(This color is Blue 4)

1. Hi, everybody, and welcome to the first lesson of the Software Engineering Course. In this introductory lesson I will first provide an overview of the whole course and then try to answer two important questions about software engineering, which are, what is software engineering and why do we need it? And to spice up the content a bit I will also interview several experts in the software engineering field from both academia and industry and ask them these very questions. So without any further ado, let's begin the lesson.

2. So we're going to start with the course overview. In this course we will cover several aspects of software engineering. We will discuss the concept of software process and study different types of processes from rigid to more agile. We will also look at the different activities or phases in the software development process and we will look at those in detail. Throughout the class set we will also learn how to use modern tools that can help improve programmer's productivity. Finally, we will put what we learn into practice through a set of realistic projects of increasing complexity. So now let me quickly go after the course structure. The course is divided into four courses, just like this cake is divided into four main parts, in four slices. Each course has a main topic and a project related to that topic. Specifically, the first mini-course introduces software engineering and presents its basic concepts, including the different software phases and the different types of development processes. The other three courses focus on a specific activity within software development and discuss a software process for which that activity is particularly relevant. The first one focuses on requirements and on prototyping, the second one on design and on the unified software process. And the third one is mostly focused on testing and on test-driven development. And as I just mentioned on top of the mini courses we will also introduce several state-of-the-art tools that we will use for the class and for the projects. Before we jump into the course let me also discuss some important requirements for the class. First of all, you will need to be able to install tools on your computer. When applicable we also try to provide you with suitably prepared visual machines that you can use out of the box. Second, most projects will require teamwork. However, and this is very important, assignments should be done individually, and not as a team effort. Third, projects and assignments will involve the submission of code, documents, or both. We will clearly tell you what, and how, to submit in due time. Finally you're strongly encouraged to read the recommended materials included in the notes for the various lessons.

3. First, let me start by asking a couple of very natural questions that you might have when considering whether to take this course. The first one is what is software engineering. And the second, very related one, is why do we need it? So what I did was actually to go out and ask some of the main experts in the field, both in academia and industry, these very questions and let's see what they said. What is software engineering and why is it important? >> Okay, can I start with another question? >> Of course. >> Okay, first what is a computer? It's a programmable device. So the essence of computing is programming. So program development is basically the most essential use of the computer. So software engineering is the discipline that investigates program development. So, how can it been done more efficiently? What's the best way of doing program development? And how can you develop reliable programs? So that's how I would define it. But I consider any software development activity software engineering activity >> Software engineering is the systematic application of methods to build software in a rigorous way. And I think one of the aspects that I like to bring into the notion of software engineering is that it's something that involves not only kind of technically building the system but understanding the requirements, working with stake holders. Trying to find a solution that balances all of the stakeholder needs in order to deliver the software thats tested and its rigorous to meet the needs of a stakeholder. Well, software engineering is the whole process of creation of software using engineering principles. >> My view is kind of a holistic view and I think about it from the perspective

of how is software engineering different from programming. So, I think that research about programming is all about the create part of software. And that software engineering is about the entire life cycle. So, that's one aspect. And the other aspect of the definition is it's about quality, the quality of software. Software engineering even considers things long after you ship which we all know is one of the, it is the largest economic piece of software development. >> So, improve, software engineering process for better software productivity and quality. >> The set of activities that one engages in when building software systems or software products. It's fundamentally a venue-creating activity. It involves social processes. >> Software engineering is the act of many people working together and putting together many versions of large and complex systems. And our world depends on software, software is immensely complex and we need many, many smart people to build these things. >> Well, engineering I think is the activity of envisioning and realizing valuable new functions with sufficient and justifiable confidence that the resulting system will have all of the critical quality attributes that are necessary for the system to be a success. And software engineering is the activity of doing this not only for the software components of engineering systems but for the system overall, given that it's so heavily reliant on it's underlying software technologies. >> So, I would say software engineering is the kind of art and practice of building software systems. >> Software engineering, in a nutshell, is a set of methods and principles and techniques that we have developed to enable us to engineer, or build, large software systems that outstrip or outpace one engineer's or even a small team of engineer's ability or abilities to understand and construct and maintain over time. So it requires a lot of people, it requires a long, term investment by an organization or a number of organizations, and often times it requires support for systems that that are intended for one purpose but end up getting used for many additional purposes in addition to the original one. >> Software engineering is about building and constructing very large-scale high-quality systems, so the high quality is the big issue. >> Software engineering is engineering discipline of developing software-based systems, usually embedded into larger systems composed of hardware and and humans [LAUGH] and business processes and processes in general. And why is that important? Well, because software is pervasive in all industry sectors and therefore systems must be reliable, safe and secure. >> Why can't we just get that by sitting down and writing software? >> Well, you could if software was small and simple enough to be developed by one or two people together in a room. But software development now is distributed, involves teams of people with different backgrounds who have to communicate with each other. It also involves customers, clients, users. Software engineers have to work with hardware engineers, with domain experts and therefore, well, no, we can't simply sit down and start coding. >> Software engineering is mostly being able to program. And you need to be able to put big systems together so that they actually work. That's my simple definition. >> And if you don't use software engineering practices, you're not going to be able to put them together? >> Well, you're not going to be able to reliably put them together. So basically, you could maybe hack something up, but it's not going to necessarily stand the test of time. If somebody wants to change it it's probably going to break. >> It's important because if you don't think about how you're building this system and how you're trading off different aspects, like performance and scalability and reliability, then it's going to end up breaking or not lasting very long or not, not doing everything that you want it to do, or being really expensive. >> If it's not done in a principled way it will be bad and every user will suffer. That's why we need software engineering. >> Why is it important? Because, I mean these two goal, productivity, faster, in developing software. And higher quality would be apparently important. Software is everywhere. >> It's important because we use software in everyday life. Everything's built on software systems. And these are ubiquitous across our society. >> It's important because software is everywhere around us and the way we build it, and the way we maintain it, is something that determines almost a basic quality of life nowadays. And getting that software right can make a difference, oftentimes, between a really fun product and one that you won't like to use a reasonably successful company, or one that fails. And in more extreme cases even the difference between life and death, if you think about the software that runs in the airplane on which

