The only thing I want to mention here is the fact that verification can be performed at different levels. In particular, it can be performed at the unit level in which we test that the individual units work as a expected. Can be performed in the integration level in which what we test is the interaction between the different units. So we want to make sure that the different modules talk to each other in the right way. And finally, there is system testing in which we test the system as a whole and we want to make sure that all the system, all the different pieces of the system work together in the right way. And this is also the level at which then we will apply validation and some other testing techniques like stress testing or robustness testing and so on. And as I said I'm not going to say anything more on this topic because we will cover verification, and validation, and testing in particular in great details in mini course number four.



7. As we discussed before software development efforts normally result in the delivery of a software product that satisfies the user requirements. So normally our software development organization will release this application to its final users, however, once the software is in operation many things can happen. So, for example, the environment might change. There might be new libraries. There might be new operating systems in which our software has to operate. Or they may be future requests, so the users may find out that, guess what, they want to do something different with the problem that we gave them. Or, again, and this is one of the most common occurrences, users might find problems with the software and may file bug reports and send the bug reports back to the software developer. These are the reasons why software maintenance is a necessary phase in software development. Software maintenance is the activity that sustains the software product as it evolves throughout its life cycle, specifically in response to bug reports, feature requests and environment changes. Development organisations perform three kinds of maintenance activities: corrective maintenance to eliminate problems with the code, perfective maintenance to accommodate feature request, and in some cases just to improve the software, for example, to make it more efficient, and finally, adaptive maintenance, to take care of the environment changes. And after this activities(即 maintance, from hand pointing) have been performed, the software developer will produce a new version of the application, will release it and the cycle will continue through out the lifetime of the software. That's why maintenance is a fundamental activity and a very expensive one. And one of the reasons why maintenance is expensive, that I want to mention now, is regression(回歸) testing. During maintenance every time you modify your application you have to regression test the application, where regression testing is the activity of retesting software after it has been modified to make sure that the changes you perform to the software work as expected, and that your changes did not introduce any unforseen effect. I'm pretty sure that you're familiar with the case of a new version of the software being released and just a couple of days later another version being released to fix some problems that occor with the new version. These problems is what we call regression errors and they're what regression testing targets and tries to eliminate before the new version of the software is released into the world.



8. Okay. Now before we jump into the next topic, I just want to take a very quick and simple quiz just to make sure that you guys paid attention to what I just discussed. So I want to ask you what are the traditional software phases. Requirements engineering, design, abstraction, implementation, verification and validation. Or maybe design, optimization, implementation verification and validation and maintenance. Or requirements engineering, design, implementation, verification and validation, and maintenance.

9. And the answer is the third one. The traditional software phases which are the ones that we just discussed are requirements engineering, design, implementation, verification and validation, and maintenance.

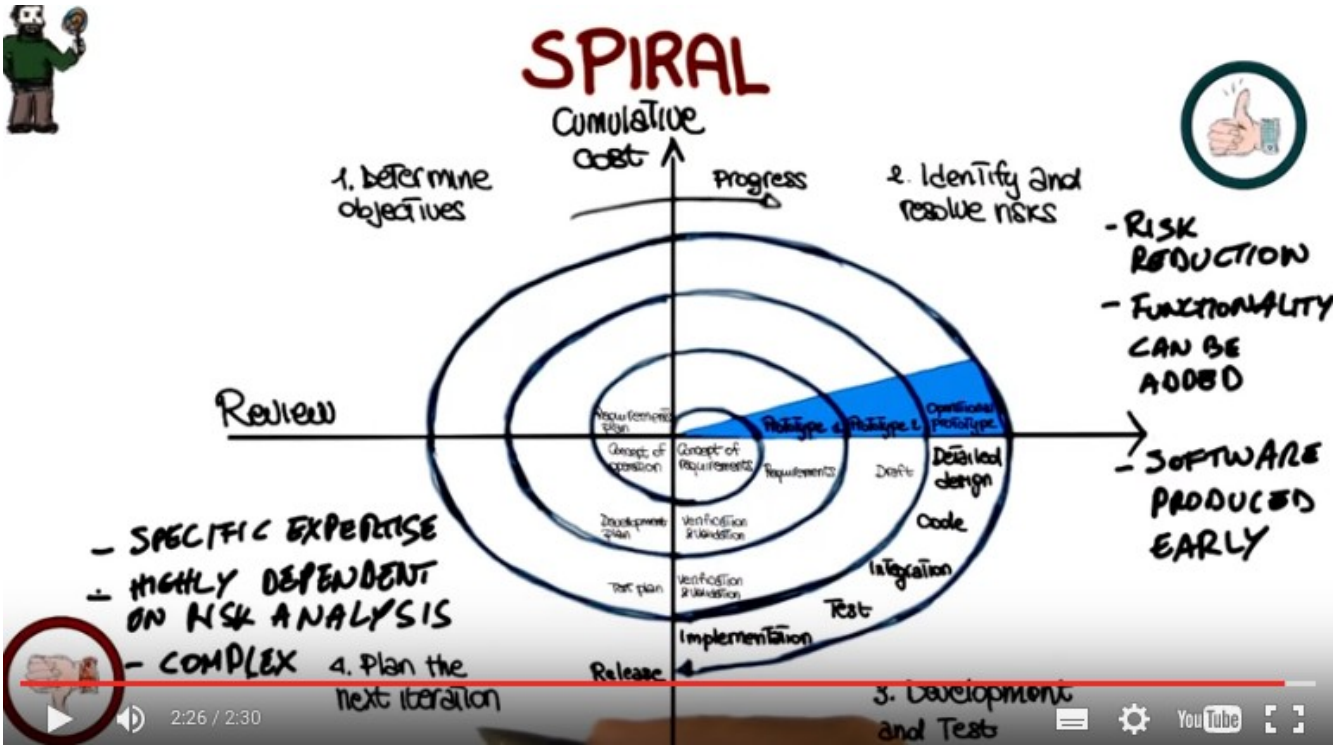
10. At this point, you know the possible activities, the possible phases performed during the software development process. But there this something that we still haven't discussed, which is very important. And that is how should we put these activities together to develop software? And this all comes down to the concept of software process model. Also called software lifecycle model. And what this is, is a prescriptive model of what should happen from the very beginning to the very end. Of a software development process. The main function of the life cycle model is to determine the order of the different activities so that we know which activities should come first and which ones should follow. Another important function of the life cycle model is to determine the transition criteria between activities. So, when we can go from one phase to the subsequent one. In other words, what the model should describe is what should we do next and how long should we continue to do it for each activity in the model. Now lets see a few traditional software process models. I will discuss them here at the high level and then revisit some of these models in the different mini courses.

