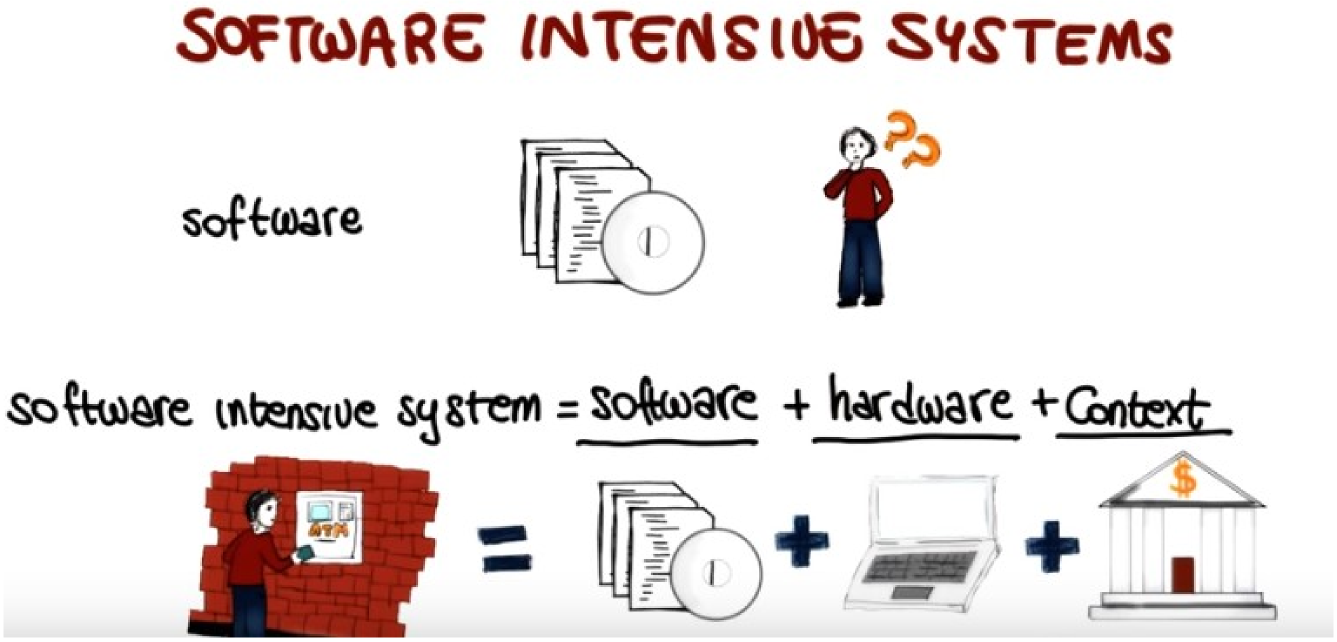
1. Hello and welcome to the second part of our software engineering course. In the previous mini-course, we discussed some basic principles behind software engineering. We provided an overview of several software process models and we introduced some important tools that can help developers increase their productivity. In this mini-course, we will focus on requirement and prototyping. More precisely, we will discuss in depth requirements engineering activities. We will also discuss techniques to perform a system analysis and design in an object-oriented fashion. So, let's start the first lesson of this mini-course, which is about the use of engineering techniques to understand and specify the purpose of a software system.

2. As we did for other lessons, before starting this lesson on requirements engineering, I want to ask a world expert on this topic a few questions. I'm here with Jane Cleland-Huang, a professor at the DePaul University. And Jane is a world expert in the area of requirements engineering, which is the theme of this lesson. So I'm talking to Jane who is currently in Chicago and I want to. Ask her a few questions about requirements engineering. So hi Jane how are you? >> Fine. Thank you Alex. >> And thank you so much for agreeing to be interviewed for our course, I'm sure the students will really benefit from this. And let me start with the first question which is what are software requirements? >> That's an interesting question. And software requirements basically provide us a description of what a system has to do. So, typically they describe the functionality of the features. That the system has to deliver in order to satisfy its stakeholders. And we usually talk about the requirement specification in terms of what the system's going to do. And we describe it sometimes formally in terms of set of shall statements, that the system shall do this or shall do that. Or we can use various templates to specify both textural requirements. But requirements can also be represented informally in, in the form of user stories, or use cases, or more formally in the form of state transition diagrams and even in kind of formal specifications. Especially for critical parts of safety critical systems. >> And another should discuss what the requirements are. What is the requirements engineering? >> So, that's also an interesting question because if you notice it's it's engineering and I'm sure in the other parts of the software engineering process that you're discussing in your course, parts such as testing or coding. They don't have the word engineering there and I think one of the reasons requirements engineering has that term is because it covers a number of different activities. So it includes things such as working with stakeholders to elicit or to proactively discover what their requirements of the system are. Analyzing those requirements so that we understand the tradeoffs. So you might have different stakeholders that care about different things, and it might not be possible to deliver all of those things, so we have to analyze the feasibility of the requirements, explore the tradeoffs, emerge conflicts. And then of course the specification part, which we talked about a little bit already, and the validation, so did we in fact get the requirements right? Did we build a system that actually matches our, our requirements. And then on into the requirements management process. And the requirements management process. Kind of like goes through things like change management. So what if customer or stakeholders need the system to change? How do we manage changing requirements? And I think this is one of the reasons that we've coined the term engineering because that it's, has to be a systematic process which extends across. The whole of this is life cycle. >> And I guess my last question here is so now that we heard about software requirements and about software requirements engineering, why is requirements engineering so important? So what happens if we don't do it right? >> Well, I'm sure that, you know, many people have probably read the kind of report like Spanish report, and other reports of failed project, and things like that. And are aware. Now one of the major reasons for projects failing is because we didn't get the requirements right in the first place. So if we don't understand the requirements then we're simply going to build the wrong system. Getting requirements right includes all sorts of things such as finding the right group of stakeholders so we don't exclude major groups of stakeholders. Understanding the requirements correctly. There will be many, many different examples of projects that have failed. For example, in America the healthcare.gov failure, and while we cannot

put the blame squarely in the area of requirements, because obviously the project was challenged for a number of different reasons. But clearly it underperformed in many respects related to security, performance, and reliability and these are all parts of the requirements process. Things that should have been discovered and the system should have been built in order to meet those requirements, getting the requirements right in the first place. Puts us, a project on the right foot. And so that gives us a much better chance of delivering to the customer what they need. And designing a solution that really meets those requirements. So, it's a critical part of the overall software engineering success. >> Okay. So that's critical. I mean, we better get our requirements right. >> Yeah. >> That's, that's the message. >> Yeah. >> Okay. Well, thank you so much Jane, for taking the time off your busy schedule to speak with us. I'm sure. The students really appreciate this, and we'll talk to you soon. >> Bye Alex, thank you. >> Bye, Jane, bye bye. Jane give us an interesting perspective on requirements engineering and its importance. Let's now start our lesson with a general definition of requirements engineering.

3. Basically, and roughly speaking, requirements engineering, which is also called in short, RE, is the process of establishing the services that the customer requires from the software system. In addition to that, requirements engineering also has to do with the constraints under which the system operates and is developed. Requirements engineering is a very important activity for several reasons. In particular, as we also saw in earlier lessons, many errors are made in requirement specifications. So many errors are made because we don't do requirements engineering in the right way. And many of these errors are not being detected early. But they could be if we were to do RE in the right way. And, unfortunately, not detecting these errors can dramatically increase software costs. So that's the reason why requirements engineering is important, and why it is important to do it in the right way. The final result of the requirements engineering process is a software requirements specification that we also called SRS. We will discuss SRS later in more details and also when we talk about the projects for the course. For now, it is enough to say that the software requirements specification and the requirements engineering, in general, should focus on what the proposed system is intended to do, and not on the how it will do it. In fact, how the system will do what it is required to do is something that we will discuss when we talk about design of a system in later phases.

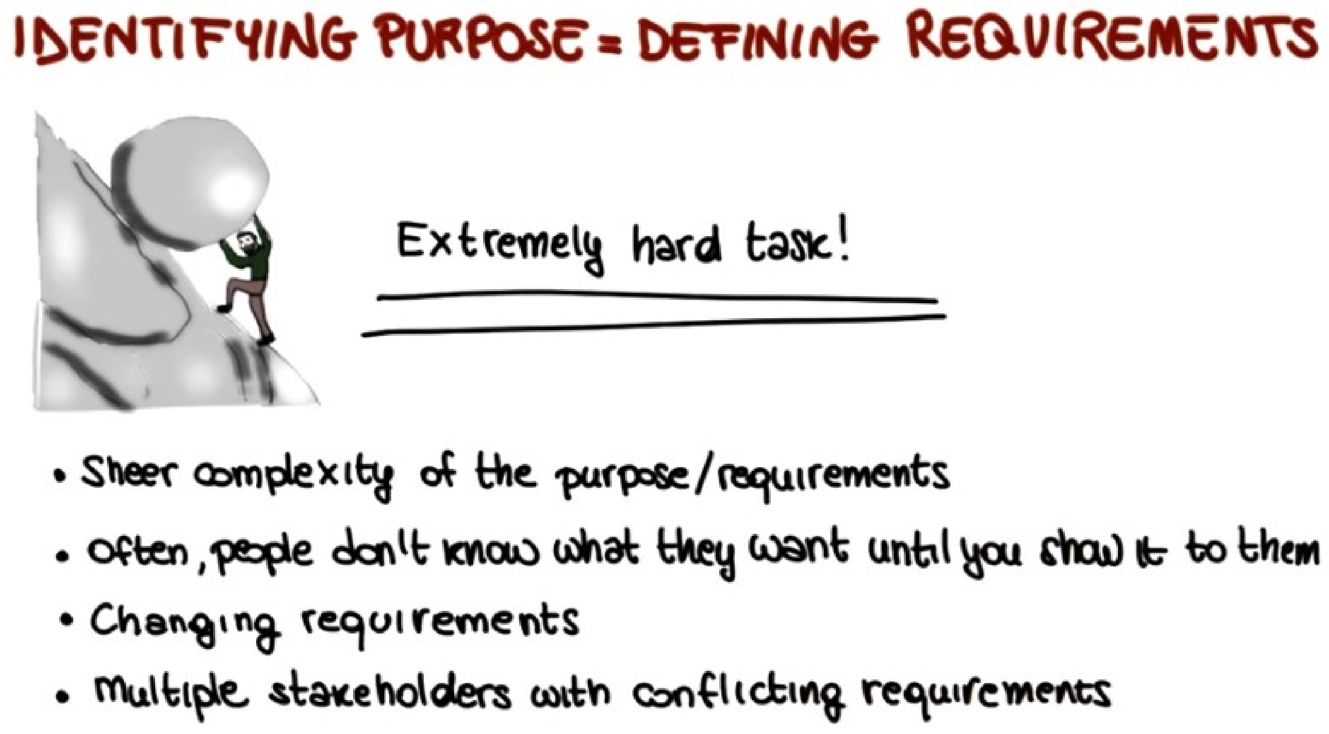
4. In our initial definition of requirements engineering, we talked about software systems. But what do we really mean when we use the term software? Software is an abstract description of a set of computations that becomes concrete, and therefore useful, only when we run the software on some hardware, and that, in the context of some human activity that it can support. So what does that mean exactly? What that means is that when we say software, what we really mean is a software intensive system. That is, the combination of 3 things, the software, the hardware on which the software runs, and the context in which the software is used. Just to illustrate, let me show you this through a picture. What I'm showing here is a customer, a user, that is using, is accessing, an ATM machine. And this action involves several things. There is the software that drives the logic of the ATM machine. There is the hardware on which the software runs. And there is the context In which the software is used. And in this case, the context is the bank. And only by considering these 3 things together can we really understand the functionality that is represented here. So the bottom line here is that we usually take hardware and context for granted in this equation. But they actually need to be explicitly considered when building a system. Otherwise, we might forget this is all the functionality, and ultimately of the requirements. And we might end up with the wrong system.


SOFTWARE INTENSIVE SYSTEMS
software
software intensive system = software + hardware + Context
ATM
$

5. So, let's see how this affects the concept of software quality. Another way to express what we just said is to say that the software runs on some hardware and is developed for a purpose that is related to human activities. And given this perspective, we can define what we mean by software quality in this light. Software quality is not just a function of the software. So, the software itself does not define the quality of the overall system. Rather, software quality is a function of both the software and its purpose. Where purpose has to do with the way in which the software will be used. So a software system can be of low quality not only because it does not work well. So, for example, not only because it crashes. Of course, that's an issue. But just as importantly, a software can also be of low quality because it does not fulfill its purpose, and this happens quite often. It is unfortunately not rare for the software producers to have an inadequate understanding, or even a complete misunderstanding of the purpose of the software, of what the users want to do and will do with it. Turning these around, we can therefore define the quality of software in terms of fitness for purpose. The more the software fulfills its purpose, the more the software is on target, the higher is its quality. And identifying the purpose of the software, so hitting this target, is exactly the goal of requirements engineering. And it is the reason why requirements engineering is such a fundamental activity in the context of software engineering.