many of you fly on a regular basis. >> There are programs out there that if they screw up we are all screwed. >> Software engineering is crucially important because it's the engineering discipline that is uniquely capable of carrying out the engineering mission for software reliant systems. >> In the U.S we've all seen an unfortunate example with a system that went badly wrong in healthcare.gov and that system wasn't engineered correctly. And I think if we look at the reasons for that, trace them back to somewhere at the intersection between requirements and architecture and politics and project management, and all of these things are important concepts that have to go into the software engineering mix. >> It would end up in lots and lots of chaos because people wouldn't know how to organize themselves and wouldn't know how to organize software. Many of software engineering has very simple rules that you need to apply properly in order to get things done. And people who look at these rules and think, these rules are so super simple. This is totally obvious. But once you try to apply them, you'll find out they're not obvious at all. >> Now that we've heard these experts, let me show you an example that illustrates what can happen when software engineering practices are not suitably applied. [NOISE].

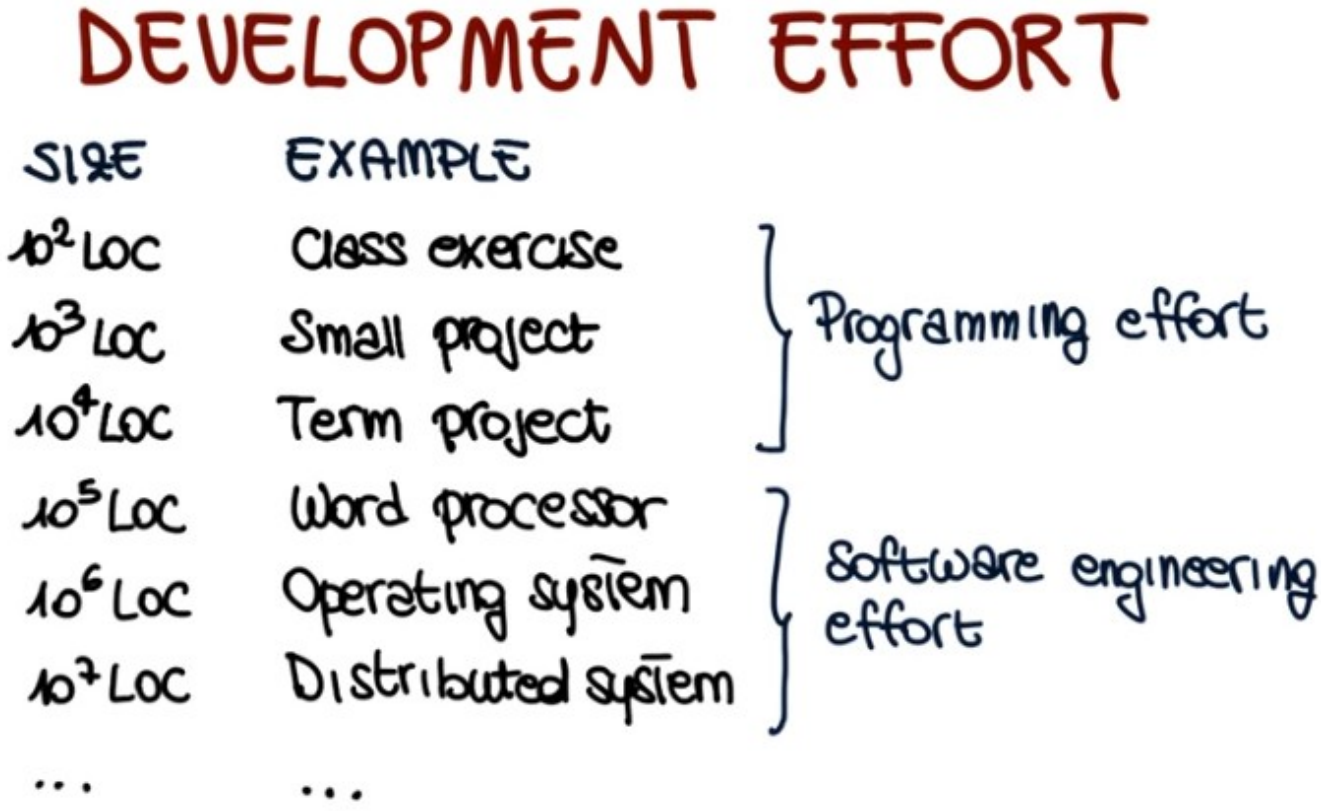
4. Now that you watched this small video, I like to ask you, what is this? Do you think it's fireworks for the 4th of July celebration, or maybe it was a flare gun in action, or maybe again it was the explosion of the Ariane five rocket due to a software error. What do you think? And in case it helps, I'm also going to show you an actual picture of this event.