11. The first model we want to discuss is the grandfather of all life cycle models. And it is the waterfall model. In the waterfall model the project progresses to an orderly sequence of steps. From the initial

software concept, down until the final phase. Which is system testing. And at the end of each phase there will be a review to determine whether the project is ready to advance to the next phase. The pure waterfall model performs well for softer products in which there is a stable product definition. The domain is well known and the technologies involved are well understood. In these kind of domains, the waterfall model helps you to find errors in the early, local stages of the projects. If you remember what we discussed, this (左上) is the place where we want to find errors, not down here because finding them here will reduce the cost of our overall software development. The main advantage of the waterfall model is that it allows you to find errors early. However, the main disadvantages of the waterfall model arise from the fact that it is not flexible. Normally, it is difficult to fully specify requirements at the beginning of a project. And this lack of flexibility is far from ideal when dealing with project in which requirements change, the developers are not domain experts or the technology used are new and evolving, that is it is less than ideal for most real world projects.

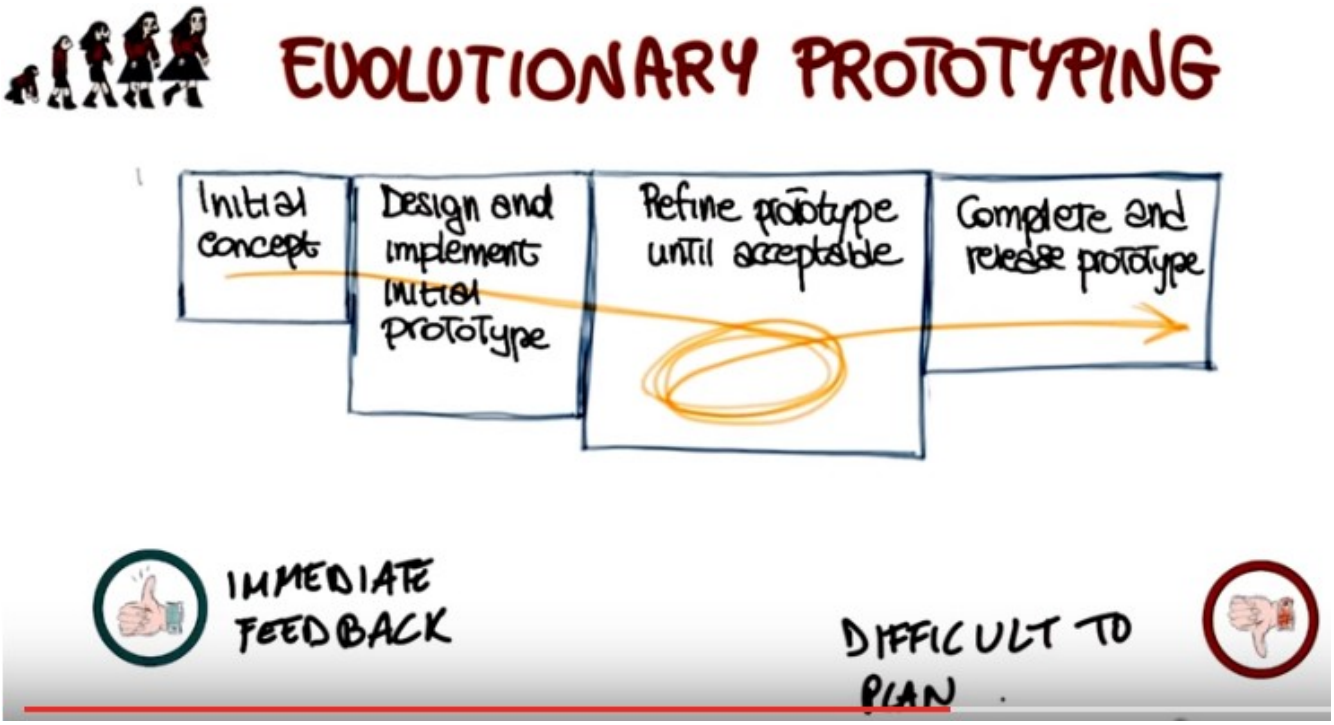
12. The next model that we will discuss is the spiral model, which was first described by Barry Boehm, which is the professor that we interviewed at the beginning of this lesson. In his paper from 1986 that was entitled A Spiral Model of Software Development and Enhancement. And one of the main characteristics of that paper is that it was describing the spiral model using a diagram, which is the one that I'm showing you here, and this diagram has become very very popular, and you probably saw it either in this form or one of the many variations of the diagram. So I'm not going to discuss all of the details of the spiral model, but I just want to give you an idea of its main characteristics. The spiral model is an incremental risk-oriented lifecycle model that has four main phases listed here: determine objectives, identify and resolve risks, development and tests, and plan the next iteration. A software project will go through these four phases in an iterative way. In the first phase, the requirements will be gathered. In the second phase, the risks and the alternate solutions will be identified, and a prototype will be produced. Software and tests for the software are produced in the development and test phase, which is the third step of the process. Finally, in the fourth phase, the output of the project, so far, is evaluated, and the next iteration is planned. So basically, what the spiral process prescribes is a way of