SOFTWARE QUALITY
software
runs on some
hardware
and is developed for a
purpose
that is related to human activities
Quality = f(,)
FITNESS FOR PURPOSE
Requirements engineering is mostly about identifying the purpose

6. And identifying the purpose of a softer system means defining the requirements for the system. And if you have ever done anything like that, for example, we did it for the first project in the previous mini course, you will know that it is an extremely hard task. Identifying the purpose of the software and defining its requirements is very, very hard. Why is it so hard? First of all, the purpose of most systems is inherently, extremely complex, so this has to do with the sheer complexity of the purpose of the requirements. Just think of how complex is the functionality provided by most systems. Second, it is hard, very hard to extract from humans this purpose and make it explicit. So, paraphrasing a famous quote from the late Steve Jobs, often people don't know what they want until you show it to them. It's hard to figure out what people really want. Third, requirements often change over time. Customers change their mind. Designing and building a system raises new requirements. So for many reasons requirements tend not to be stable, tend to evolve. And that, of course, makes it harder to collect them. Finally, for any realistic system, there are many stakeholders and they often have conflicting goals and requirements. And it can be very hard to reconcile the possibly conflicting requirements that might emerge in these cases. So for all these reasons, it is very, very difficult to perform requirements engineering in an effective way.

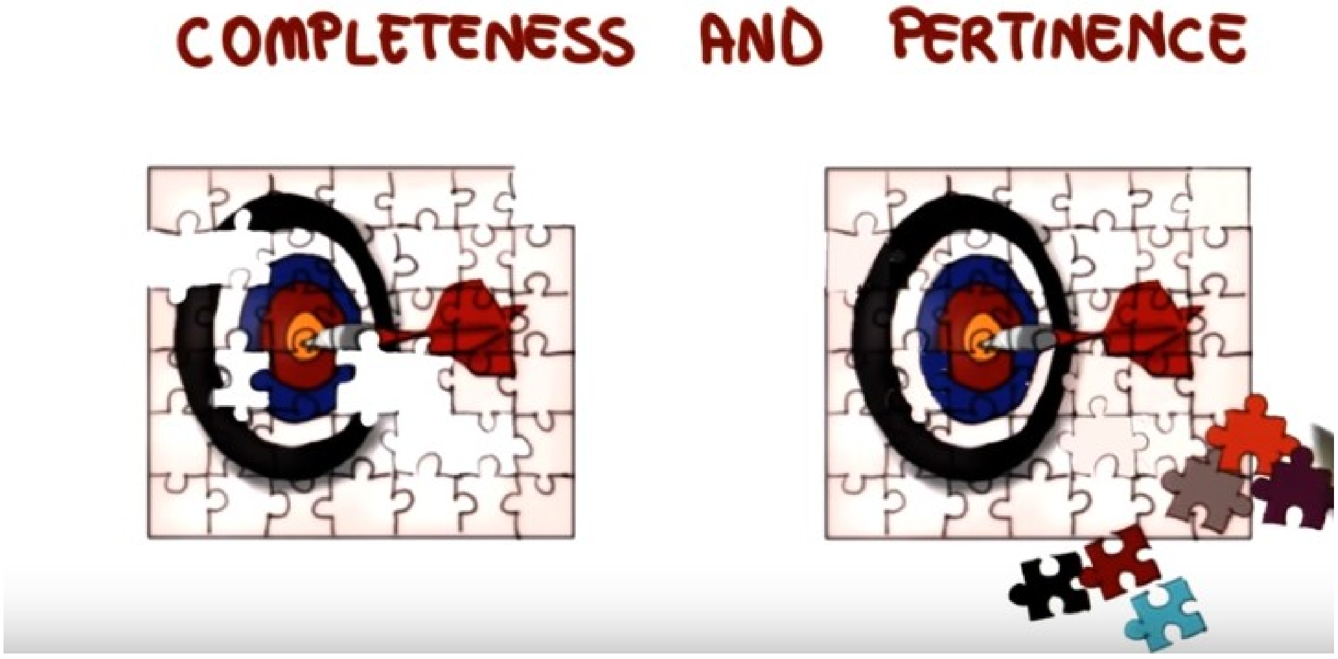


IDENTIFYING PURPOSE = DEFINING REQUIREMENTS

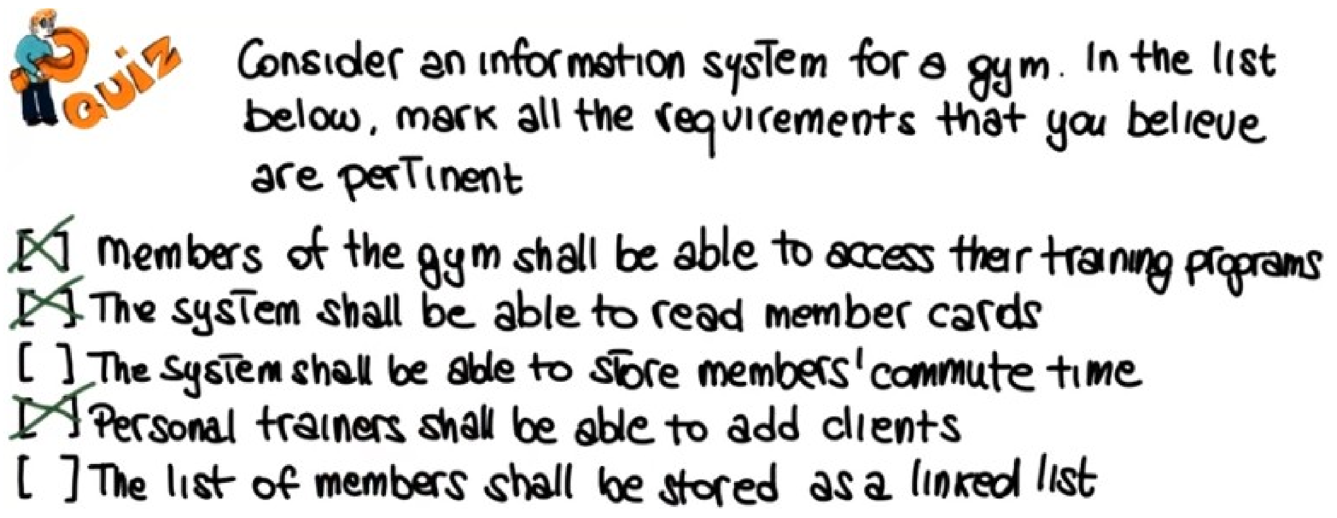
Extremely hard task!

- Sheer complexity of the purpose/requirements
- often, people don't know what they want until you show it to them
- Changing requirements
- Multiple stakeholders with conflicting requirements

7. These issues and difficulties can result in requirements that show various problems. Two particularly relevant and common problems are completeness and pertinence(妥當,適當). Or better, the lack of completeness and pertinence. Completeness refers to the fact that it is often extremely difficult to identify all of the requirements. That is it is very difficult to have a complete picture of the purpose of the software. So what happens is that incomplete requirements are collected and the software is missing functionality that is important for the user. Pertinence conversely has to do with the relevance of the requirements. To avoid completeness problems developers often end up collecting a lot of irrelevant when not conflicting requirements. In this cases what can happen is that the software could either end up being bloated that is it might contain a needed functionality. The functionality represented by these extra requirements or it might even be impossible to build the software due to the conflicting additional requirements. And to make things even worse collecting all of these requirements sometimes doesn't even solve the completeness issue. So you might end up with a set of requirements that is not only incomplete but it also contains extra information that can be harmful to the system. So again the bottom line is that gathering an adequate, accurate, complete, and pertinent set of requirements that identify the purpose of a software system is an arduous task.



8. Now that we talked about completeness and pertinence, let's consider an information system for a gym. I'm going to give you a list of possible requirements and I want you to mark in that list all the requirements that you believe are pertinent. So let me read the list. Members of the gym shall be able to access their training programs. The system shall be able to read member cards. The system shall be able to store members' commute time. Personal trainers shall be able to add clients. And the list of members shall be stored as a linked list.



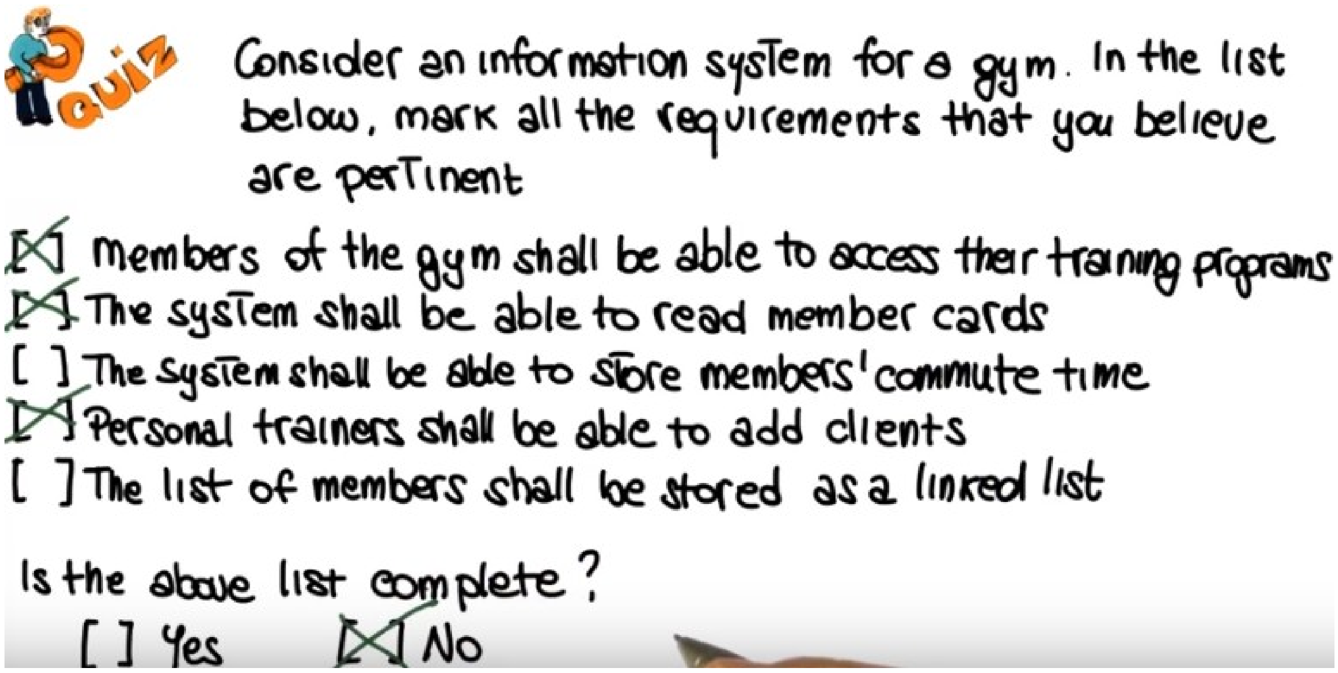
Consider an information system for a gym. In the list below, mark all the requirements that you believe are pertinent

[✓] Members of the gym shall be able to access their training programs
[✓] The system shall be able to read member cards
[ ] The system shall be able to store members' commute time
[✓] Personal trainers shall be able to add clients
[ ] The list of members shall be stored as a linked list

9. So the first requirement is definitely pertinent. Members of the gym shall be able to access their training programs. It's pretty normal for members of the gym to have a training program. And therefore, the system should allow them to access them. Similarly for the second one. The system shall be able to read member cards. Normally when you get into a gym if you have a member card, you'll have to either show it to somebody, or nowadays swipe it, and so the system should be able to recognize the customer given the card. The third requirement is probably not pertinent, because I cannot think of any

meaningful case in which the system should know what is the members' commute time. The fourth requirement, personal trainers shall be able to add clients, is also probably pertinent. Assuming that we have personal trainers in the gym, and they should be able to get clients, to work with the clients of the gym, and therefore, they should be able to add them as their clients to the system. And finally, the last requirement, the list of members shall be stores as a linked list. This is really something about the how, more than the what. And therefore, for what we say before is probably not a pertinent requirement, so we're not going to mark this one.