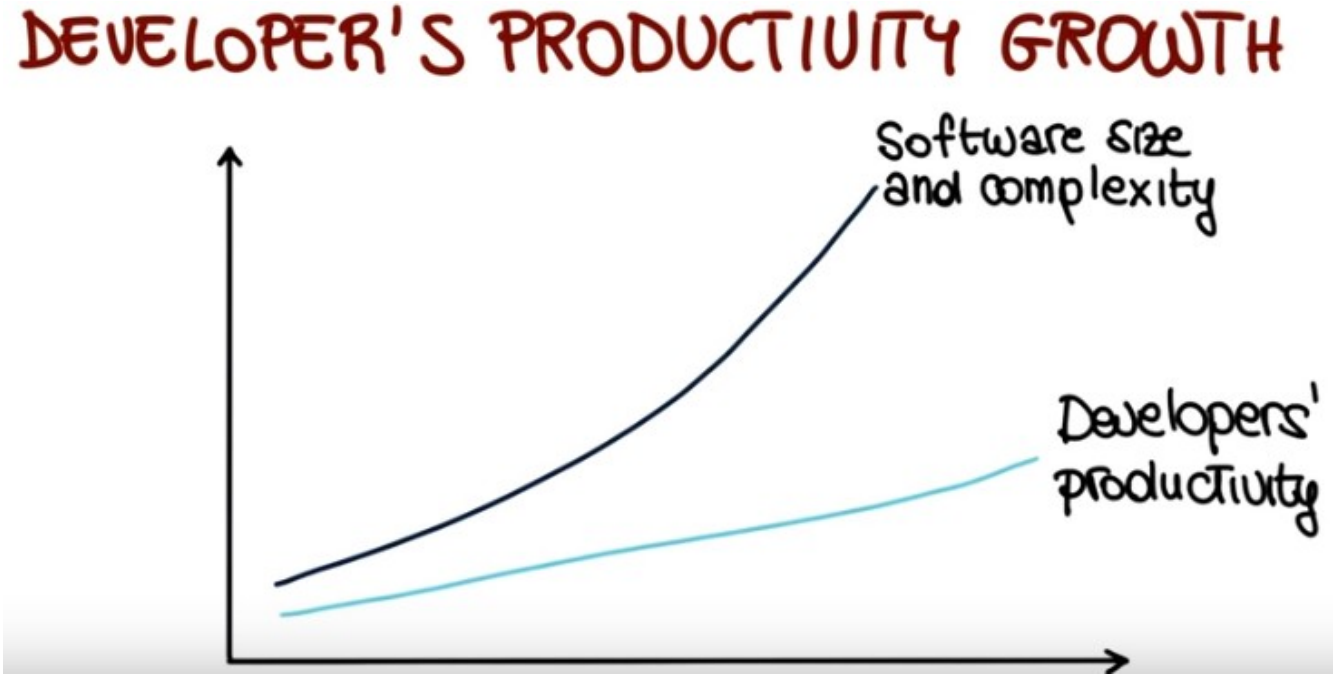
5. As you probably guessed, these are not fireworks for the 4th of July but, rather, the explosion of the Ariane 5, which happened 30 seconds or so after takeoff due to a software error. And this is just an example of what can go wrong when we don't build software and we don't test and verify and perform quality assurance of software in the right way, and quite an expensive one. In fact, to develop and to build the Ariane 5 it took 10 years. The cost was around $7 billion and there were $500 million of cargo on board. Luckily, at least there were no humans on the rocket. And you can find more details in case you're interested about the Ariane 5 accident in the lesson notes. I put a couple of links there.