developing software by going through these phases in an iterative way, in which we learn more and more of the software, we identify more and more, and account for, more and more risks and we go more and more towards our final solution, our final release. There are several advantages of using a spiral model. The first one is that the extensive risk analysis does reduce the chances of the project to fail. So there is a risk reduction advantage. The second advantage is that functionality can be added at a later phase because of the iterative nature of the process. And finally, software is produced early in the software lifecycle. So, at any iteration, we have something to show for our development. We don't wait until the end before producing something. And then of course there's also the advantage that we can get early feedback from the customer about what we produced. The main disadvantages on the other hand of the spiral model, are that the risk analysis requires a highly specific expertise. And unfortunately, the whole success of the process is highly dependent on risk analysis. So risk analysis has to be done right. And finally the spiral model is way more complex than other models, like for example, the water fall model. And therefore it can be costly to implement.



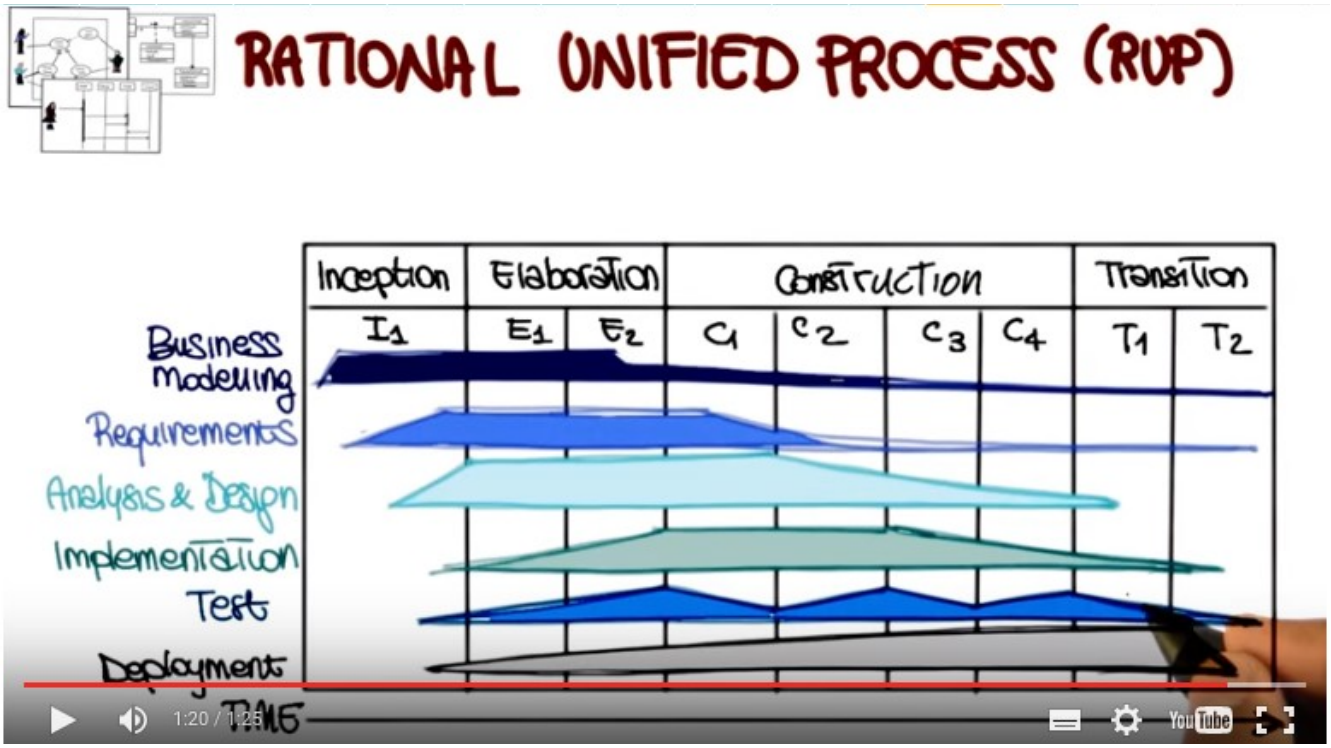
13. The next process model I want to discuss is evolutionary prototyping(prototype: 原型, 雛型), which works in four main phases. We start from an initial concept, then we design and implement a prototype based on this initial concept, refine the prototype until it is acceptable, and finally we complete and release the prototype. Therefore, when developing a system using evolutionary prototyping, the system is continually refined and rebuilt. So it is an ideal process when not all requirements are well understood. Which is a very common situation. So, looking at this in a little more details, what happens is that developers start by developing the parts of the system that they understand, instead of working on developing a whole system, including parts that might not be very clear at that stage. The partial system is then shown to the customer and the customer feedback is used to drive the next iteration, in which either changes are made to the current features or new features are added. So, either the current prototype is improved or the prototype is extended. And finally, when the customer agrees that the prototype is good enough, the developers will complete all the remaining work on the system and release the prototype as the final product. So let's discuss as we did for the previous process models,