10. So now that we saw which of these requirements are pertinent and which ones are not, can we consider the above list of requirements of the list of these requirements marked here as a complete list of requirements for a gym? And you have two options, yes or no.



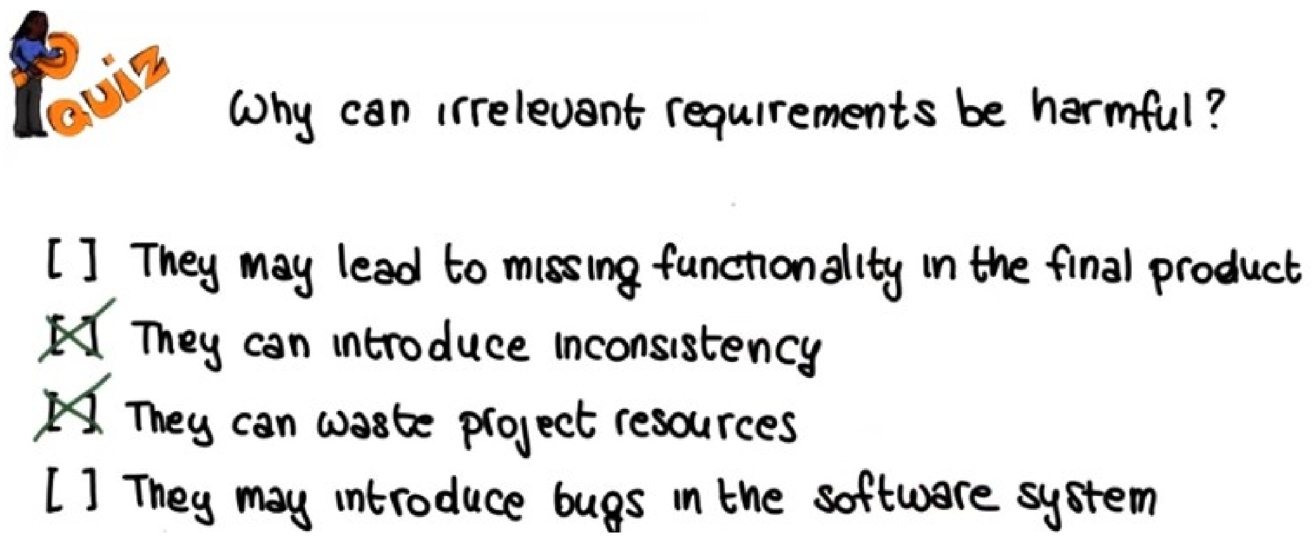
Consider an information system for a gym. In the list below, mark all the requirements that you believe are pertinent

[✓] members of the gym shall be able to access their training programs
[✓] The system shall be able to read member cards
[ ] The system shall be able to store members' commute time
[✓] Personal trainers shall be able to add clients
[ ] The list of members shall be stored as a linked list

Is the above list complete?
[ ] Yes [✓] No

11. And the answer is clearly no. Obviously, there are many missing requirements here. For example, requirements about registration for the customers, requirements about fitness program creation, membership types, and so on. Plus, we are also missing all of the nonfunctional requirements, which we haven't seen yet, but that we will discuss in a bit.

12. In the previous quiz, we saw that some of the requirements that they put in the list were not pertinent. They were irrelevant. So let me ask you. Why can irrelevant requirements be harmful? Why is that a problem to have irrelevant requirements? So here, I'm giving you four possible answers. And I'd like for you to mark all that apply. Can irrelevant requirements be harmful because they may lead to missing functionality in the final product. Because they can introduce inconsistency. Because they can waste the project resources. Or because they may introduce bugs in the software system. And as I said, more than one can be a valid reason.



Why can irrelevant requirements be harmful?

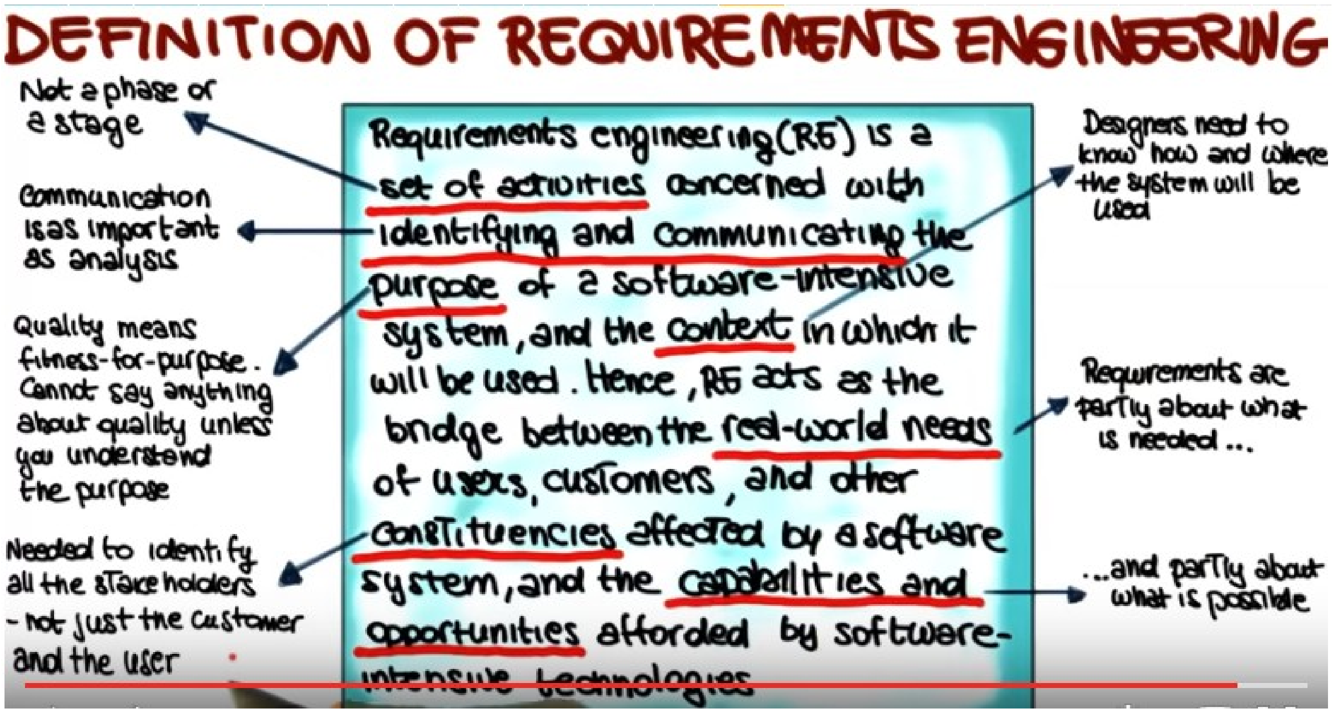
[ ] They may lead to missing functionality in the final product
[✓] They can introduce inconsistency
[✓] They can waste project resources
[ ] They may introduce bugs in the software system

13. So let's go through the list. Definitely irrelevant requirements cannot lead to missing functionality in the final product, because irrelevant requirements actually refer to unneeded functionality in the system. So functionality that is put in the requirements, but it is not really needed. So we're not going to mark this one. Indeed, irrelevant requirements can introduce inconsistencies. So they could be irrelevant requirements that not only are not pertinent but they are inconsistent with some of the pertinent requirements. They can also waste project resources, because if we spend time designing and then implementing the parts of the system that we referred to this irrelevant requirements, of course, we are wasting project resources. And I will not mark the last one because there's really no correlation between any irrelevant requirements and bugs in the software system. Of course, by implementing the part of the system that refers to an irrelevant requirement you might introduce a bug. But that's not necessarily the case, and there really no correlation between the two.

14. But we collect requirements all the time, right? Every time we build a software system. So how do people cope with these difficulties? Well there are the best practices. In practice, developers or analysts usually identify a whole bunch of requirements. Sometimes the easiest and most obvious ones. They bring those to the stakeholders, and the stakeholders have to read the requirements, understand them, and if they agree, sign off on them. And the problem is that in general, these requirements documents are difficult to read. They are long, they are often unstructured. They typically contain a lot of information. And in general, they are not exactly a pleasant read. So what happens is that often the stakeholders are short on time, overwhelmed by the amount of information they're given and so they give in to the pressure and sign. And this is a bit of a dramatization clearly but it's clear that what we are looking at is not an ideal scenario. Clearly this is not the way to identify the real purpose of a software system to collect good requirements. And since one of the major causes for project failure is the inadequacy of requirements, we should really avoid this kind of scenario. We should follow a rigorous and effective requirements engineering process instead.

15. But how can we do that? How can we identify the purpose of the system and collect good requirements? To answer that question, let me give you another definition of requirements engineering. And this one is a classical one, one that summarizes what we discussed so far, and then we can use as some sort of reference. And it is a little long. Definitely longer than the one that we saw at the beginning. But it's an important one and it contains a lot of very relevant points. So, we're going to go through it and highlight these points. So the definition says, that the requirements engineering is a set

of activities concerned with identifying and communicating the purpose of a software intensive system and the context in which it will be used. And this is exactly what we said at the beginning. But something we can highlight in here, is the fact that we're talking about a set of activities. So, what that means is that requirements engineering is not just a phase or a stage. It also says that it's about identifying and communicating. And what that is telling us is that communication is as important as the analysis. So, it's important to be able to communicate these requirements not only to collect them. And we will discuss many reasons why that is the case. It explicitly talks about purpose. So that allows me to stress, once more, that quality means fitness-for-purpose. We cannot say anything about quality unless we understand the purpose. And the last thing I want to point out in this first part of the definition is the use of the term context. This is also something else that we mentioned at the beginning, that designers, analysts, need to know how and where the system will be used. Without this information, you cannot really understand what the system should do and you cannot really build the system. So now, let's continue and read the second part of the definition that says, hence. Requirements engineering acts as the bridge between the real-world needs of users, customers, and other constituencies affected by a software system and the capabilities and opportunities afforded by software-intensive technologies. This is a long sentence, but also here, we can point out a few interesting and relevant points. Let me start by highlighting two parts. Real-world needs, and the capabilities, and opportunities. So, what are these two parts telling us? They are telling us that requirements are partly about what is needed, the real-world needs of all these stakeholders. But they're also partly about what is possible, what we can actually build. We need to compromise between these two things. And, finally, I would like to point out this term constituencies, which indicates that we need to identify all of the stakeholders, not just the customer and the users, so anybody who is affected by a software system. It is very important to consider all of these actors. Otherwise, again, we'll be missing requirements, we'll be missing part of the purpose of the system and we will build a suboptimal system.



DEFINITION OF REQUIREMENTS ENGINEERING

Requirements engineering (RE) is a set of activities concerned with identifying and communicating the purpose of a software-intensive system, and the context in which it will be used. Hence, RE acts as the bridge between the real-world needs of users, customers, and other constituencies affected by a software system, and the capabilities and opportunities afforded by software-intensive technologies

Not a phase or a stage

Communication is as important as analysis

Quality means fitness-for-purpose. Cannot say anything about quality unless you understand the purpose

Needed to identify all the stakeholders - not just the customer and the user

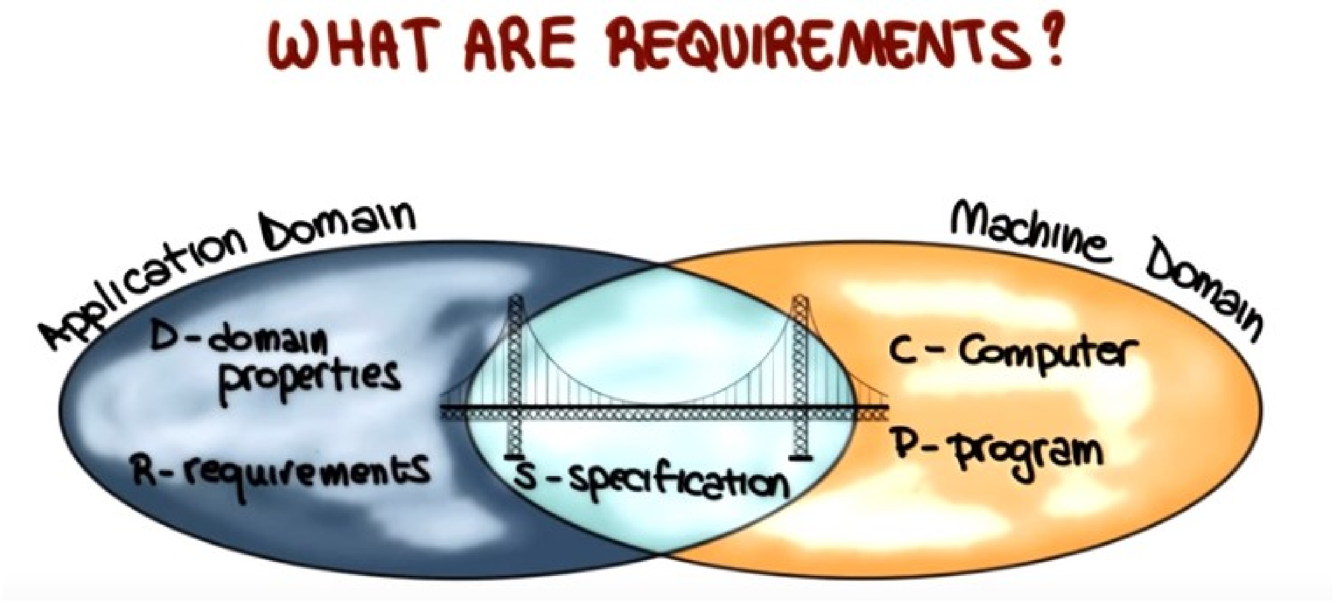
Designers need to know how and where the system will be used

Requirements are partly about what is needed ...

