

This presentation about software project management actually compare that with I present a class in project management. And typically, that's a four or five-day class of eight to five with exercises. Tonight, of course, we don't have enough time for exercises. And in this class I cover about 30 subjects spread over five days.

Tonight I'm going to talk to you about three subjects. One is project management itself and, specifically, the idea of the broadness of the project management. I'll come to that later on, and I'll explain what broadness is here. But I will also talk about two tools that absolutely determine the flavor of project management that you get from me.

The first one is requirements and requirements engineering, which I look at as a tool, as a project management tool. It's more than that, but allow me to talk about it as in project management.

And the other is something that I will really explain, and that this process assessment. It's also a project management tool, and more specifically, it's a risk management tool. And these are the three subjects that I'm going to cover tonight. Instead of the 30 that you get in a full-time, 1-week course.

All of this, unless I specifically excluded it, all of this applies to both development and acquisition and the profession of software as a service. So, do not assume that I am just talking about software development here. And when I talk about software, and I specifically mean software engineering to include development and acquisition and the profession of a service.

OK. Let's start with the first section, requirements. This requirements, at the same time that belongs to the subject of requirements, I'm going to talk about stakeholders and stakeholder needs. And I'm going to talk about quality. And I will define those, and I will illustrate that.

But before I actually get into the technical thing, I need to give you a warning and heads-up here. Because what you hear in this section is not generally accepted. Now, there are many people that do accept it, but there are other people that don't. And that's what I'm trying to put forward here.

And that is in the profession, the requirements practice remains controversial. If you have a computer now or later, you can Google, for example, no stinking requirements, and you get a number of hits from software people that are still of the opinion that we don't need requirements in order to get good software. I say you need requirements to get good software. And doing requirements engineering determines the form of project management that you can apply.

So, what are the objections? I mean, these people have their reasons, of course. I mean, they say, we don't have time for requirements. And requirements do change. So why bother?

Some other people say, oh, but we have alternatives. Instead of requirements, we come up with a project charter and a document that outlines what the project is all about.

And I have done that myself. As I've dealt with managers that said, we don't have time for requirements. Why don't you do a project charter? And you'll learn, of course, along the way, and now I know better.

But I've done project charters. And you write things down what this project is all about, and then the document gets filed and nobody ever looked at it. And if somebody would look at it, it would not answer their question why they looked at it. So indeed, that is wasted time and effort to do a project charter.

So specifically, as a project manager and as an advisor to project managers, I still have to fight, in many cases, for doing requirements. Overcoming the objections of, we don't have time, there are alternatives. And then my arguments, of course, are the requirements will answer why this project, they set the scope of the project.

And what I have to do, then is admit up front that requirements will change. Yes, they will change. It's taken into account that they will change.

This is not the idea that you start with the requirements up front, and then you move like the waterfall down to design and implementation and test. So, I admit that up front, that requirements do change.

But then, I also point out that they are signposts and yardsticks in the course of the project. And they are critical in judging the readiness of the software at the end of your project. But overall, whether you do requirements or not do requirements, determines the kind of project management that you can apply. And what you see in the following slides is all based on the fact that, yes, there will be requirements.

I should also point out that what you see here, we did not learn in class. We did not learn from books. We learned this in practice along the way.

And I made mistakes and, hopefully, learned enough from that to do better the next time around. But I could not point to something in the literature or in class where we picked this up originally. Not at all.

Let me see. I promised to talk about, in addition to requirements, about three other concepts as well. Let's start with the idea of stakeholders. The idea of stakeholders counters the idea that all we need to be concerned about is the customer or the users.

And that's a very important point where we still stand out with general practice as well. So, what are these types of stakeholders? A stakeholder is a person with interest in anything related to the software. Which can be the

software itself and then any property of the software.

Originally, software engineering started out with an idea of interest in the function of the software. But I say here, not only the function but any property or attribute of the software. Whether it is the throughput of the software, the quality of the software, et cetera. Not just the function of the system.

And the kind of stakeholders that we talk about there are, sure enough, the customers there, the owner, the users. But also say, for example, the legal department. Nearly, from the beginning, in software project management, you have to deal with the legal department, as an example. Because you may be using somebody else's intellectual property. And you may be sued if that somebody finds out and takes you to court.

So the interaction with the legal department is very strong. And the legal department is a stakeholder often, nearly always, in my projects. These are people with an interest in the software itself.