what are the main advantages and disadvantages of evolutionary prototyping. In this case, the main advantage is the immediate feedback. Developers get feedback immediately as soon as they produce a prototype and they show it to the customer and therefore, the risk of implementing the wrong system is minimized. The main negative is the fact that it's difficult to plan. When using evolutionary prototype it is difficult to plan in advance how long the development is going to take, because we don't know how many iterations will be needed. And another drawback is that it can easily become an excuse to do kind of do cut and fix kind of approaches in which we hack something together, fix the main issues when the customer gives us feedback, and then continue this way, until the final product is something that is kind of working, but it's not really a product of high quality. Something else I want to point out before we move to the next software process model is that there are many different kinds of prototyping, so evolutionary prototyping is just one of them. For example, throwaway prototyping is another kind of prototyping in which the prototype is just used to gather requirements, but is thrown away at the end of the requirements gathering, instead of being evolved as it happens here.

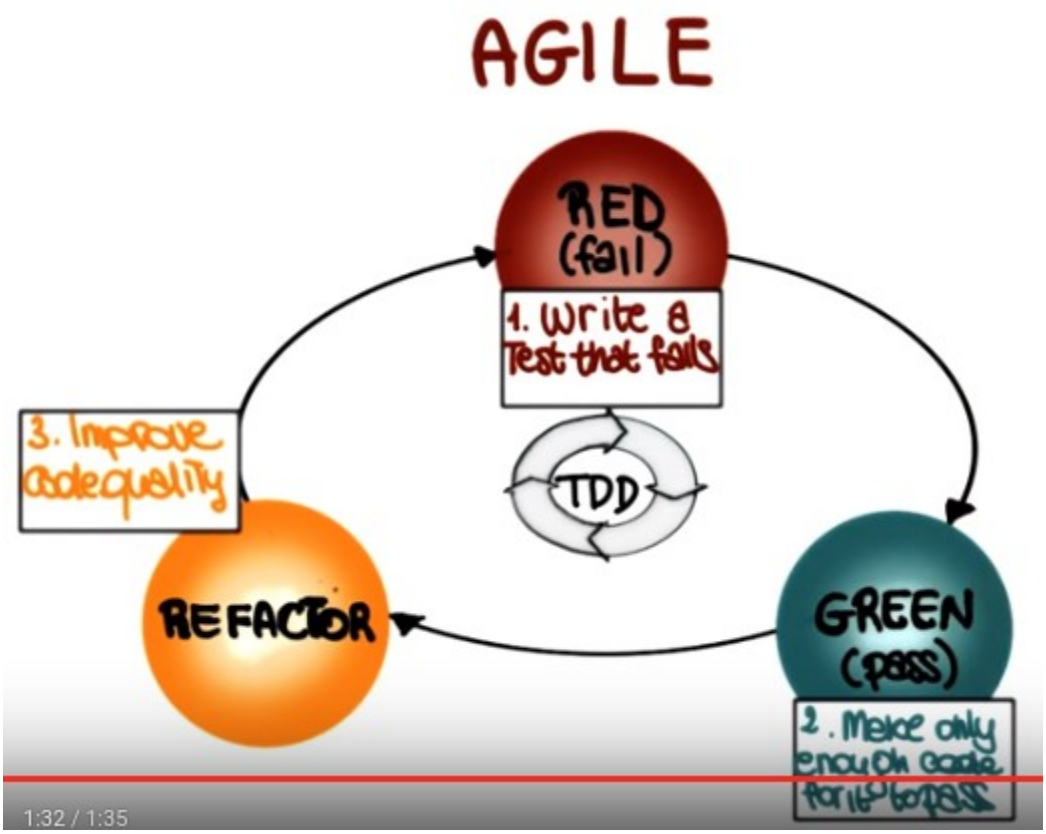


14. There are two more software process models that I want to cover, so bear with me. The first one is the Rational Unified software Process or RUP, which is s a very popular one based on UML. RUP works in an iterative way, which means it that it performs different iterations. And at each iteration, it performs four phases. So what I'm showing you here, is a high level view of the process. And I don't want you to focus on all the different details, because we will discuss these details later on, in a lesson that is actually dedicated to RUP. What I want to give you now, is just the gist of how this works. So, in each one of these four phases, which I'm going to describe in a second. We perform standard software engineering activities, the ones that we just discussed. And we do them to different extent, based on the phase in which we are. In particular, in the inception phase the work is mostly to sculpt the system. So basically figuring out what is the scope of the work, what is the scope of the project, what is the domain. So that we can be able to perform initial cost and budget estimates. The elaboration phase is the phase in which we focus on the domain analysis and define the basic architecture for the system. So this is a phase in which analysis and design are particularly paramount. Then there is a construction