6. And even if we don't go to these extreme examples, I'm sure that you have all experienced software problems, typically manifested in what we call a crash. And that crash might happen while you're finishing your homework or that three-page long email that you were preparing for the last two hours. But why's it so difficult to build software, or better, why's it so difficult to build good software? And how can we do it? This is exactly the topic of this class. And the reason why software engineering is a fundamental discipline in computer science. To motivate that, in this class, we will study a set of methodologies, techniques, and tools, that will help us build high quality software that does what it's supposed to do. And therefore, makes our customers happy. And that does it within the given time and money constraints. So within the budget that is allocated for the software. Before jumping into today's software engineering techniques though, let me take a step back and look at how we got here, as I believe it is very important to have some historical perspective on how this discipline was born and how it was developed over the years.

7. To do that we'll have to go back in time to the late 60s. So what was happening in the 60s? Well for example the first man landed on the moon. That was also time when Woodstock took place and also the time when the first 60 second picture from Polaroid was created. Concurrently to these events, which you probably didn't witness in first person, that was also the time when people started to realize that they were not able to build the software they needed. This happened for several reasons and resulted in what we call the software crisis. So let's look at some of the most important reasons behind this crisis. The first cause was the rising demand for software. Now you're used to see software everywhere: in your phone, in your car, even your washing machine. Before the 60s, however, the size and complexity

of software was very limited and hardware components were really dominating the scene. Then things started to change and software started to be increasingly prevalent. So we move from a situation where everything was mostly hardware to a situation in which software became more and more important. To give an example, I'm going to show you the growth in the software demand at NASA along those years. And in particular, from the 1950s to more or less 2000. And this is just a qualitative plot but that's more or less the ways things went. So the demand for software in NASA grow exponentially. And the same happened in a lot of other companies. For example, just to cite one, for Boeing. So the amount of software on airplanes became larger and larger. The second cause for the software crisis was the increasing amount of development effort needed due to the increase of product complexity. Unfortunately, software complexity does not increase linearly with size. It is not the same thing to write software for a class exercise or a small project, or a temp project, than it is to build a software for a word processor, an operating system, a distributed system, or even more complex and larger system. And what I'm giving here is just an indicative size for the software so the class exercise might be 100 lines of code, the small project might be 1000 lines of code, in the other thousand lines of code, and so on and so forth. For the former, the heroic effort of an individual developer can get the job done. So that's what we call a programming effort. If you're a good programmer, you can go sit down and do it, right. For the latter, this is not possible. This is what we called the software engineering effort. In fact, no matter how much programming languages, development environments, and software tools improve, developers could not keep up with increasing software size and complexity. Which leads us to the third problem that I want to mention and the third reason for the software crisis. And this cause is the slow developer's productivity growth. So let me show this again with a qualitative diagram. And this is taken from the IEEE Software Magazine. And what I'm showing here is the growth in software size and complexity over time, and how the developers' productivity really couldn't keep up with this additional software complexity, which resulted in this gap between what was needed and what was actually available.