The next one is, not only in the software itself, but there may be stakeholders with interest in the environment in which is software. This software may be part of a larger system, and the stakeholders, maybe their interest is not in the software itself, but maybe in the things around it.

For example, you'll hear the designer of a business application process. He's not so much interested in the software, as in finding out what the software is going to be so that he or she can do the design a business application.

And, finally, that took me some time to find this, but that is the stakeholders also include the people that may not be interested in the software or even the environment in the software, but are interested in the processes that you use. I know it was eye-opening for me that along one of the projects, the internal audit people. It's a typical department in a large corporation that makes sure that, financially, anything that goes on inside the corporation is above board.

And they came along and said, we are a stakeholder in your project and I objected. I said, but you're not going to tell me how I do my work, are you? And they said, yes, we are going to tell you how you do, at least on the following points. Because we want you to work in such a way we can know for sure that what you tell is going to be in the software, indeed, will be in the software.

So these people, they don't have an interest in the software. They don't have an interest in the environment in the software. But they're interested in the processes that we'll use for software engineering.

OK, here I have to tell you an IBM story. Somewhat in the beginning, I was a project manager. A small team, it was not a very big team.

But I was asked to take a piece of software, a program, that a salesman had written for his customer, a large American insurance company. And it was a very handy program. Also, the customer liked it. And IBM said, gee, probably other customers will like this program as well, this app.

[INAUDIBLE] and his team are going to generalize this for a worldwide market. Not just for this particular insurance company, but we're going to generalize it for everybody. And, sure, I started the job. And soon I found out that this particular project, relatively small, as it was, had more than 160 stakeholders.

Different organizations, different groups, different people, but always some stake in this not too big application. Now, there was the legal department in it. There was the pricing people in it. There was the world in it.

Also, in the case of IBM, IBM was, at that time, very much split between the American part and the rest of the world. So every function that you had on the American side, you had the world trade side as well. So, that was automatically doubling of the number of stakeholders.

But I was surprised to find more than 160 stakeholders for this relatively small program. On the other hand, and I will mention it later on as well, there is a trade-off between the width of your net that you throw out, there's a relation between the number of stakeholders that you take into account, and the quality that you can achieve.

And with these 160 stakeholders, we ultimately produced a piece of program, an app that you find back in every piece of IBM software these days. As well as it has been duplicated, imitated in many other pieces of software itself outside IBM. All of this, because attention was paid to a multitude of inputs on what this general piece of software had to look for where initially it was nothing else than an app program by a single salesman for a single customer.

I'm going to distinguish, and this is an essential point. Again, we kind of stand out compared with other stories that you may hear from other people. But we make a very sharp distinction between stakeholder needs, on the one hand, and software requirements on the other hand.

And I'm going to explain the difference. But let's start with stakeholder needs. Stakeholder needs is a condition that must or should be fulfilled according to one or more stakeholders. With the strength of the need varying between one need and another. With needs, hopefully, stated quantitatively, not good and bad, high or low, hot or cold, large or small. But in numbers.

Also with a clear owner of that need. If I have questions about that need, I know where to go and get more information. But also, one of the things that you find out if you do these things is that stakeholders are not too good in coming up with their needs. I mean, you have a session and you say, what are your needs? And the first

thing you look of, what do you mean?

So you have to, elicit is the word that we like to use in that. Draw it out of them. And there are several ways of doing that. I'll talk about that later on, as well.

So that's a stakeholder need. And the fulfillment of that need may depend upon the software itself, and/or the environment of the software, and/or the process of the software.

For example, with the process these days, there's a lot of interest in cybersecurity. Part of cybersecurity has to do with the fact of the processes that you use. So in order to get the proper cybersecurity, the processes are involved, as well in the fulfillment of that cybersecurity need.

All of this suggests, and I'll show you on the next page, that there is an infinite number of types of categories of stakeholder needs. Every new project that comes along, there is a possibility that you run into a type of need that you have never seen before. And, originally, there was only one need. There was functionality.

But then all these others came along. And I've just tried to remember a couple of projects, and I've written down what I could remember. But like I said, a new project comes along, and you may have a new need. Let me give you an example.

It has to do with cybersecurity, as well. But for very good reasons, governments ask certain pieces of your software to be breakable. I don't know whether I listed it here as a need, but systems must be secure, but not too secure. If there are, for example, lawful reasons to, for example, criminal reasons to go into records, it must be possible.

So, I'd never encountered myself that particular requirement. But, like I said, every new project there's a possibility that you run into a type of stakeholder need that you have not seen before.