phase, which is where the bulk of the development actually occurs. And as you can see here, is where most of the implementation happens. And finally, the transition phase is the phase in which the system goes from development into production, so that it becomes available to users. And of course, this is the phase in which the other activities in software development become less relevant and deployment(部署, 配置) becomes the main one.

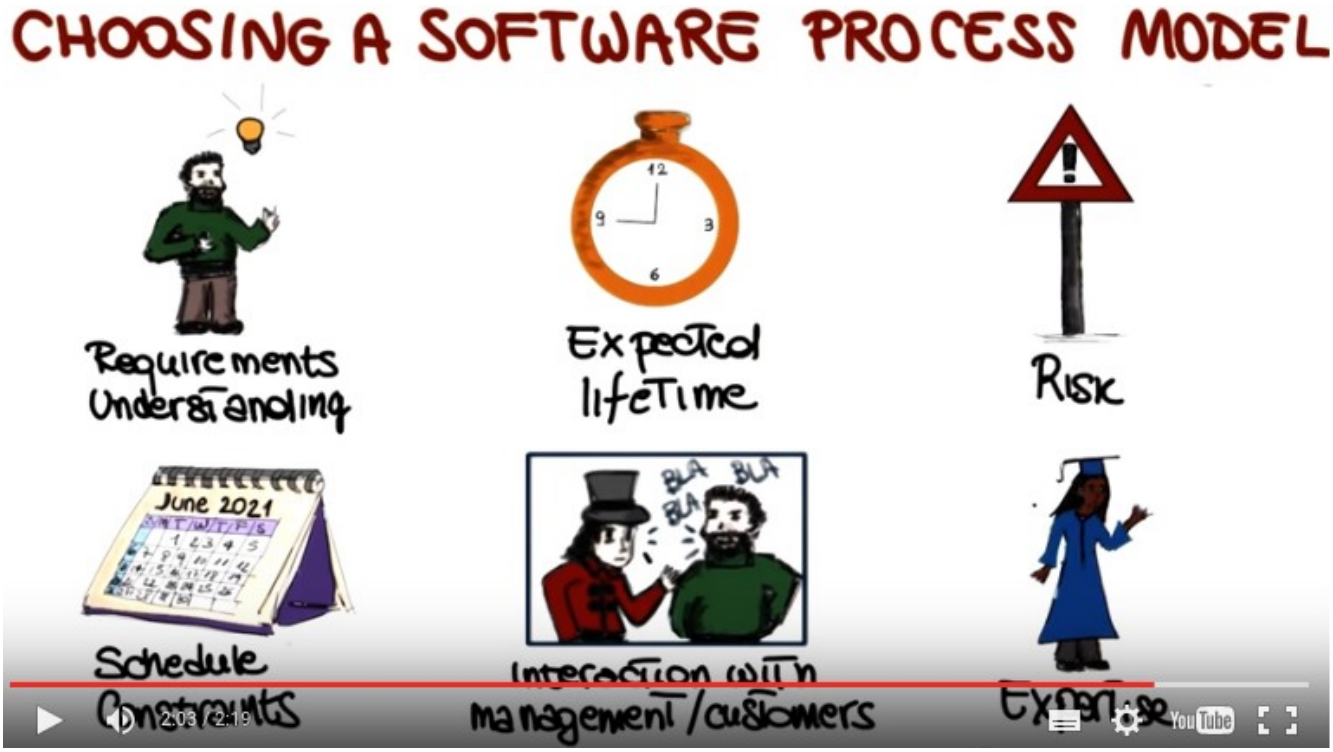


15. The next type of software process models that I want to discuss are Agile Software Development Processes. And this is a group of software development methods that are based on highly iterative and incremental development. And in particular, I'm going to discuss Test Driven Development or TDD. The space on the iteration of three main phases. In the first one that we mark as red, we write test cases that encode our requirements, and for which we haven't written code yet. And therefore, they will fail, obviously. So we're in this sort of red or fail phase. From this phase, we move to this phase, in which after we write the just enough code to make the test cases pass. We have a set of test cases that are all passing. And therefore, we can consider this as the green phase. We had enough code to make the test cases pass because the test cases encode our requirements. We have just written enough code to satisfy our requirements. When we do this over time though, what happens is that the structure of the code deteriorates, because we keep adding pieces. So that's why we have the third step, which is refactoring. In this step, we modify the code, and we will talk about refactoring extensively. We'll devote one lesson to it. We modify the code to make it more readable, more maintainable. In general, we modify to improve the design of the code. And after this phase, we will go back to writing more test cases for new requirements, write code that makes these test cases pass, and so on. So we'll continue to iterate among these phases. And also, in this case, we will talk about Agile Software Processes. And in particular, about extreme programming, or XP, and Scrum in more details, in minor course number four.