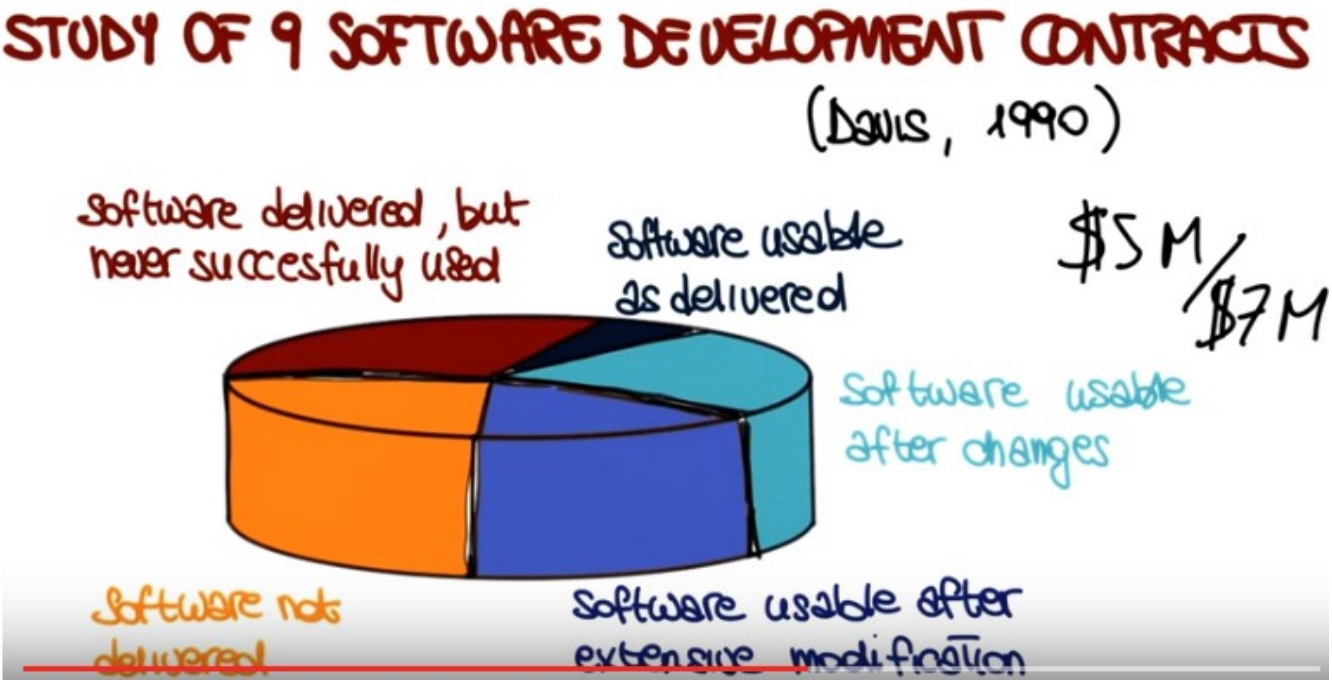
8. So now let's take a quick break and have a recap of what we just discussed. I want you to think about what are the major causes of the software crisis. I'm going to provide you a set of possibilities and I would like for you to mark all that apply. Was that increasing costs of computers? Was it increasing product complexity, or maybe the lack of programmers? Or was it, instead, this slow programmers productivity growth? The lack of funding for software engineering research? The rise in demand for software? And finally, was it maybe the lack of caffeine in software development organizations? Again, mark all that apply.

What are the major causes of the software crisis? (mark all that apply)

- [ ] Increasing cost of computers
- [x] Increasing product complexity
- [ ] Lack of programmers
- [x] Slow programmers' productivity growth
- [ ] Lack of funding for sofware engineering research
- [x] Rising demand for software
- [ ] Lack of caffeine in software development organizations

9. So, if you think about what we just discussed. Definitely one of the causes was the increasing product complexity. Products were becoming more and more complex and software was replacing more and more, what was before, provided by hardware components. Slow productivity growth was another problem, because programmers could not keep up with the additional complexity of the software that they had to develop. I would like to say there was lack of funding for software engineering research because I'm a software engineering researcher, but that was not one of the reasons for the software crisis. Instead, it was the rising demand for software. Again, more and more software was being required and more and more software was replacing hardware.