All of this, and I'm not suggesting, not at all, that every project has all these kind of stakeholder needs. Typically, there are four, five, six, seven. If it's a large project, tens of stakeholder needs that really do matter. But all together as a universe in software engineering, these are the type of stakeholder needs that you may encounter. It all depends very much on the field in which you are working.

OK? I will now use a couple of diagrams to explain the difference between stakeholder needs and software requirements. So, let me start with this simple diagram. It's a glob of software. I am not suggesting nice, rounded software, but a glob of software in an environment. It's kind of an abstraction.

But all I try to do is explain the relationship between stakeholder needs and software requirements. So this is my

starting point. The next one is, in this diagram, I am now showing you what stakeholder needs are. And stakeholder needs, as a matter of diagram, are all over the place.

In fact, you see the piece of software is toned down. I mean, in fact, stakeholder needs should not be concerned at all about the borderline between the software and the rest of the world. They may be, and they should be, all over the place if things are normal.

They are irrespective of the software project boundary, and they are problem-oriented or goal-oriented. But they certainly are not solution-oriented. I'll come to that with software requirements.

Because if you keep this in mind and look at the next picture, you can start to see what I'm driving at. And that is the software requirement in that same kind of picture looks like this. This is what a software requirement looks like. And they delineate the subject software.

In words, software requirements are a condition on the software, or on the software process, to make software acceptable to the stakeholders. Acceptable to the stakeholders. That's what a software requirement is.

Now, there are many similarities between stakeholder needs and software requirements. And that's why it's easy to put them all together and treat them as the same. They are not.

But things that are the same between needs and requirements is that with requirements, as well, the strengths may vary. Some of the requirements are a must. Others are shoot. Others are may. Not all requirements are equal.

But the same is true for needs, as well. But requirements should also have a permissible cost of data. I mean, if it's a strong requirement and permissible and the cost to engineer this piece of software, this requirement, is low, it's more than likely that you'll pick up that requirement and include it in your project.

On the other hand, if there is a strength of a requirement is so-so, and the cost to engineer this requirement is very high, it's more than likely that you will drop that along the way. Because you need to complete your project presumably in a certain amount of time. Also, and that's another point, state your requirements in the positive.

For example, it's easy to say there should be no security holes. But it's very difficult to prove there are no security holes. Tell me what you do to close off the security holes, and then I can't verify whether that requirement has been fulfilled.

So, yes, one way of coming up with requirements is requirements in the negative. But they are very hard to prove, and I try to avoid them all together. OK? With that said, like needs, requirements may change. And they do

change.

Talk about requirement types, like I talked about the stakeholder needs. Similar to the needs, they address everything of interest to the stakeholders. Requirements must address everything that may influence the acceptability of the software at the end of the project.

And here you have, I could have come up with similar lengths of lists for requirements, but various examples of types of requirements. I include very much software processes and schedule. Schedule is one of those requirements that people typically try to put on the side.

No, they say, that's not a requirement. But then you say, can I come up with this project any time you like? No, no, we need to be finished by then.

That is a requirement that needs to be, and probably will be, traded off against function or some other quality. All of these things are requirements. What you see here is a very broad approach to the idea of software requirements.

Next, I will put them next to each other on the next slide. Like I said, I spent a good amount of attention to the distinction between stakeholder needs and software requirements. But because it's so essential to the approach of software project management that I will talk about. Stakeholder needs are the responsibility of the stakeholders.

On the requirements side, it's the software engineers that are responsible for it. Left is what the problem is. Right, what the solution should be. On the left, we do it by, for example, modeling of the domain, the business processes, the workflows, domain scenarios.

And mostly in the terms of feedback, same as needs, it's very difficult to get. But I had already mentioned it's very difficult to get stakeholder needs out of the stakeholders. So, it's primarily through feedback probing that you get an idea of what the stakeholder needs are.

On the right side, you have the modeling of the software in the form of use cases. You probably mentioned that it was mentioned in class as well.

My own favorite is defining software requirements in the form of test cases. My most successful projects and my easiest projects are where I was able to convince, or where I was able to consult, with project managers that do test-driven development. Do you talk about test-driven development in class?

Mentioned it.

Mentioned it. I love it, if it's possible. Because he used it as software engineer, as a project manager, you put

some test cases together, and you start talking with the stakeholders. You do a review, or you do an inspection. And you start to talk.