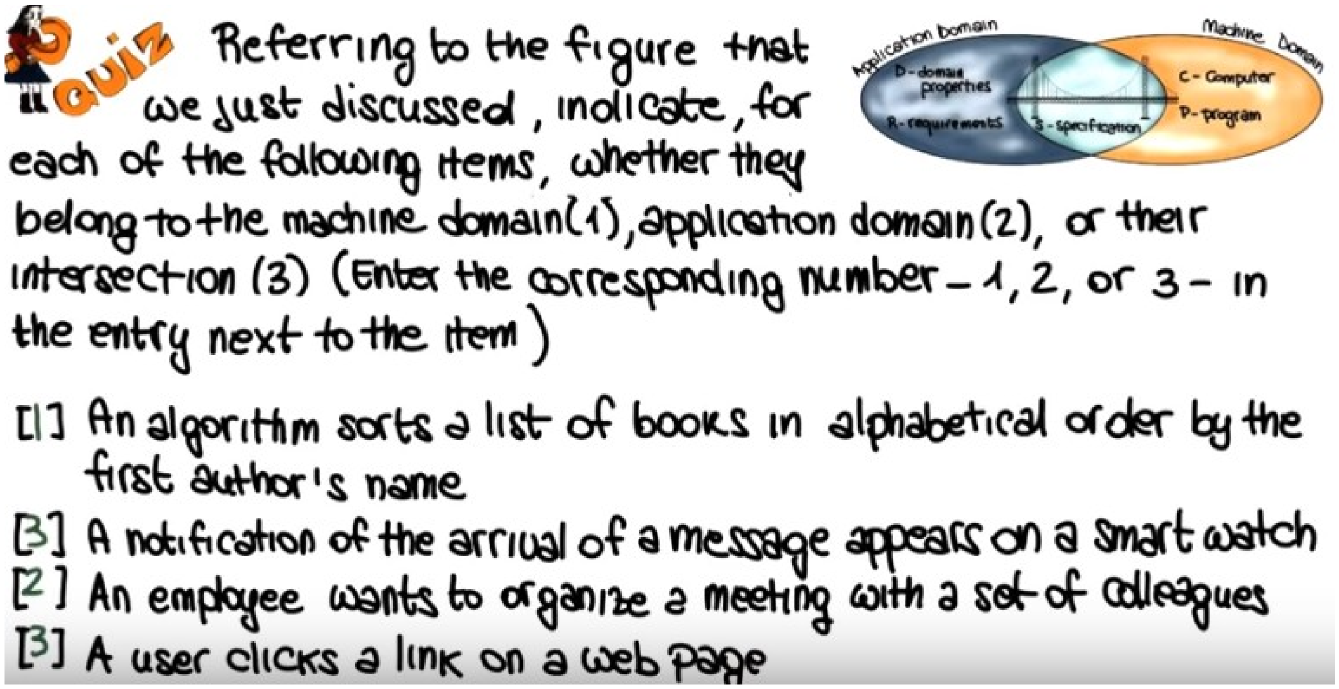
...and partly about what is possible

16. So at this point, we have talked quite a bit about requirements engineering, but we haven't really discussed what are requirements exactly. So what is a requirement? To define that I am going to use this diagram which is a classical one. So you might have seen it before. So, discussing this diagram

allows me to point out a few interesting things about requirements and define them in a better way. At a high level this diagram contains two main parts, the domain of the machine, which is the hardware, operating system, libraries and so on, on which the software will run. And the domain of the application, which is a world in which the software will operate. And the machine domain is characterized by computers, which are the hardware devices, and programs, which is the software that runs on these devices. The application domain, conversely, is characterized by domain properties, which are things that are true of the world anyways, whether I'm building my system or not, and requirements, which are things in the world we would like to achieve by delivering the system that we are building. Basically, to put it in a different way, the former, the domain properties, represents the assumptions that we make on the domain. And the latter, the requirements, are the actual requirements that we aim to collect. So we have something here, right, at the intersection of this application domain and this machine domain. And what is that? And this is what we normally call the specification, which is a description, often a formal description, of what the system that we are building should do to meet the requirements. So this is a bridge between these two domains. And as the graphical depiction shows, the specification is written in terms of shared phenomena. Things that are observable in both the machine domain and the application domain. And just to make things a little more concrete, I want to give you a couple of examples of what these phenomena, these shared phenomena, are. And we can think about two main kinds of phenomena. The first one are events in the real world that the machine can directly sense. For example, a button being pushed or a sensor being activated. These are events that happen here, but that the machine can detect. So they're events that can be used to define the specification. And the second type of phenomena are actions in the real world that the machine can directly cause. For example, an image appearing on a screen or a device being turned on and off. Again, this is something that the machine can make happen and then can have manifestation in the real world. And again this is therefore something on which the specification can predicate, something that we can describe in our specification. And this is sort of a philosophical discussion, but even if you don't care about the philosophical discussion, the one take away point that I would like for you to get from this discussion is the fact that when writing a specification you have to be aware of the fact that you're talking about shared phenomena. Events in the real world that the machine can sense and actions in the real world that the machine can cause. So this is what the specification is about, a bridge between these two worlds that define what the system should do to satisfy the requirements.



17. Since we just discussed application domain, machine domain, and the specificiation, let's make sure that these concepts are well understood. To do that, I'm going to use a quiz, and I would like for you to refer to the figure that we just discussed that I'm also reproducing here on a small scale on the right. And then referring to the figure, you should indicate for each of the items that I'm going to show you here shortly. Whether they belong to the machine domain. In this case, we're going to put a one next to the icon. The application domain, in this case you should put two. Or their intersection, and in this case you should put three. So this is the lists of items. So let me read it. An algorithm sorts a list of books in alphabetical order by the first author's name. A notification of the arrival of a message appears on a smart watch. An employee wants to organize a meeting with a set of colleagues. And finally, a user clicks a link on a web page. So again, put 1, 2, or 3 here in these lots, depending on whether you think that these items belong to the machine domain, the application domain, or their intersection. So, their specification, here.



Referring to the figure that we just discussed, indicate, for each of the following items, whether they belong to the machine domain(1), application domain(2), or their intersection (3) (Enter the corresponding number - 1, 2, or 3 - in the entry next to the item)

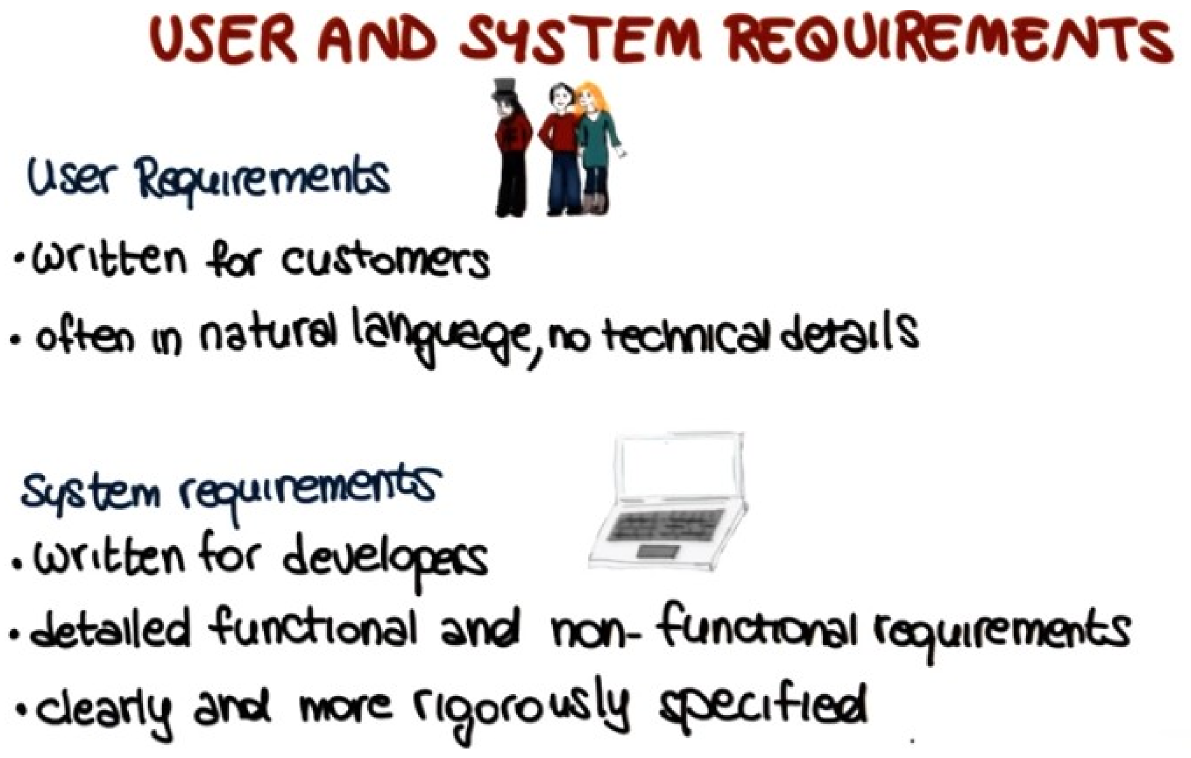
[1] An algorithm sorts a list of books in alphabetical order by the first author's name
[3] A notification of the arrival of a message appears on a smart watch
[2] An employee wants to organize a meeting with a set of colleagues
[3] A user clicks a link on a web page

18. So let's look at each one of these items individually, starting from the first one. And here this item has to do with how the machine stores the information and how the corresponding algorithm is written. But it has no bearing in the real world. That is, in the application domain. Therefore this. Definitely belongs to the machine domain, and we're going to put a one here. What about a notification of the arrival of a message on a smart watch? This is an event that is generated within the machine, but it has an effect, an observable effect, in this case, in the real world as well. Therefore, we're going to mark this as three. So this is an event. This is something that belongs to the intersection between the application domain and the machine domain. So it's something that could be in the specification. Now what about an employee that wants to organize a meeting with a set of colleagues? This is an event that belongs to the application domain because it is a fact that it's true that exists. In the real world independently from the existence of a machine. Therefore, we're going to mark this as two. Finally, the event of a user clicking on a link on a web page is an event that occurs in the real world but that has an effect also within the machine and, therefore, we're going to mark this as three, something that happens at the intersection. between these two domains, and once more, something that could be in a

specific issue.

19. Among the requirement that we can collect from the application domain, we need to distinguish between two main types. And you've probably heard about these ones. Functional requirments and non-functional requiremnts. Functional requiremetns have to do with the functionality of the system, with what the system does with the computation. For example the elevator shall take people to the floor they select. That's a functional requirement, that has to do with the functionality of the system. Or for a very simple one, the system has to output the square root of the number past as an input. So these kind of requirements have in general well defined satisfaction criteria. So, for example, if for the latter one that we mentioned it is pretty clear how to check whether the output is actually the square root of the number passed in input. Non-functional requirements, conversely, refer to a system's non-functional properties, systems qualities. Such as security, accuracy, performance, cost. Or, you know, usability, adaptability, interoperability, reusability and so on. So, all these qualities the don't necessarily have to do with the functionality. And, unlike functional requirements, non functional requirements Do not always have clear satisfaction criteria. For example, if we say that the elevator must be fast, that's a non-functional requrement. Right? It has to do with the speed of the elevator, which is a quality of the elevator. But, it, it's not clear how such a requirement could be satisfied. How could we tell whether the elevator is fast or not. So, what we need to do in these cases Is that we need to refine these requirements so that they become verifiable. For the example that I just mentioned, for instance, we might say that the elevator must reach the requested floor in less than 30 seconds from the moment when the floor button is pushed. This is still a non-functional requirment, but is a verifiable one.