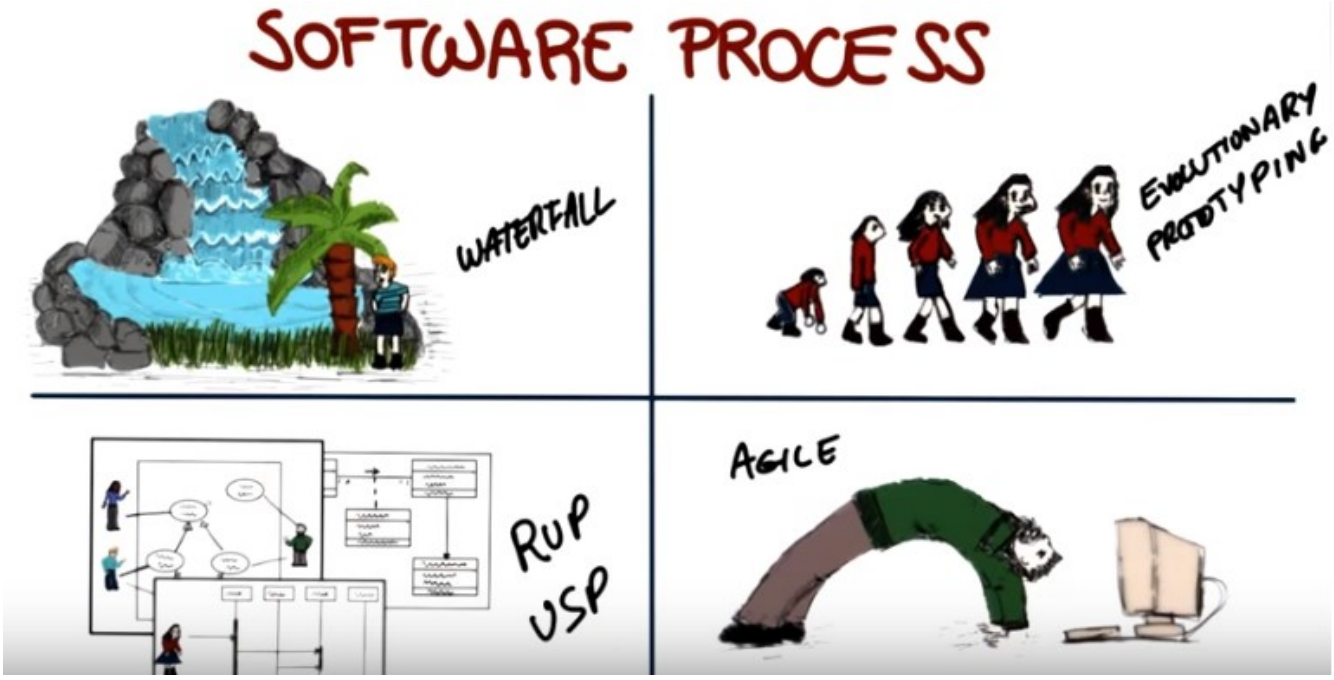
10. After recapping the three major issues that characterize a software crisis let's see what was the evidence that there was indeed a crisis. So what I want to discuss now is the result of a study performed by Davis in 1990s. So in even more recent times than the 60s and the 70s. And the study was performed on nine software projects that were totaling a cost around $7 million and I'm going to show you how this projects went using this representation, this pi representation, in which I'm going to discuss what each of the segment of the pi represent. So lets start looking at the first one. This is a software that was usable as delivered. Other software was delivered, and usable, either after some changes or after some major modifications, so within additional costs involved. But the striking piece of information here is that the vast majority of the software, so these two slices, were software that was either delivered but never successfully used or software that was not even delivered. And this corresponded to five over the seven total million dollars for all the projects. So clearly, this shows a pretty grim picture for software development and its success. In short, there was clear evidence the software was becoming to difficult too build and that the software industry was facing a crisis. And this is what led to the NATO Software Engineering Conference that was held in January 1969, which is what we can consider the birth of software engineering. And what I'm showing here is a drawing of the proceedings for that conference. And if you look at the class notes you can see a link to the actual proceedings, in case you are interested in looking at the issues that were discussed.



11. Now that we saw how software engineering was born and we saw some of the problems that led to

the birth of software engineering. Let's see how we can do better. How can we preform software development in a smarter, in a better way, a more successful way? So what I'm going to show here is the way I see software development. To me software development is fundementally going from an abstract idea in somebody's head, for example, the customer's head, to a concrete system that actually implements that idea and hopefully it does it in the right way. And this is a very complex process. It can be overwhelming. So, unless we are talking about the trivial system, it's very complex for us to keep in mind all the different aspects of the systems, and to do all the different steps required to build this system, automatically. So that's when software processes come to the rescue. So what is a software process? A software process is nothing else but a way of breaking down this otherwise unmanageable task into smaller steps. In smaller steps that we can handle. And that can be tackled individually. So having a software process is of fundamental importance for several reasons. First of all, for non-trivial systems, you can't just do it by getting it, by just sitting down and developing. What you have to do instead is to break down the complexity in a systematic way. So software processes are normally systematic. And you need to break down this complexity, in a more or less formal way. So software processes are also a formal, or semiformal, way of discussing, or describing, how software should be developed. So what are the steps involved in developing software?

12. One thing you need to know right away about software processes is that there's not just one single process, but there are multiple, possible processes, depending on your context, depending on the kind of applications that you are developing. In this course, we are going to try to cover the spectrum of the possible processes, as much as possible, by focusing on four main software processes (在 P1L2 課中, 它們被稱為 software process models 或 software lifecycle model, 注意 P1L2 的題目就是 Lif Cycle Models). The first one is what we call normally the waterfall process. And, we call it waterfall because in the process we go from one phase to the other in the same way in which water follows the flow in a waterfall. The second process that we consider is what we call evolutionary prototyping, and in this case, instead of following this set of rigid steps, all we're trying to do is to start with an initial prototype and evolve it based on the feedback from the customer. We will then move towards a slightly more formal process, which is the rational unified process (RUP) or the unified software process (USP). And this is a kind of project heavily based on the use of UML, so we will also cover UML when discussing this kind of project. Finally, the fourth kind of process we will consider is the family of agile(靈活的) software processes. And these are processes in which we sacrifice the discipline a little bit in order to be more flexible and be more able to account for changes and in particular for changes in requirements.