And before you know, you not only talk about this test case is right or wrong, it's a good test case or no, we don't need a test case. Very soon, you see the stakeholders scratching their heads and asking themselves, what this is system supposed to do? Only because you put in front of them a test case where they say, no, that's not a proper test case.

But before too long, they will ask themselves, what is the requirement. And then finally, for both of them, and you will see that coming back as well, you see ideas of agile engineering here. Stakeholder needs and software requirements are things that play throughout your project.

There's not the assumption that you can do a kind of a waterfall model and nicely do implementation and testing on the basis of very hard stakeholder needs and software requirements. They will strengthen along the way, but they certainly are not there in the beginning.

After requirements and stakeholders and stakeholder needs, I have one more subject that I promised to talk about under the heading of requirements, and that is quality. And consensus and quality. We're looking for stakeholder consensus.

And again, this is an element where we kind of stand out. There are several other ways of skinning the cat. I don't think they are nearly as good as what you see here.

This is what I've done. And that is that for stakeholder consensus, I stay away from stakeholder needs. Typically, stakeholder needs do conflict with each other. Not all of them, but some conflict with some others.

And the typical process there is to talk about with you as a project manager, to talk with the stakeholders, and try to split the baby or something like that. And come up with a need that they both can agree on. I have found that it's a waste of time. I've tried it. I've done it. But unsuccessfully.

What I have found, and what I do ever since, is that I get the discussion away from stakeholder needs, and I get them to agree, hopefully, on software requirements. Use requirements as a base on which to come to a consensus. So I happily move along in a project with conflicting stakeholder needs, as long as we can get consensus, and negotiate consensus, about a set of software requirements.

A priori on what is to be engineered, and a posteriori, after the fact that the actual software, indeed, is acceptable. OK? More.



That's also a very essential point and difficult point to act on. But the transformation from needs to requirements is non-deterministic. What I mean is that if you have a certain need, it does not follow that it leads to certain requirements. You have a choice.

It's non-deterministic. Which is very important, because in many cases, you seem to end up with a set of requirements that either you cannot get everybody to agree on or are infeasible. The project is too large.

And what you then have to do is you have to keep in mind, OK, if these were the needs, even if they were conflicting among themselves, but maybe I can come up with some other requirements that avoids the problem of a too-lengthy project or a standoff between one set of stakeholders and another set of stakeholders. I can tell myself that the transformation from needs to requirements is non-deterministic. But to act on that in the middle of a project is very, very difficult. It's one of the most difficult things.

Because here you are set on a certain path. You have it in your mind. You have to all figured out.

And then you run into barriers that seem hard to impossible to overcome. And then, somehow, you need the agility, the flexibility, to go back and say, OK, but my transformation from needs to requirements was non-deterministic. Maybe I can do a different kind of transformation. And with another set of requirements that avoid the barrier in which I seem to run.

Easier said than done. And I have to remind myself in each situation as it arises that, keep in mind, that you can go back to needs and try another transformation into different requirements that avoids your problem. It's a great opportunity, very hard to realize, but very powerful.

OK? With that said, there's more. Requirement must be formulated in a way that makes it possible for the test for conformance in the course of the project. It is difficult to come up with an example. But requirements could be stated in something that can only be verified after, say, a year in use.

For example, reliability is a typical example. Where a certain amount of up time is guaranteed. It's a requirement that I try to avoid and try to replace with requirements that I can verify at least by the end of the project, not after a year, a full year of operation.

So again, it's a thing that I have to keep myself aware of, that you have to avoid any requirement that is impossible to verify before the end of your project. And then, finally, a working definition for, a working definition. I emphasize working, because I'm going to do a little bit more about that in a moment. Is that quality is the degree that the software at the end of the project meets requirements.

That's a working definition. But from everything that I've told you so far, that probably seems to you like a

reasonable definition of quality. And, in fact, it's a definition of quality that is used by many in the field.

The one thing, I learned many things from IBM, but one thing I learned from IBM was that this is not enough. And some of you in previous sessions like this, raised, in fact, two questions that I'm going to talk about in a moment. Because they put question marks behind this statement of requirements defined quality. I've already put wiggly lines around that warning that more to come.

Let's talk about the questions. And they were excellent questions. The first question that I got in one of these sessions was, what if there's disagreement on the requirements?

And the answer to that is, my answer. What I've done. And what the answer in class was, the project manager negotiates.

To a large degree, software engineering is negotiating. And, particularly, the project manager and he negotiates in terms of requirements, rather than stakeholder needs. He has to be creative, or she has to be creative. You have to come up with different views.