20. Another important distinction, when talking about requirements, is that between user and system requirements. So, let's start with defining user requirements. Those are requirements that are written for the customers and they're often in natural language and they don't contain technical details. And the reason for that is that their purpose is to allow customers, stakeholders, to check that the system will do what they intended. So it's a way for the analyst, the developers, to communicate with the customers, with the stakeholders. System requirements, on the other hand, are written for developers. Contain detailed functional and non functional requirements. Which we just discussed, and which are clearly and more rigourously specified than the user requirements. And the reason for this difference is that the purpose of the system requirements is to tell developers what to build. They must contain enough details so the developers can take them and use them to design and then develop a system. Just to give you a concrete example, here I'm showing you a user requirement that just says that the software must provide a means of representing and accessing external files created by other tools, and the corresponding system requirement. And as you can see, even if we don't read the whole requirements. The former is an informal and high level description of a piece of functionality, whereas the latter describes the same functionality but in a much more extensive and rigorous way. As I said, this is something that the developers can use to design and then build a system whereas this is something that can be used to communicate. With the stakeholders, with a non-technical audience. And we need to define both because they serve different purposes.



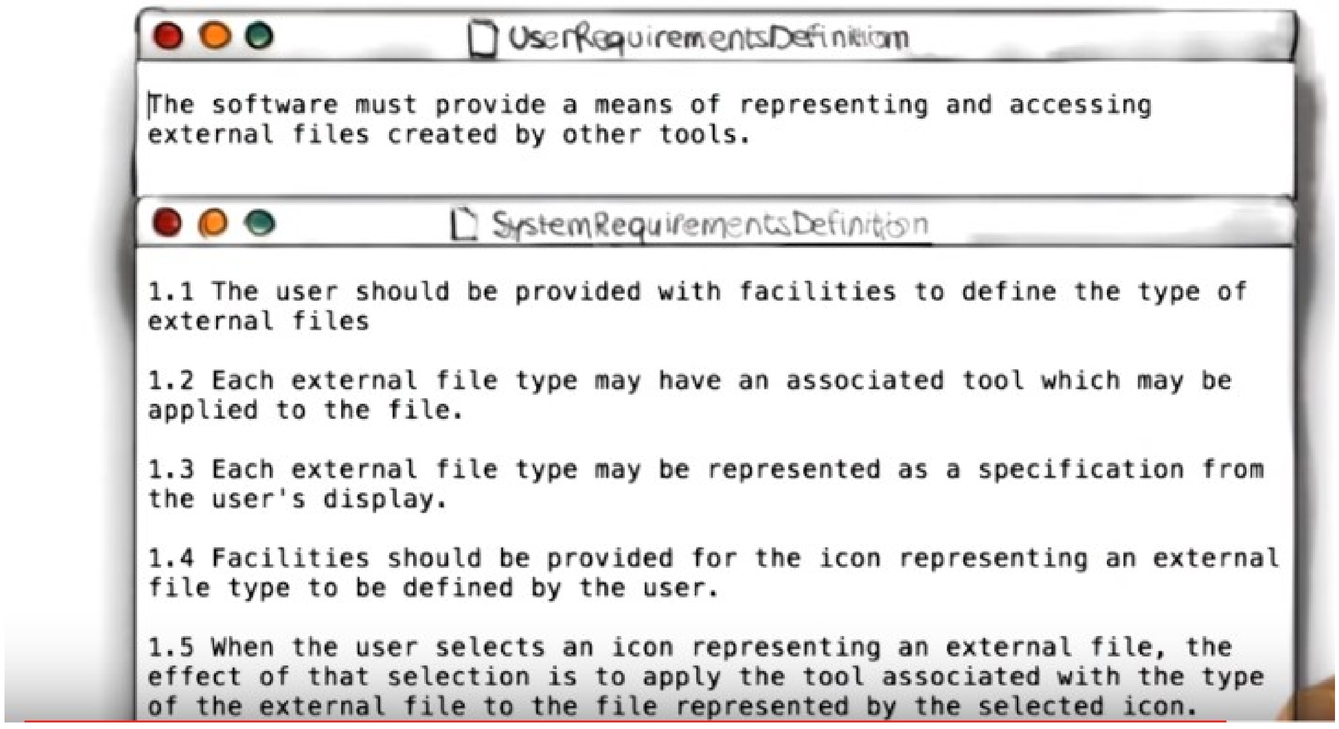
USER AND SYSTEM REQUIREMENTS

User Requirements

- written for customers
- often in natural language, no technical details

System requirements

- written for developers
- detailed functional and non-functional requirements
- clearly and more rigorously specified



UserRequirementsDefinition

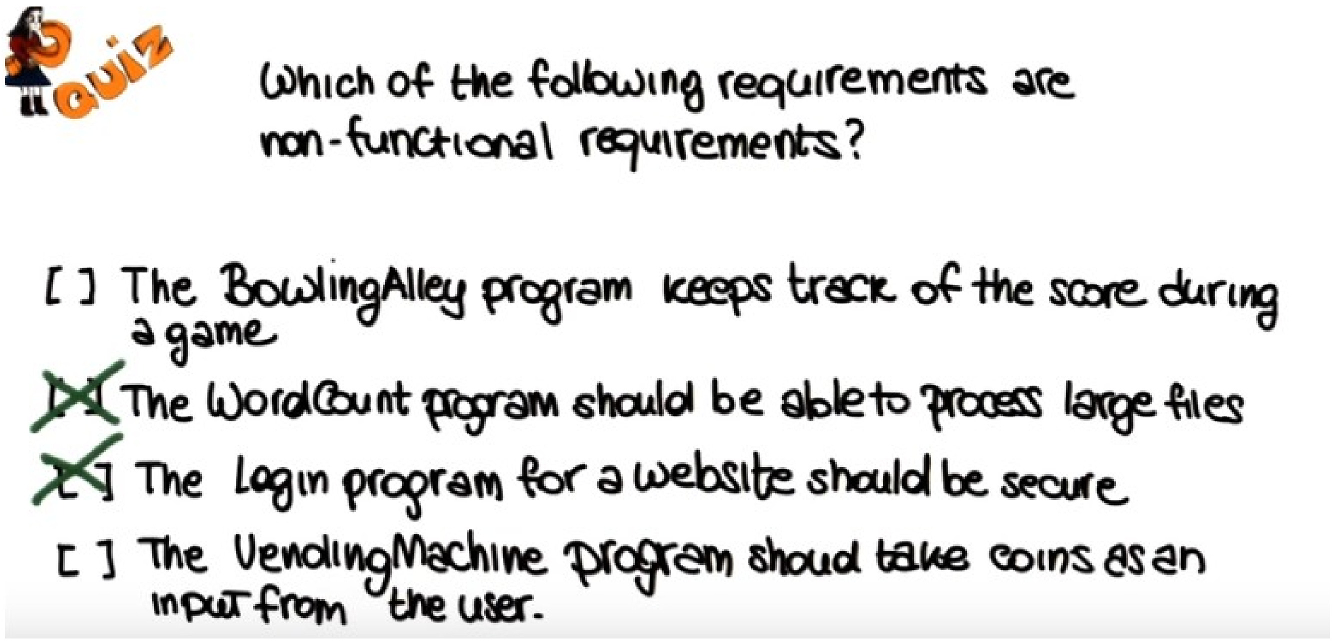
```
The software must provide a means of representing and accessing
external files created by other tools.
```

SystemRequirementsDefinition

```
1.1 The user should be provided with facilities to define the type of
external files

1.2 Each external file type may have an associated tool which may be
applied to the file.

1.3 Each external file type may be represented as a specification from
the user's display.

1.4 Facilities should be provided for the icon representing an external
file type to be defined by the user.

1.5 When the user selects an icon representing an external file, the
effect of that selection is to apply the tool associated with the type
of the external file to the file represented by the selected icon.
```

21. After all these talking about requirements, let's have a small quiz, I want to ask you which of the following requirements are non-functional requirements? And here I'm listing the requirements, the first one says at the BowlingAlley program keeps track of the score during a game, the second one is that the WordCount program should be able to process large files. The third one is that the Login program for a website. Should be secure, and finally the last one says that the vending machine

program should take coins as an input from the user. So, I want you to mark all the ones that are non-functional requirements, that don't refer to the functionality of the system.



Which of the following requirements are non-functional requirements?

[ ] The BowlingAlley program keeps track of the score during a game

[✓] The WordCount program should be able to process large files

[✓] The Login program for a website should be secure

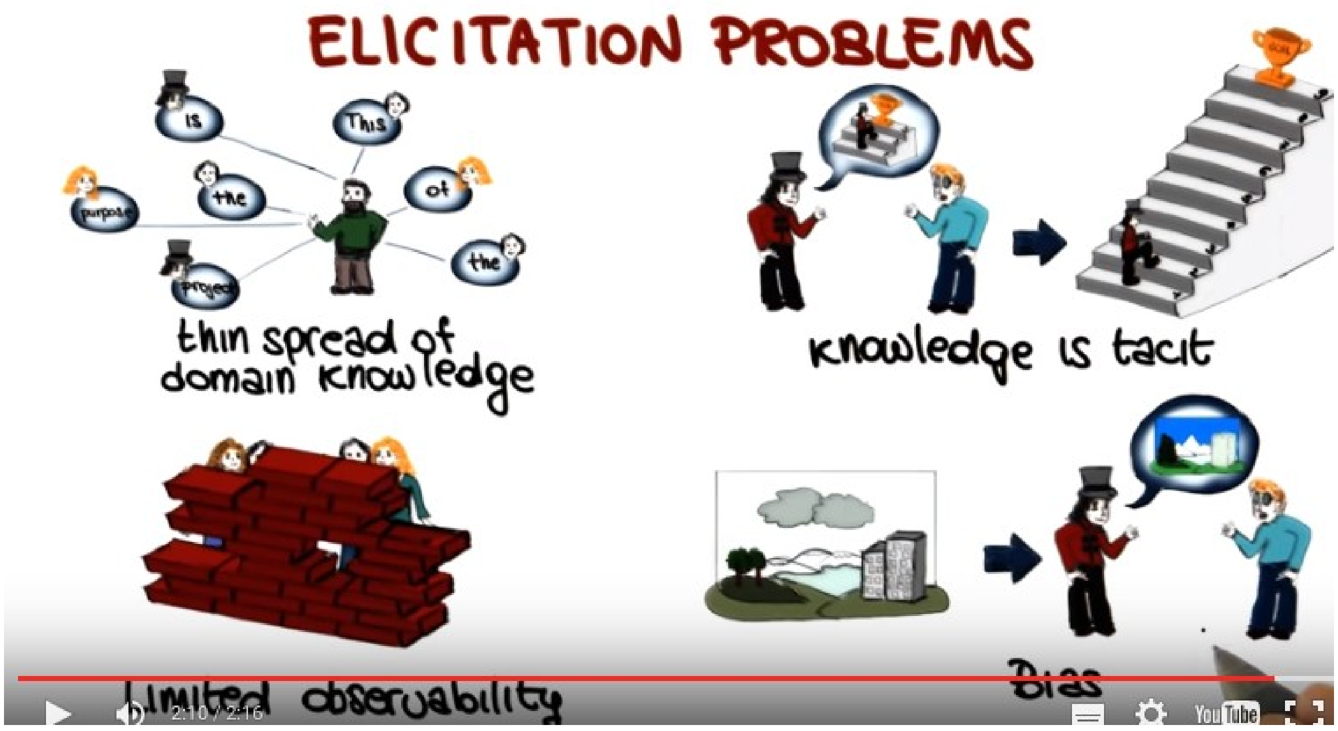
[ ] The VendingMachine program shoud take coins as an input from the user.

22. So, the first requirement clearly refers to some specific functionality of the Bowling Alley system, because it talks about what the system has to do from a functional standpoint. So, it's definitely not a non-functional requirement. On the other hand, the fact that the Word Count system should be able to process large files, is telling us something not about the functionality of the system, but rather about its qualities. The fact that it has to be scalable, that it has to be efficient and so we can consider this to be a non-functional requirement. Similarly, the fact that the Login program for a website should be secure is definitely telling us something about the quality of the system that has little to do with its functionality. And so this is also a non-functional requirement. Finally, the fact that the Vending Machine program should take coins as an input from the user is telling us something about the functionality of the program and therefore, is a functional requirement.