We are going to cover each one of these four processes extensively in the rest of the class.

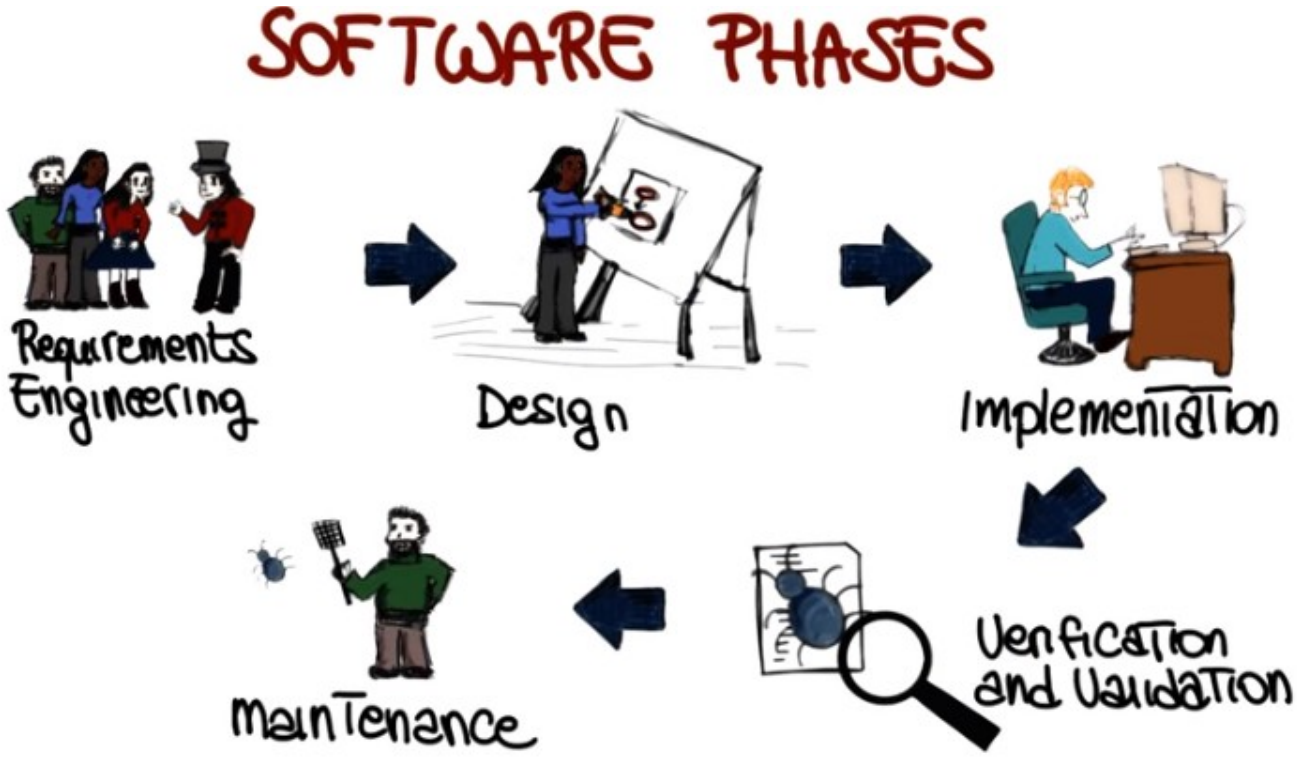
13. So, now before we actually jump to the discussion of software processes I want to ask you a couple of preliminary questions. The first one is, what is the largest software system on which you had worked? And you should enter here the size. And the second question I'm going to ask is how many LOC or how many lines of code per day you were producing when working on this system?

14. We're going to go back to these two questions and to your answers later. But I wanted to gather this information beforehand, so that your answers are not biased, they're not influenced by this subsequent discussion.

15. So now I want to ask you one additional question, which is how many lines of code a day do you think professional software engineers produce? Do you think they produce 25 lines of code? Between 25 and 50? Between 50 and 100? Between 100 and 1000? Or more than 1000 a day? And remember that here we're talking about professional software engineers.