Changing the language helps very much in the negotiation. And I wish that more universities would teach as part of the software engineering class negotiating skills. Because negotiating is a big part of the job.

And he or she can negotiate. For example, my typical approach is, OK, here's a current iteration of the software requirements. What's wrong with it? Tell me what's wrong with it? Give me your words. OK?

And then we start talking, and I can see maybe an opening where I can reformulate a requirement or change it all together. And that's what I mean with rinse and repeat. I formulate something and then you start over again. Do you agree with this? Et cetera, et cetera.

At the end of the day, what matters here is the profit and the mission of the organization. But then there's always the possibility that there's a persistent disagreement. I will mention the case here of IBM. It's long ago. I can talk about this openly.

IBM, at one point, had a project under way, Future Systems. And it was a big project. More than 5,000 people, engineers, involved in the project.

It was so big that, in fact, they drew, which was really not allowed at the time, they drew engineers from Europe to the United States to help. Because more engineers were needed. I was one of the people that was drawn out of Europe, and that's how I ended up in the United States with IBM's Future Systems.

Basically, Future Systems was the kind of system to end all systems. IBM had to come up with things like IBM 360,

IBM 370. And this was the next generation. And there never would be a need for any more systems than that. OK?

The technology was there, it seemed. But there was persistent disagreement. At the end of the day, IBM was not able to resolve that disagreement.

The stakeholders that were opposing this approach of Future Systems, they didn't articulate their reasons very well. But later on, you could reconstruct it. It basically was a standoff between the technologists coming up with a system incredible functionality, incredible other qualities.

And here were the salespeople. And there was a situation where there were host applications, and all they needed were more cycles, more power. OK? They were I mean they could sell computers like they were cookies. And if there were more powerful machines, they would sell them right away.

They didn't need more functionality. They didn't need systems that ended all other systems. They needed more power.

So they kept disagreeing Future Systems was technology driven, and the sales people, at the end, were able to get the project killed. But the technology was still good. I mean, the underlying idea.

So instead of great new functionality, the sales people got more power, I mean, more cycles. Machines more powerful.

And, for example, airline reservation systems were running out of power. The computers were not large enough. Airline scheduling or radar tracking or all of these applications, suddenly, were running out of cycles.

And, with the killing of this project, another approach was taken where the hardware improvements were passed through as more powerful machines. And the idea of the system to end all other systems died on the fine.

So, what I'm saying here is that, yes, disagreements among the stakeholders, or disagreements between the project manager and the stakeholders, can lead to a discontinuing of the project. And at the time, and every time that it occurs, it's not very pleasant. But it's much better than dragging on and wasting a lot of money and time.

And, in fact, leaving your sales people without powerful machines that they could sell right away to anybody coming along and willing to pay for it. So that's the answer to one question.

What if disagreement on requirements? The first answer is negotiate. The second question is, you may have to face the end of your project if the disagreement really continues.

Now the second question assumes that there is agreement on the requirements. All the stakeholders and the project manager, everybody agrees these are the requirements. But the question from the class was, what if the requirements are inconsistent, defective? Whether we have realized it or not, there may be things in these requirements that are nonsense.

So that was a question from one of you in a previous class. And I answered that with basically remarking that requirements are not only, at least initially, defective, they are largely unknown to begin with. Every project has a beginning. You have to build it up. But even after you have built up a number of these requirements, there are the emergent properties.

I call them the bane of the software project manager. Emergent properties are properties that appear that you could not predict by making models or making prototypes, beta software. You find things, behavior of the software, that were hard to predict from what you put in.

So that's one reason of the unknowability of software requirements, at least initially. Another, there are several more reasons, but another reason that I would like to mention is that whether we like it or not, the design that you choose after you've done the requirements, may influence your requirements. OK?

Over and over again, you run into situations where you start with requirements, and you come up with a design, and you show that to your important stakeholders. And they say, yeah, but now that I know how you're going to solve this problem, I have the following requirements. So they go back to their requirements and reformulate them.

And that's a very true case and one that you have to accommodate. So all together, it's not just that agreements are defective, may be defective, but they are, at least initially, largely unknowable as well. And you have to work on that.

OK what you see coming here is, of course, agile engineering, which I have shown here within spring and increments. That's how you can work this. What it is, is an incremental evolution of needs, an incremental evolution of the needs to requirements to the software. Each increment based on the best available needs and requirements. Each increment, a complete engineering cycle but short.