23. Now that we know what the requirements are and their main types, let's discuss where requirements come from and there are many possible sources for requirements so I'm going to list here the main ones. The first one are clearly stakeholders, anybody who is effected by the system and its functionality. Customers, users, and so on. The second typical social requirement is the application domain. For example, the fact that my software is running within a bank, or within a school. Why is the application domain a social requirement? Well, because there are constraints that are characteristics of the application domain that will affect the functionality of the system. For a simple example, just think about regulations. So banking regulations and school regulations in these cases. Those are things that might affect the functionality of my system and, therefore, that may become part of my requirements. And, finally, documentation can be an additional source of requirements. For example, notes, papers, manuals, books. So everything that refers to the functionality of the system that we're going to build.

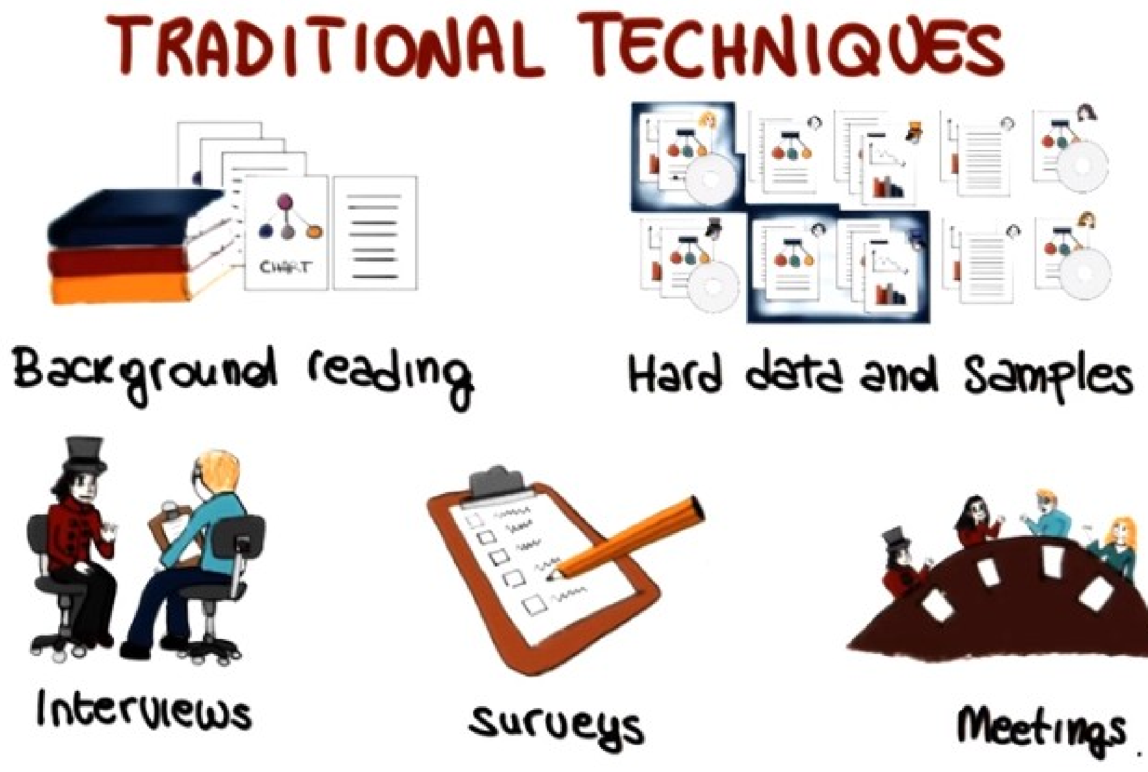
WHERE DO REQUIREMENTS COME FROM?
Stakeholders
...
Application domain
SCHOOL
...
Documentation
CHART
my personal notes
...

24. Unfortunately, extracting requirements from these sources is not a straightforward task, as there are many issues involved with the requirements elicitation(誘出,抽出). One first problem is the thin spread of domain knowledge. Knowledge is rarely available in an explicit form, that is, it is almost never written down. Moreover, knowledge is often distributed across many sources. For example, in the graphical depiction here, to find out that this is the purpose of the project. The developer, the analyist, needs to talk to a lot of different people. And, to make things even worse. There are often conflicts between the knowledge gathered from different sources. A second issue is the fact that the knowledge is often tacit(沈默寡言的,心照不宣的). What is also called the say, do problem. In the example shown here. For instance. We have a customer that is describing to the analyst. The way in which he accomplishes a task. So it performs these three steps and reaches the goal. Whereas in practice, the actual way in which this task accomplished is by going through a larger number of steps to get to the same goal. So the point here is that, even if the knowledge were more concentrated, so not as spread as in this example. People simply find it hard to describe knowledge that they regularly use. So it is hard to make this knowledge explicit, to pass this knowledge to someone else. Yet another problem is limited observability. Identifying requirements through observation is often difficult as the problem owners might be too busy to perform the task that we need to observe. Or they might be doing a lot of other things together with the task that we need to observe, so that becomes confusing. That introduces noise. Moreover, even when this is not the case, the presence of an observer might change their problem. It is very typical for human subjects to improve or modify an aspect of their behavior, which is being experimentally measured in response to the fact that they know that they're being studied. You know that somebody's studying you and you change the way in which you behave. A typical issue. Finally, the information that we collect might be biased. For several reasons. People might not feel free to tell you what you need to know. Or, people might not want to tell you what you need to know. For example, in all the common cases in which the outcome might effect them, people might provide you a different picture from the real one. In order to influence you. So, they might have a hidden agenda, and mislead you, either consciously or unconsciously. So, all these issues add to the complexity of collecting requirements, of identifying the purpose of a system.

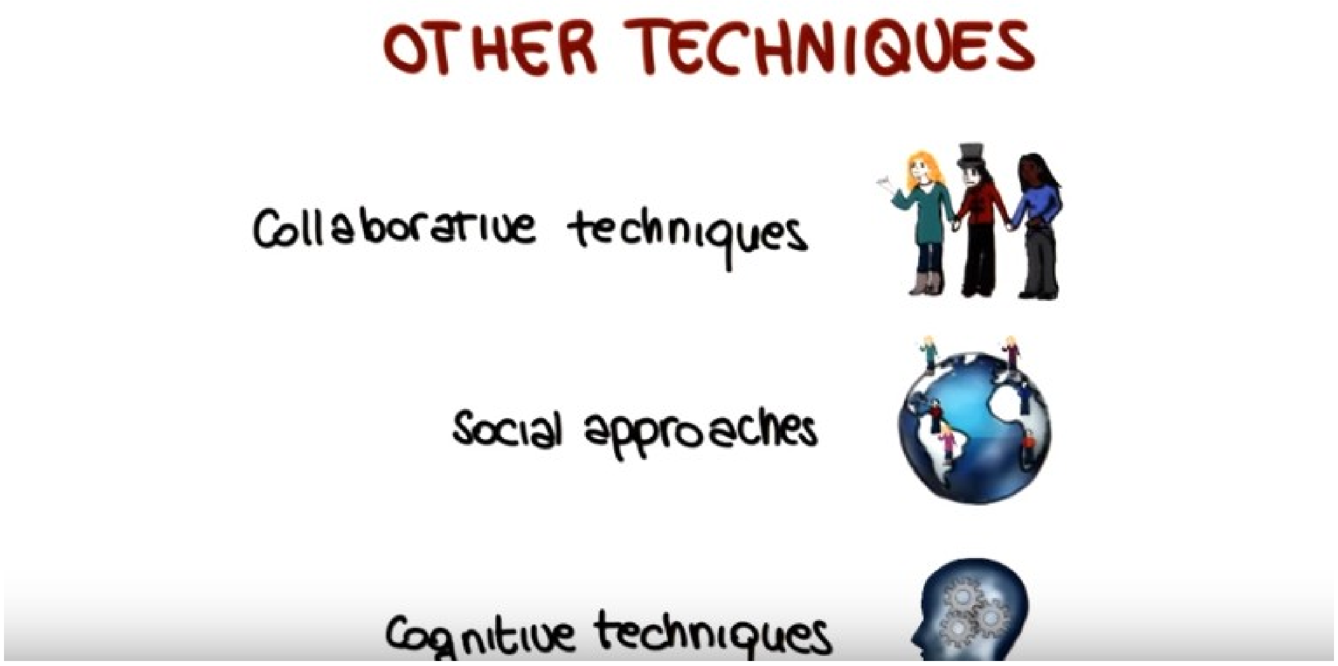


25. To cover the intrinsic problem of eliciting requirements, many different techniques have been proposed. So here I list some of most traditional techniques for requirement elicitation and as I present those, please keep in mind that these techniques can be used separately or combined. A first technique is called background reading. And, this technique involves collecting information by reading existing documents such as company reports, organizational charts, policy manuals, job descriptions, documentation of existing systems and so on. And, this technique is especially appropriate when one Is not familiar with your organization for which the requirements are being collected. So you want to get some background before interviewing actual people. And one of the main imitations of these kinds of approaches is that recent documents may be out of sync and they often are out of sync with reality. Tend to be long winded. It may contain many relevant details, so you may have to look at a lot of materials to extract enough information. The hard data and samples techniques consist in deciding which hard data we want to collect and choosing the sample of the population for which to collect such data and hard data includes facts and figures such as farms invoices, financial information, server results, marketing data, and so on. And the sampling of this data can be done in different ways. For example, the typical ways to do random selection. Interviews are another typical approach for requirement solicitation, and this is the approach that we use for the first project in this course, for instance. Interviews can be structured in which case there is an agenda of very open questions or they can be open ended in which case there is no preset agenda and the interview is more of a conversation. On the positive side, interviews can collect a rich set of information because they allow for uncovering opinions as well as hard facts. Moreover, they can probe in depth through follow up questions. On the more negative side, interviewing requires special skills that are difficult to master and require experience. And it is not enough to collect a lot of information. If this information is hard to analyze or even irrelevant, it might become useless. So you need to know how to conduct an interview in order to take advantage of these techniques. Surveys can also be extremely useful for gathering new requirements because they can quickly collect information from a large number of people. Moreover, they can be administered remotely. For example, by email, through the web. On the other hand, surveys tend to severely constrain the information that the user can provide and might miss opportunities to

collect unforeseen, relevant information. Finally, meetings are generally used for summarization of findings and collection of feedback,so as to confirm or refute what has been learned. So the only additional thing I want to mention about meetings is the fact that it is fundamental that have clearly stated objectives and are planned carefully. this is something that should be quite obvious, but doesn't always happen in practice.