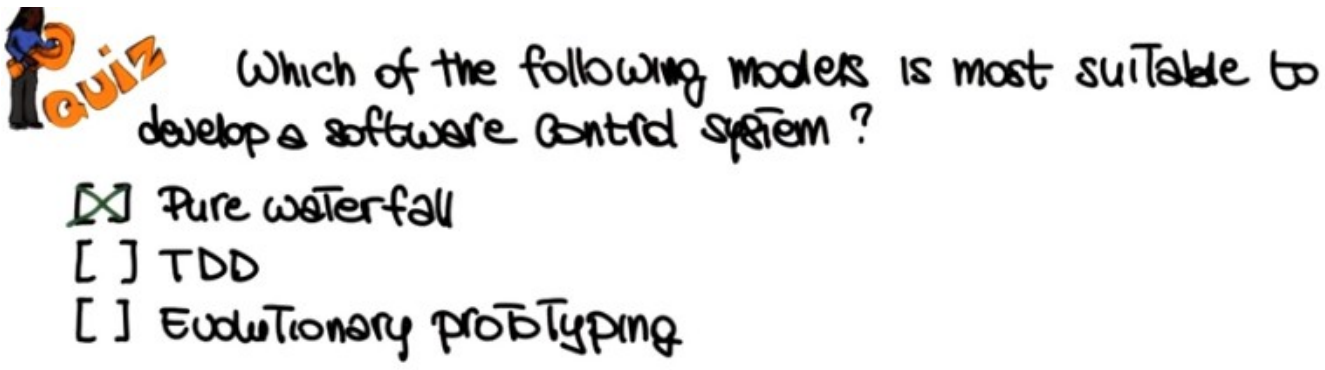


16. We just saw several software process models, and there are many, many more. And because these process models define the master plan for our project, the specific process model that we choose has as much influence over a project's success as any other major planning decision that we make. Therefore, it is very important that we pick the appropriate model for our development process. Picking an appropriate model can ensure the success of a project. On the contrary, if we choose the wrong model, that can be a constant source of problems and ultimately, it can make the project fail. So how can we choose the right model for a project? To be able to do so, we have to take into consideration many factors. In particular, we need to be aware of what level of understanding we have of the requirements. Do we understand all the requirements? Are we going to be able to collect all the requirements in advance, or collecting requirements is going to be hard and therefore, we might want to follow a process that is more flexible with that respect. Another important point is the expected lifetime of the project. Is this a quick project that we are putting together for a specific purpose or something that's going to last for for a number of years and that we're going to maintain over all those years? That's going to make a difference in the way we decide to develop that project. Also, what is the level of risk involved? Do we know the domain very well? Do we know exactly the technologies involved? Well, if so, we might go with a more traditional process. Otherwise, we might want to be more agile, more flexible. It is also very important to know the schedule constraints. How much time, how many resources do we have for this project? What is the expected interaction with the management and the customer? In particular for this ladder, there are many processes that rely on the fact that there can be a continuous interaction with the customer. If that interaction is not there, there's no way we are going to be able to use these processes. Conversely, there are processes that don't require the presence of the customer at all, except for the initial phase and maybe some checking points and so if the customer is very inaccessible, we might want to follow one of those processes, instead of one of the more demanding ones in terms of customer's time. Finally, it is important to take into account the level of the

expertise of the people involved. Do we have people that know the technologies that we're using? Do we know people that know a specific kind of process? Some processes require some specific expertise and we're not going to be able to follow that process if we don't have the right expertise. So we need to take into account all of these aspects, and sometimes more, in order to be able to make the right decision and pick the right software process model for our project.