The theory is three weeks. I'll talk about that. each Increment is released and used. And, because of use, you get feedback. Again, this is theory. But that is not theory, and that is defined quality as satisfying the stakeholders.

And that's one thing that forever I will credit IBM with. They got us away, they got me away, and many other people away, from the idea that the quality is a matter of meeting requirements. No, ultimately, quality is satisfying the stakeholders, the customer, the users. But everybody else that is in that large group of stakeholders that your

profit probably has.

OK, to a large degree this is what you would like, that you hope, will achieve. Again, I showed this in the picture as a feedback loop. On the left, you have the requirements. Excuse me, the stakeholder needs.

And here's one increment, engineering of an increment. Remember, I always say engineering. I don't say development or acquisition, because that same loop applies to development and acquisition.

You engineer the next increment. You use the increment and then you have new or changed or refined stakeholder needs and software requirements. And you start the loop again.

And after a number of increments, you come to the end, and you release your final increment. One to four-week cycle, dependency on users able to take the delivery. Because you're very dependant on their use. And the stakeholder participation, hopefully, would be used as feedback.

Not too long ago, I've done a project very much in the spirit of agile engineering. Sorry, I was a consultant to the project manager for this project. And the agile method that they used was well-known. Scrum method.

And they did everything according to the book. But, what we found all together was a number of very practical problems. The first one was that designing in small chunks is nearly impossible. Certainly impractical.

You need more force. You cannot take part of your backlog and design that and hope that design will fit with what comes next. So practically speaking, that was a big problem.

And sure enough, before too long, we had an architecture and design group on the side looking further ahead and testing the increments, each increment against our overall design. But that was already taking away from the pure idea of agile engineering.

We also found that the quality of the increments, getting the increments to a quality where they could be used, was nearly as big as getting the final system to an acceptable level of quality. So, most of the time, the one to four-week rule was violated. We kept the increment longer in test. And, even then, it was very hard to get to quality.

But all together, the time it took, I mean, this was a company that made some business software. Very much for their cash flow, it was very important that they came out with new releases every 12 months. And that the new releases had quality. They had that time problems, and they had quality problems.

And customers were not satisfied with new releases. And if your company depends upon satisfied customers, they were feeling it. So, that's why they went to agile to begin with. After which, I was called in to see if I could help

them.

All together, the project did not take 12 months. It took nearly two years. So, what I learned, at least in this particular case, was that, certainly, it's not a faster.

Yes, there is this idea that after one to four weeks, you have a new cycle and you can do some testing. But that's not how it worked, at least in this particular case. So it took much longer.

The absolutely exciting thing, and at the end, the company was very satisfied with what they had done, including the agile approach. That the quality of the system, the customer satisfaction with the new release was great. It was better than, it was bad before. But the customers were excited.

Yes, they had to wait. But for a customer to wait nine or 12 months, that's not too bad. For a company that is dependant upon cash flow and income, it's very bad. But the company was able to overcome that.

But at the end, the customers were excited about the functionality and all the other attributes of the software. So, in my book, at least, based on this particular example, it's a great way to get to customer satisfaction, which is one of my goals. But it takes time. And it's certainly not a speedy method to get over and done. Yeah?

You're talking about the designing in small chunks, it also includes, the big part that includes, is the impact assessment that you need to do on the existing system--

Absolutely.

--and there, again, then you have a distributive system. For example, you have the same software launch in North America and you have the same software launch in Europe and in Asia. Because of the different countries involved, and so on, the impact assessment that you have in North America could be significantly different than the same software functioning in Europe or in Asia.

Yeah.

So, the one to four-weeks rule, though the software works perfectly, North America might absolutely blow up.

Yeah. I will give you a little bit more about the background of this. I am not going to mention companies, but this is a company that makes timekeeping software and timekeeping hardware.

If you look at your local Wal-Mart store, it has a couple thousand employees. And time is kept with a machine and cards and software that tracks all these things, and makes sure that you comply with labor laws. There's a whole lot, there are millions of lines of code behind this.

And you're absolutely right, this whole software and this whole system, is not just for North America, it's for the world. It's for China. It's for Europe.

And what we may think here works, may not work over there. You get into different, like I already said, labor laws are very different between jurisdictions. So there are complications galore to this. To just think that after one to four-weeks you can get back to another working system is baloney.

More questions?

I still go, now and then, I go to a store, and I casually ask people, I mean, people that work at Trader Joe's and places like that, you like your system? Oh, yeah, yeah.