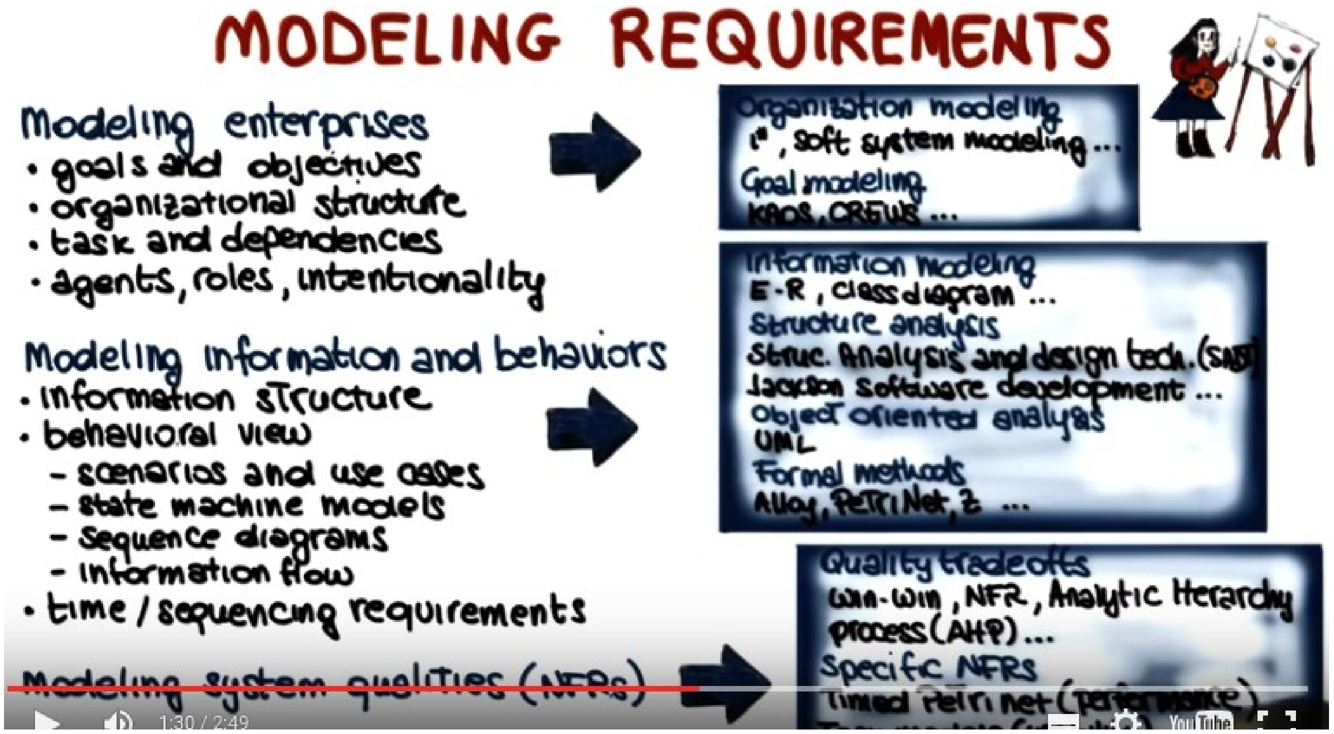


26. So just for completeness, I want to mention some other techniques besides the traditional ones that we just saw that can be used for requirements solicitation. And these other techniques can be divided in three main groups. There are collaborative techniques that were created to support incremental development of complex systems with large diverse user populations. An example of such techniques which is widely used and you might know is brainstorming. There are also social approaches and these are approaches, techniques that explore the social sciences to better collect information from the stakeholders and the environment. And among those I just want to mention ethnographic techniques which are based on the idea of collecting information on the participants by observing them in their original environment. Finally cognitive techniques, leverage cognitive science approaches to discover expert knowledge for example they can be used to understand the problem solving methods. And in case you're interested in finding out more about this and other techniques, I'm providing some references in the notes for the lesson.



27. Once we collected the required knowledge on the requirements for the system that we're developing, we need to model it in a structured and clear way, so that it can be analyzed and refined. And there are really tons of ways to do that, depending on your focus and objectives. More specifically, when modeling requirements you need to decide what you want to model and how you want to model it. So let's look at these two aspects independently. What you decide to model depends on where your emphasis is. That is on which aspects of the requirements you want to focus. For example if your emphasis is on the characteristics of the enterprise of the company that you are analyzing you may want to model goals and objectives of the company, or its organizational structure, its task and dependencies and so on. Conversely, if your focus is on information and behaviors, you might want to concentrate on aspects such as the structure of information, various behavioral views some of which we will see in the next lesson, or maybe time or sequencing requirements. Finally, if you're mostly interested in the quality aspects of your system, you will focus on the various non-functional properties of the software that are relevant in the context considered. For example reliability, robustness, security, and so on. You will just pick the ones that are relevant for your context. And as we said, there's a second dimension. After you have decided what to model in your system, you have to decide how you want to model it. So I want to show here some options for modeling enterprises, information, and quality aspects. And as you can see here for each type of information there are many possible models that we can use to represent it. And all these models have advantages and disadvantages, different levels of formality and different focus. Something else that I want to point out about these models is the fact that these models are often orthogonal to one another, especially if we consider models in different categories. So what that means is that they're complimentary rather than mutually exclusive. Different models can be used to provide views of the requirements from different perspectives, and we will not see most of these models in this course, but I wanted to list them anyways to give you an idea of how many there are and how vast is this area. As far as we are concerned in the course and for the projects we will express requirements using one of two main ways. Using natural language, that is informal specifications and using UML diagrams, which is graphical models. And we will introduce UML and the most important diagrams in the next lesson. And the only other type of models that I want to mentions explicitly are goal models because they're extremely popular. So the main idea with goal models is it start with the main goal of the system and then keep refining it by decomposing it in sub-

goals. So it's kind of a very natural way of progressing. And you continue this refinement until you get to goals that can be operationalized, and represent the basic units of functionality of the system.



MODELING REQUIREMENTS

Modeling enterprises
- goals and objectives
- organizational structure
- task and dependencies
- agents, roles, intentionality

Organization modeling
i*, soft system modeling...
Goal modeling
KAOS, CREWS ...

Modeling information and behaviors
- information structure
- behavioral view
  - scenarios and use cases
  - state machine models
  - sequence diagrams
  - information flow
- time / sequencing requirements

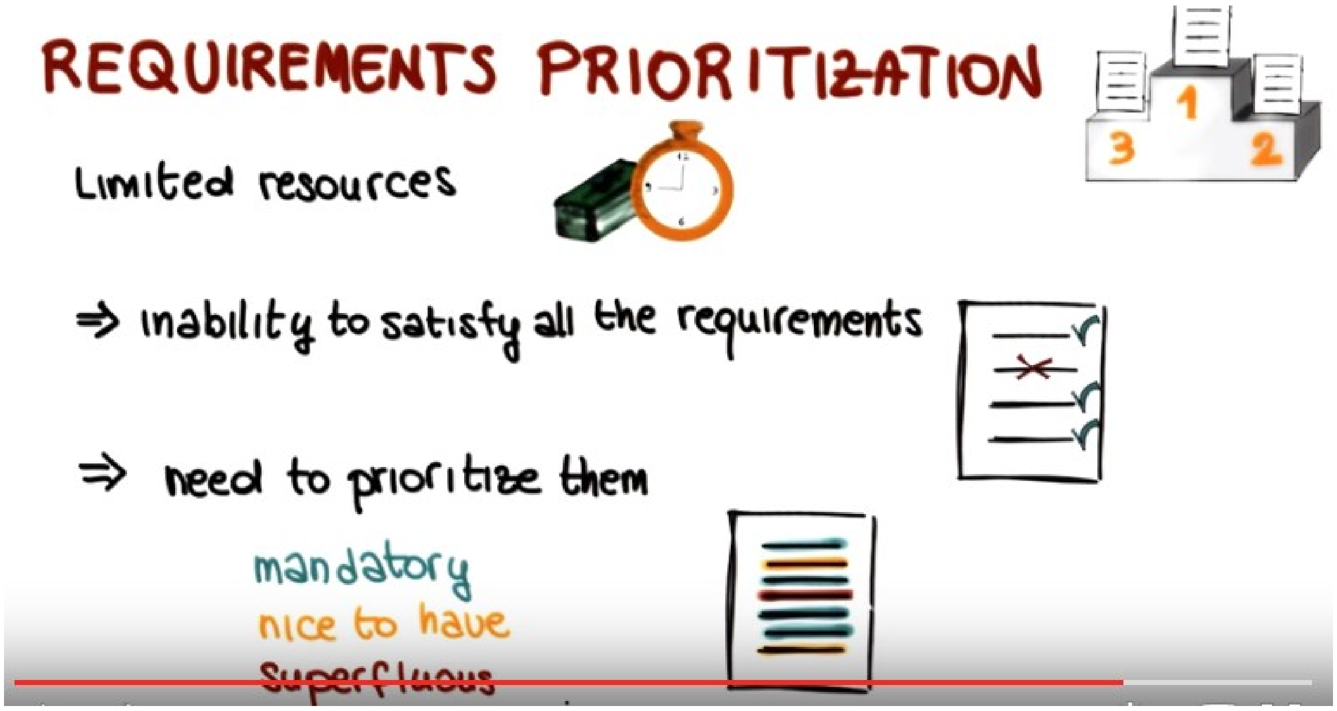
Information modeling
E-R, class diagram ...
Structure analysis
Struc. Analysis and design tech. (SA
Jackson Software development ...
Object oriented analysis
UML
Formal methods
Alloy, Petri Net, Z ...

Modeling system qualities (NFRs)

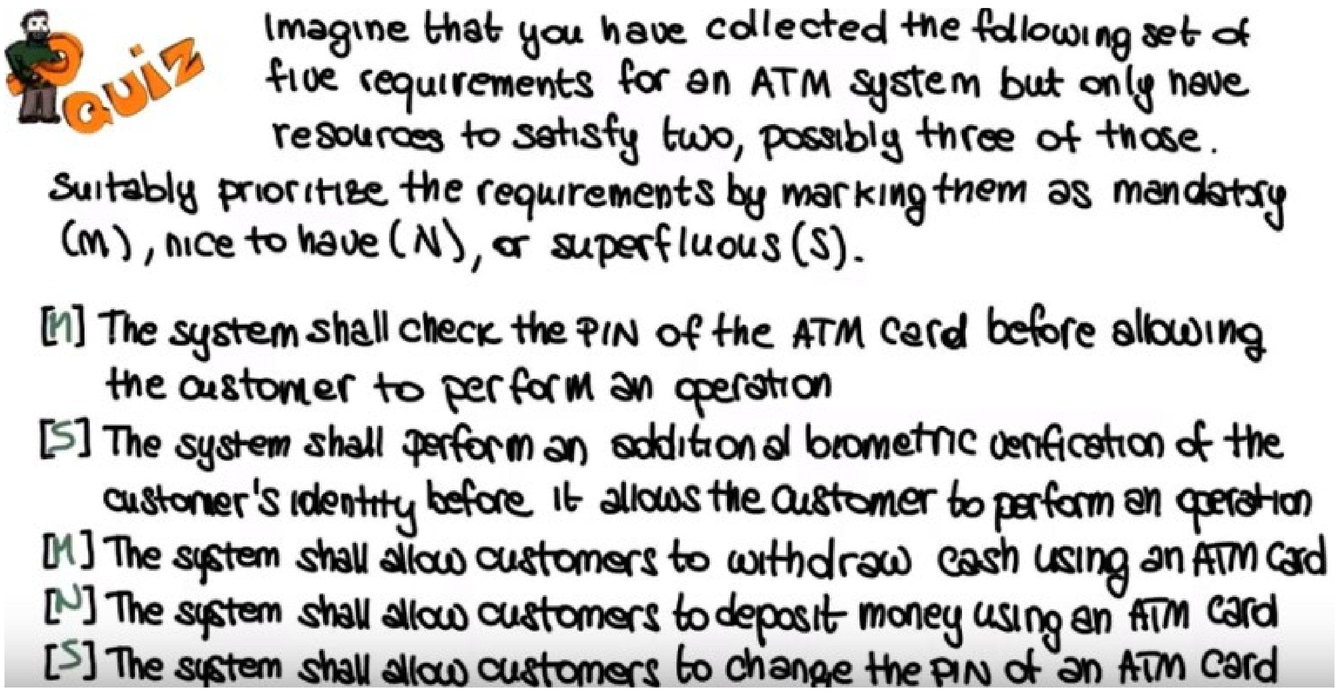
Quality tradeoffs
win-win, NFR, Analytic Herarchy process (AHP) ...
Specific NFRs
Timed Petri net (performance)

28. Now we are at the point in which we have collected and modeled our requirements. So the next thing that we can do is to analyze the requirements to identify possible problems, and specifically there are three types of analysis that we can perform. The first type of analysis is verification. So in this case we're talking about the requirements verification. And in verification developers will study the requirements to check whether they're correct, whether they accurately reflect the customer needs as perceived by the developer. Developers can also check the completeness of the requirements, check whether there are any missing pieces in the requirements as we discussed earlier. They can check whether the requirements are pertinent, or contain irrelevant information, like the one shown here. And they can also check whether they're consistent, unambiguous, testable and so on, so all those properties that should be satisfied for the requirements. A second type of analysis that is typically performed on requirements is validation. And the goal of validation is to assess whether the collected requirements define the system that the stakeholders really want. So the focus here is on the stakeholders. And in some cases, stakeholders can check the requirements directly if the requirements are expressed in a notation that they understand. Or they might check them by discussing them with the developers. Another possibility is that stakeholders asses the requirements by interacting with a prototype of the system, in case the requirements engineering process that is being used involves early prototyping. And finally surveys, testing, and other techniques can also be used to validate requirements. A final type of analysis that we can perform on requirements is risk analysis. And risk analysis aims to identify and analyze the main risks involved with the development of the system being considered. And if some requirements are deemed to be too risky, like in this case, this might result in changes in the requirements model to eliminate or address those risks. And note that all these analysis activities can be performed in many different ways depending on the modeling languages chosen to represent the requirements and on the context.