16. Studies has shown that, on average, developers produce between 50 and 100 lines of code a day. And that might not seem much. Why, why only 50 to 100 lines of code in a whole day? And the answer is because coding is not everything. When you develop a system writing code is not the only thing you have to do. It's not the only activity that you have to perform. And that's a very important point.

17. In fact, software processes are normally characterized by several phases, what we call the software phases, and only one of these phases is mainly focused on coding. The other phases are meant to support other parts of software development. The first of these phases is called requirements engineering and that's the phase in which we talk to the customer, to the stakeholders, whoever we are building the software for. And we try to understand what kind of system we need to build. Then we use this information to define our design and the design is the high-level structure, that then can become more and more detailed, of our software system. Once we've defined our design we can actually move to the next phase, which is the implementation, in which we write code that implements the design which we just defined. After implementing the code, we need to verify and validate the code. We need to make sure that the code behaves as intended. And finally, we need to maintain the code. And maintenance involves several activities like, for example, adding new functionality or eliminating bugs from the code or responding to problems that were reported from the field after we released the software. We will look at all of these activities and of the software development process in detail, in the rest of the class. And for each activity, we will look at the fundamental principles and how it is done currently. And in some cases, we will also look at some advance ways to do it. For example, more research approaches for that activity.



18. We will also look at how tools can improve software phases, the software activities, and can support software development tasks in general. And this is something that I will repeat over and over in the class, tools and automation are fundamental, in software engineering. And they're fundamental for improving productivity, not only efficiency but also effectiveness of our activities in the software development process. So let me go back to one of the diagrams that I showed you before. If you remember we had this qualititive diagram in which we were showing that one of the issues that led to the software crisis was the fact that developers' productivity was not able to keep up with the software size and complexity, with the growth in the importance and the complexity of software. What tools can help us to do is to change this and basically move this curve from this original position up here. So that it gets closer and closer to what we need to develop the software that we need to build. So let me discuss examples on how tools can improve productivity. For example, if we are talking about development, think about what kind of improvement it was to go from punch cards to modern IDEs. If we're talking about languages, think about of how much more productive developers became when going from writing machine code to writing code in high-level languages. And finally, if we talk about debugging, which is a very important and expensive activity, moving from the use of print lines to the use of symbolic debuggers dramatically improve the effectiveness and efficiency of development. And these are just some of the tools that we will discuss in the rest of the class and notice that we will also use the tools in practice. So we will use the tools before projects and also during the lessons and for assignments. In particular, we will use three main kinds of tools. The first type is IDE's. And I'm pretty sure you're familiar with IDE's. These are integrated development environments. So, advanced editors in which you can write, compile, run, and debug and even test your code. We'll also use a version control system, systems that allow you to save, and restore, and check the differences between different versions of the code, in particular we will be working with git. We will also be looking at other kinds of tools like coverage and verification tools. These are tools that can help you during testing and I'm a big fan of these tools, so I'm really going to stress the usefulness of these tools and how you should use them in your development.

19. To conclude this introductory lesson, I want to provide you with another view of the four projects that you will perform during the class. And I don't want to say too much about the project, though, because I like to keep a little bit of a surprise element. I just want to make sure that you're aware of what's coming up. The first project is a simple project to get your feet wet. Get familiar with one of the most basic processes, the waterfall process. And the typical software documentation that you will produce during such a process. In Project one you will realize how difficult it is to understand what customers want. In the second project we will explore how prototyping can help requirement gathering. And we'll study different ways of creating prototypes. In the third project you will build a more complex application. And you should realize during the project that the process we are using, which is going to be the rational unified process or unified software process in this case, is actually helping you handle the additional complexity involving this larger development effort. Finally project, this is what normally what I enjoy the most, is the project I consider to be the most fun. Because we get to play with testing and development in child software development, which is sort of a new way of developing compared to the previous projects. Basically what we do is turn things upside down, instead of writing tests before we write the code, and we proceed in quick utility cycles. I hope you'll enjoy this just as much as I enjoy it.

20. Now we're going to conclude this introductory lesson with a very simple quiz, whose purpose is just to see whether you understood an important point about the projects. So the question I want to ask you is how will the project be done? Will they be done individually, in teams, in collaboration with an external company, or none of the above?

21. And the answer is, the project will be done in teams. So you will be collaborating with other students in the class for the different projects. You will collaborate with different students for different projects, and I want to stress once more that although the projects are preformed in teams, the individual assignments will be performed individually.

22. Hi, I'm Sarah Spikes and I'm the Udacity Course developer for the software development process course. You'll see me popping up from time to time usually to introduce an assignment or a project. To help us create well balanced teams for the projects, we'd like you to fill out a survey so that we can gauge your experience level. You will find links to the survey in the instructor notes for this video and in the following quiz.