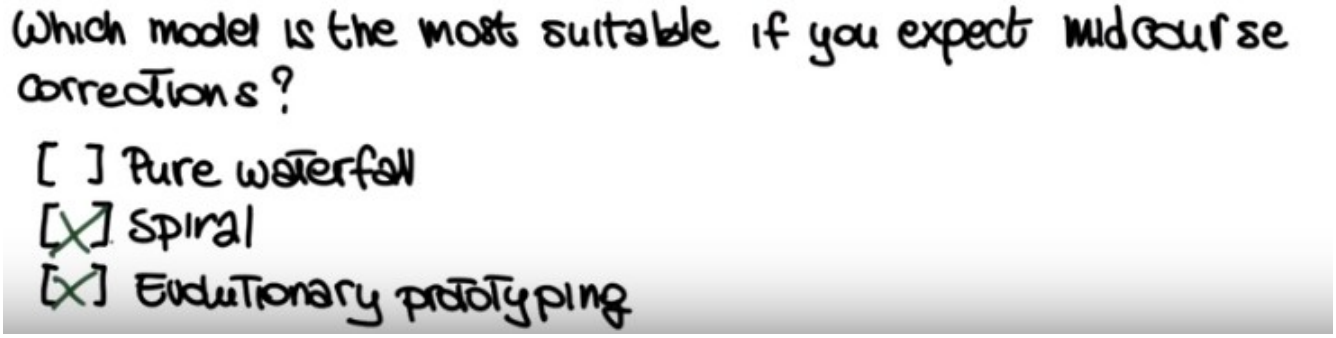
17. Now before we move to the last part of the lesson, let's have a quick quiz on software process models to make sure that we are all on the same page. So I am going to ask you two questions. The first question is which of the following models is most suitable to develop a software control system? And when you think about the software control system, you can think about for example the control system for the software in an airplane. Would you rather use a pure waterfall model? Test driven development? Or an evolutionary prototyping approach?



18. This is the context in which, typically, a pure waterfall process will work well. Why? Well, because it's a context in which requirements are usually well understood. The domain is well understood, so that

kind of system has been built many times before. And also, it's a system in which we don't expect requirements to change dramatically over time. Therefore, a waterfall model, in which we collect all the requirements at the beginning and then we move to the subsequent phases might be the most appropriate one. Probably we don't want to do evolutionary prototyping in the case of the control system for an airplane. Same thing holds for TDD, so we want to be a little more rigorous in those cases.

19. The second question I want to ask you is which model is the most suitable if you expect mid-course corrections? Would you rather use a pure waterfall model, a spiral model, or evolutionary prototyping?



20. In this case, I think about the spiral model, and evolutionary prototyping model will work. Definitely you don't want to have a pure water from water. Why? Well because it is very expensive with a pure waterfall model to make changes during the course of the project, especially changes that involve requirements. Why? Because we saw that it can be very expensive. Whereas with the spiral model, we saw that being iterative, we can actually make correction throughout development. Similarly, with evolutionary prototyping, we keep evolving our system based on the customer feedback. And therefore, if something changes, we will get feedback right away, and we will be able to adapt. So the key thing here is that anything that is iterative works better in the case of changing environments. So, situations in which your requirements, the situation, the project might change. Whereas waterfall is more appropriate for situations in which the requirements are stable, we know the domain, and possibly we also know the technologies involved.

21. Now that we discussed softer process models, there is another important point I want to cover, because it's going to be useful for your projects. Documenting the activities carried out during the different phases of the softer lifecycle, is a very important task. The documents that we produce are used for different purposes, such as communicative details of the software systems. To difference the colors, ensure the correct implementation of the system, facilitate maintenance, and so on. There are standardized document that are provided by IEEE that you can use for this purpose. However, they're kind of heavyweight. So for the project in this class, when we will need them, I will rather use this lightweight documents. That we created by modifying the original ones, and make them a little simpler. In this, our documents are actually used, while teaching this class in the past. So they're well tested and work well for the kind of projects that we will perform. I provide information on how to access these documents in the class notes.

22. Now we get to the final part of the lesson. And in this part I want to talk about well known, ineffective development practices. These practices, when followed, tend to lead to predictably bad results. So let's look at some examples of these classic mistakes. And we're going to start with mistakes involving people. And notice that there is a long list. So I'm going to discuss just a few of those mistakes. And I'm going to point you to more information on this topic in the class notes. And some of these mistakes are actually kind of entertaining. So I'll recommend that you look at the class notes and

go in more depth in this list. So the first people mistake I want to mention is the one that I define, heroics. And this refers to too much emphasis on can do attitudes, so this idea that one person by himself or by herself can do everything and can make the difference in the whole project. And unfortunately, this encourages extreme risk taking and discourages cooperation, which is plain bad for the project. For example, it might force people not to report schedule slips. It might force people to take on on too much responsibility. And normally, and I saw it happen many times, the final result is a failure. So what you want when you're developing a larger project, is actually to apply soft engineering principles. Have teams, have team work, and have cooperation among the different team members, without pointing too much on single individuals. Another classic mistake is to not create the right working environment. We all like to work in nice environments. And there is strong evidence that the working environments can play a big role in productivity. There is evidence that productivity increases when the workplace is nice, quiet, warm, and welcoming. Finally, some of the most important people relating mistakes are due to poor people management. For example, lack of leaderships, or leadership that is exercised using the wrong means in the wrong way, which can lead to very unhappy personnel and therefore, low productivity, or even people leaving teams. Another classic example of poor management is adding people to a project that is behind schedule (指 project is behind schedule), which never works. Why it doesn't work? Because these new people need to be brought up to speed, and that causes further delays rather than improving the situation with the project schedule.