29. Why collecting, modeling, and analyzing requirements? We might realize that the resources available for the project are not enough to satisfy all of them. For example, there's not enough time, not enough money, not enough manpower. And therefore, there are some requirements that we won't be able to satisfy. In this cases, we must prioritize our requirements, by classifying them in one of three classes. The first class is mandatory requirements, and these are the requirements we must satisfy. Then there are the nice to have requirements. They are the ones that we will satisfy if resources allow. And finally, there are the superfluous requirements, and those are the requirements that we're going to keep around, but that we're going to postpone. For example, we might decide to satisfy them in the next release.


REQUIREMENTS PRIORITIZATION
Limited resources
⇒ inability to satisfy all the requirements
⇒ need to prioritize them
mandatory
nice to have
Superfluous
1
2
3

30. Now that we talked about requirements prioritization, let's try to see how this might work in practice. Imagine that you have collected the folowing set of five requirements for an ATM system, but only have resources to satisfy two of them. Possibly three. I would like for you to look at this list and suitably prioritize the requirements by marking them as mandatory, in this case you're going to put an M in the space. Nice to have, in this case you're going to put an N. Or superfluous, in this case you're going to put an S. This is the set of requirements, the first one says that the system shall check the PIN of the ATM card before allowing the customer to perform an operation. The second says that the system shall perform an additional biometric verification of the customer identity for example a check of the customer's finger prints before it allows the customer to perform an operation. Then we have that the system shall allow customers to withdraw cash using an ATM card. The system shall allow customer to deposit money using an ATM card. And the system shall allow customers to change the pin of their ATM card. So again, mark those as mandatory, nice to have, or superfluous considering the fact that you can satisfy only two, possibly three of them.



Imagine that you have collected the following set of five requirements for an ATM system but only have resources to satisfy two, possibly three of those.

Suitably prioritize the requirements by marking them as mandatory (M), nice to have (N), or superfluous (S).

[M] The system shall check the PIN of the ATM card before allowing the customer to perform an operation

[S] The system shall perform an additional biometric verification of the customer's identity before it allows the customer to perform an operation

[M] The system shall allow customers to withdraw cash using an ATM card

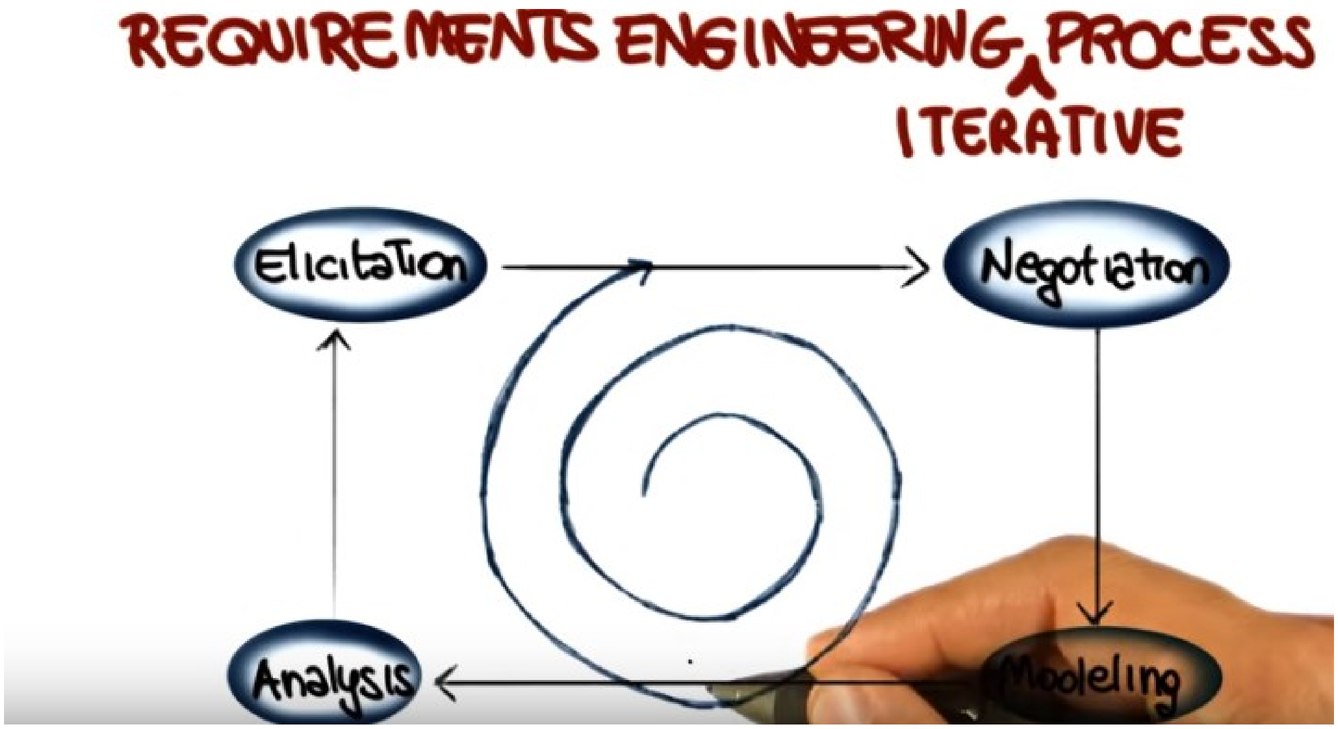
[N] The system shall allow customers to deposit money using an ATM card

[S] The system shall allow customers to change the PIN of an ATM card

31. Looking at the requirements, and knowing that we have only two that we can satisfy for sure, it makes sense to first mark as mandatory the ability to withdraw cash, which is the most typical use of an ATM machine. We are therefore going to mark this requirement with an M, for mandatory. It also makes sense to mark as mandatory the fact that the ATM system checks the PIN of the card being used by the customer, as that's the typical level of security that the customer would expect, therefore we're going to mark as mandatory also the first requirement here. And of course we could also perform biometric verification, but based on our knowledge of the domain, it seems like that should be an additional verification, rather than the main and only verification for the system. We will therefore mark it superfluous. That is something that we can postpone until a later release, the second requirement. Finally, another typical operation that customers perform at ATM machines is depositing. Whereas changing an ATM card's PIN is not such a common operation. We'll therefore mark it nice to have this fourth requirement and as superfluous, the last one. So at the end, what we have is that we have two mandatory requirements which are the two that we can satisfy for sure. One, nice to have the requirement, which is the possible third requirement which we might have time to satisfy. And the other two that are marked as superfluous, as something that we might do later, for example in a subsequent release. And of course there is something subjective in this answers. But again, based on our knowledge on our understanding of the domain, these are the one that makes more sense for an ATM system as we know it.

32. Let's now put together all that we have discussed and see how a requirements engineering process actually works. So, first of all, we saw that requirements engineering consists of three main steps. Elicitation of the requirements, in which we extract requirements from various sources. Modeling in which we represent the requirements using one or more notations or formal reasons and analysis, in which we identify possible issues with our requirements and there is actually a 4th step that we kind of mention but not explicitly. And this is the negotiation that can happen between the stakeholders and the developers, during which requirements are discussed and modified until an agreement is reached. So if you want to think of this as a process, so as a sequence of steps, we can see that we start from elicitation. So we start by eliciting an initial setup requirements. We negotiate and refine this set, then

we model the resulting requirements. And finally, we analyze such requirements. However, the process doesn't really stop here. Why? Well, because as a result of the analysis, we might have to perform further elicitation. And so this process is not really a sequential one, but rather an iterative process. So, in practice, we continue to iterate over these four steps gathering a better and better understanding of the requirements at every iteration until we are happy with the settle requirement that we gather and stop the process.



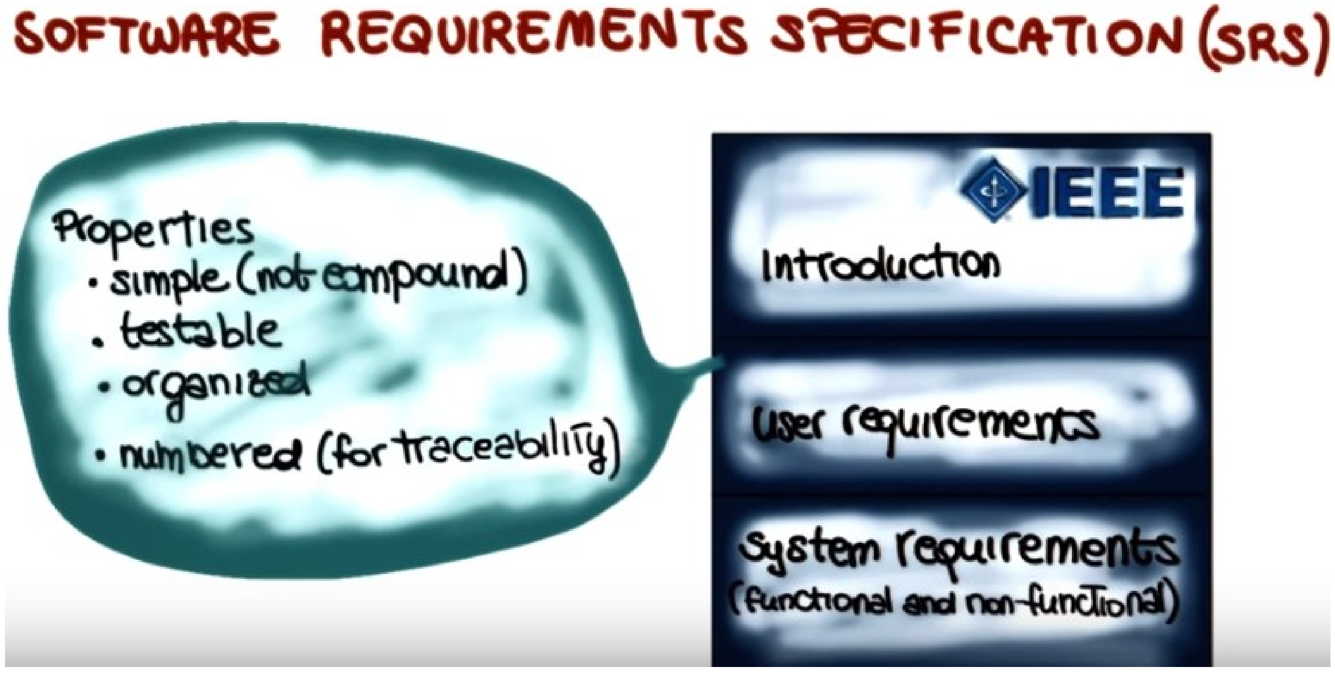
33. Before I conclude this lesson, I want to say a few additional things about the Software Requirement Specification document or the SRS. And I want to do that because this is a very important document and some of the projects actually require you to produce one. So why is the Requirement Specification such an important document? That's because a Software Requirement Specification document is an important fundamental way to communicate. Requirements to others. For example they represent a common ground between analysts and stakeholders. Note however, that different projects might require different software requirement specifications so you need to know your context. For example, the SRS document that you have to create for a small project performed by a few developers can in most cases. Be a concise and informal one. Conversely the software requirement specification for a multi project, involving a number of developers can be a fairly complex and extensive document. So again you have to be aware of your context and build your software requirement specification accordingly. In order to have a common format for the SRS document, IEEE defined a standard that divides the document in predefined sections. And in the context of this course, we will use a simplified version of the IEEE SRS format that includes three main sections. An introduction, which discusses the purpose, context, and objectives of the project. A user requirements definition, which contains the user requirements. And the system requirements specification, which includes both functional and non-functional requirements. And we provide more information about this format when we discuss the projects. So to conclude the lesson, I want to point out and in some cases recap a few important characteristics that requirements should have. First of all, requirements should be simple. Not compound. Each requirement should express one specific piece of functionality that the system should provide. Requirements should be

testable. We mentioned this before, but I want to stress it because it is a very important point. Untestable requirements such as the system should be fast, are useless. Requirements should be organized. Related requirements should be grouped, more abstract requirements should contain more detailed requirements, and priorities should be clearly indicated when present. Finally, requirements should be numbered, so that they can be traced. For example, numbered requirements will allow you to trace them to design. Implementation and testing elements and items, which is something that you might have to do for one of the projects. And that we will discuss in more detail in a later class.

SOFTWARE REQUIREMENTS SPECIFICATION (SRS)

Way to communicate requirements to others

Different projects require different SRSs.



SOFTWARE REQUIREMENTS SPECIFICATION (SRS)

Properties
- simple (not compound)
- testable
- organized
- numbered (for traceability)

IEEE

Introduction

User requirements

System requirements (functional and non-functional)