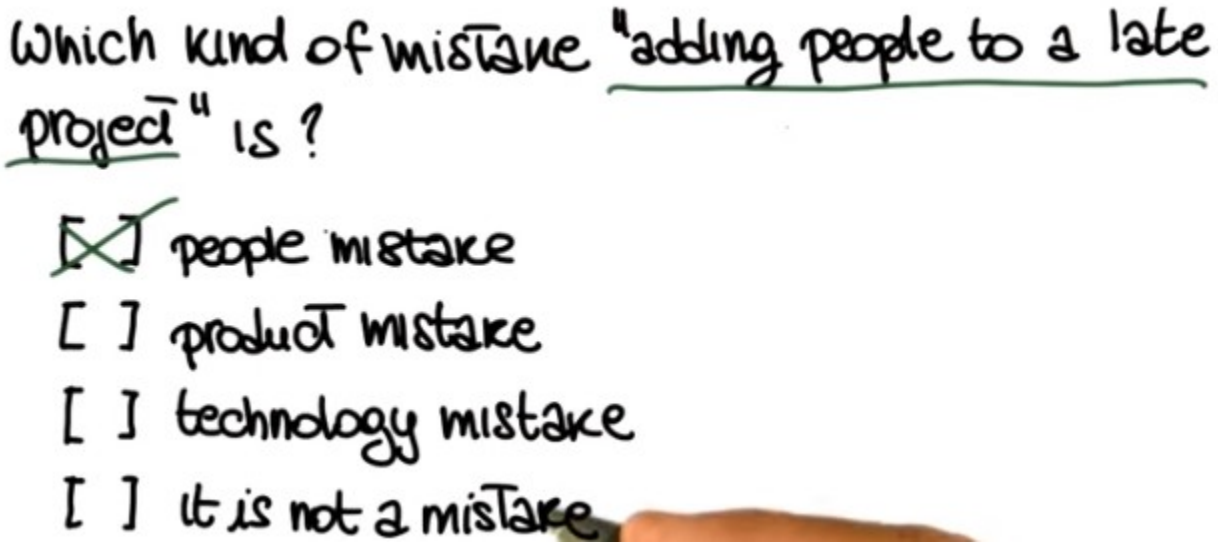
23. Another type of classic mistakes are process-related mistakes. And also in this case, these kind of mistakes can be due to many reasons. And they are of many types. One typical example are scheduling issues, which are due to the fact of being unable to come up with a realistic schedule. So to have an overly optimistic schedule. And this can be because we underestimate the effort involved in different parts of the project. Because we overestimate the ability of the people involved. Because we overestimate the importance, for example, of the use of tools. But no matter what the reason is, the result is typically that the projects end up being late, which is a very common situation. So this is somehow related to planning. And in general, planning is a fundamental factor in software processes and in software development. Mistakes in planning, such as insufficient planning or abandoning planning due to pressure, usually lead inexorably to failure. And speaking of failures, often there are unforeseen failures. Such as failures on the constructor's end, for example, that might lead to low quality or late deliverables, which ultimately affects the downstream activities.

24. The third category of mistakes that I want to mention is product-related mistakes. A typical example of product-related mistake is gold plating(鍍金) of requirements. And what that means is basically is that it's very common for projects to have more requirements than they actually need. For example, marketing might want to add more features than the ones that are actually needed by the users. And of course having more requirements lengthens the project's schedule in a totally unnecessary way. Feature creep(爬行) is another common mistake and consists in adding more and more features to a product that were not initially planned and are not really needed in most cases. And here there is evidence that the average project experiences 'about a 25% growth in the number of features over its lifetime which can clearly highly effect the project schedule'. Finally, if you're working on a project that strains(拉緊, 盡力) the limits of computer science. For example, because you need to develop new algorithms for the project, or you have to use new techniques. Then that project might be more research than actual development. And therefore, it should be managed accordingly. For example, by taking into account that you will have a highly unpredictable schedule.

25. The final type of classic mistakes that I want to mention are technology related mistakes. One typical mistake in this context is the silver-bullet syndrome. What does that mean? Well, the silver-bullet syndrome refers to situations in which there is too much reliance on the advertised benefits of

some previously unused technology. For example, a new technology. And the problem here is that we cannot expect technology alone to solve our software development issues. So we should not rely too much on technology alone. Another typical mistake is to switch or add tools in the middle of a project. And sometimes it can make sense to upgrade a tool, but introducing new tools, which can have a steep learning curve, has almost always negative effects. Finally, a common unforgivable mistake is the lack of an automated version control system for your code and for your various artifacts(人造物品). Manual and ad hoc solutions are just not an option. It is way too easy to make mistakes, use out of date versions, be unable to find a previous working version, and so on. I saw that happening many times, and it always results in a disaster. So be warned, use a version control system and an automated one. And actually we will use version control systems in our projects.

26. To conclude this lesson, I'm going to have a simple quiz and what I'm going to ask you is, which kind of mistake adding people to a late project is? And you can pick between a people mistake, a product mistake, a technology mistake, or maybe this is not a mistake at all, it is actually okay to add people to a project that is late.



27. You probably got this one right. The right answer is that this is a people mistake. And despite the fact that this is an easy answer, I just want to make sure to stress once more. Because this is a very classic mistake. And one that can have dire consequences. You should never add people, to a late project. Because in 99.9% of the cases, that's only going to make things worse. Why? Because these people have to be brought up to speed, and also because having more also makes the communication more difficult, the meetings more difficult and so on. So in short, do not add people to a late project.