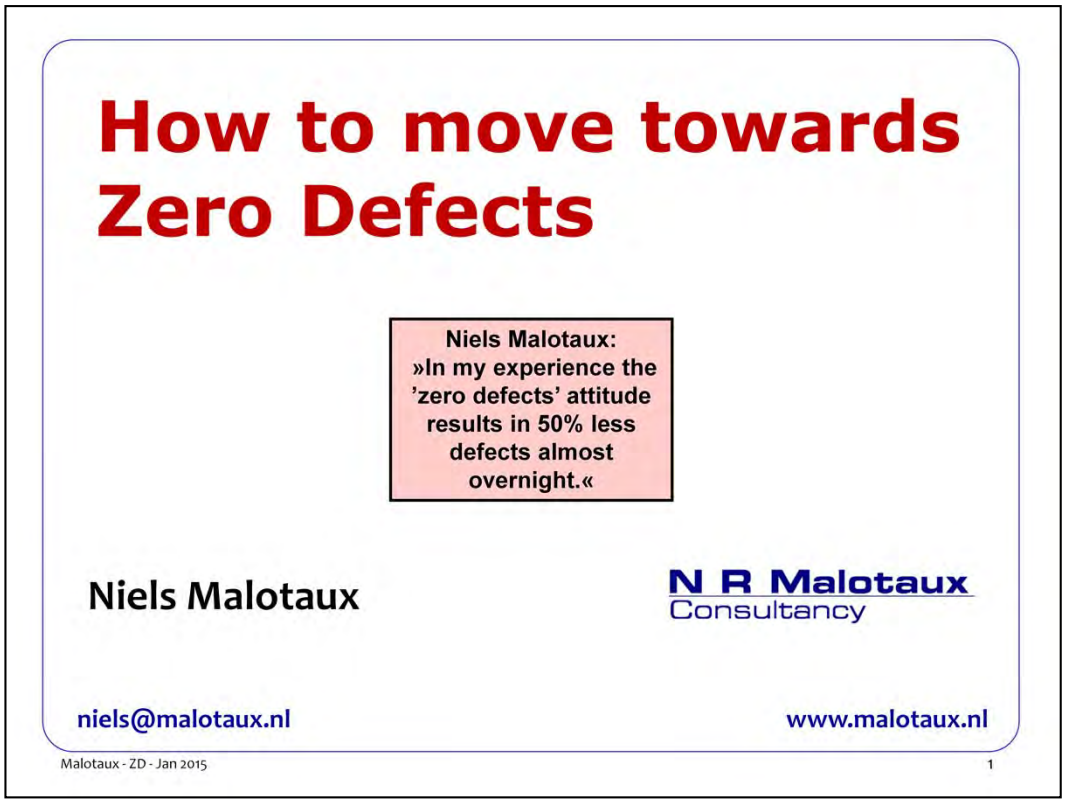
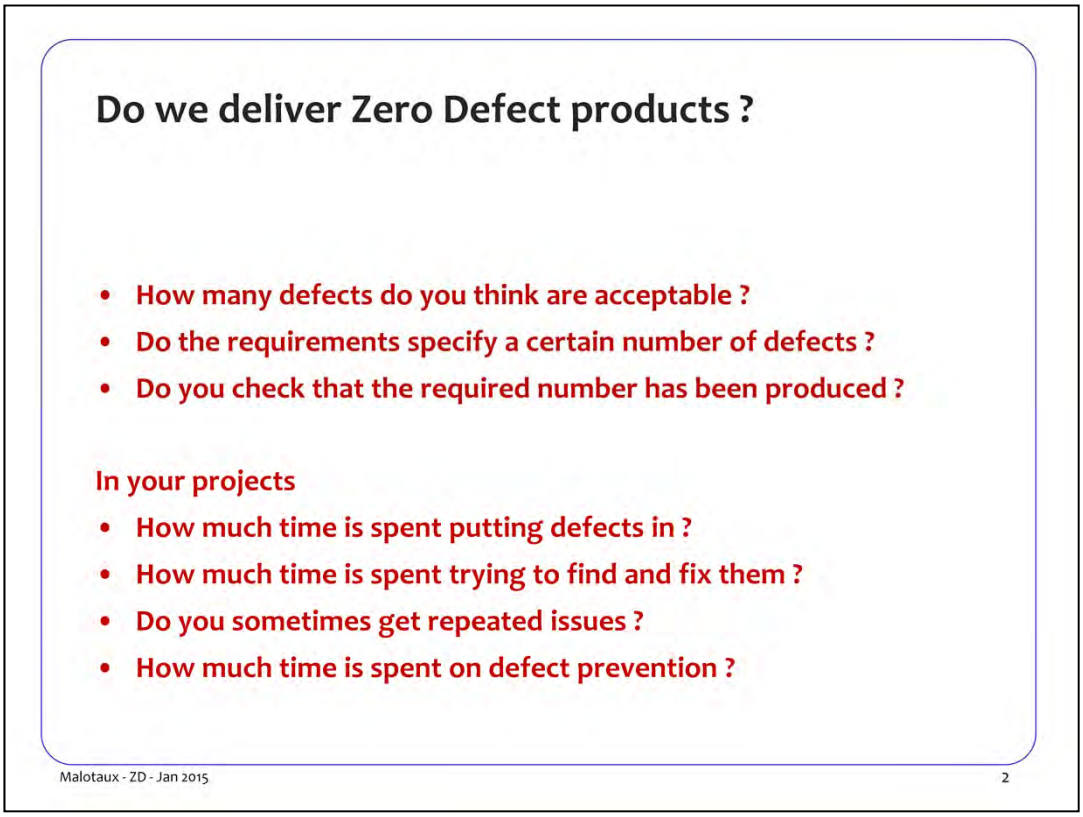


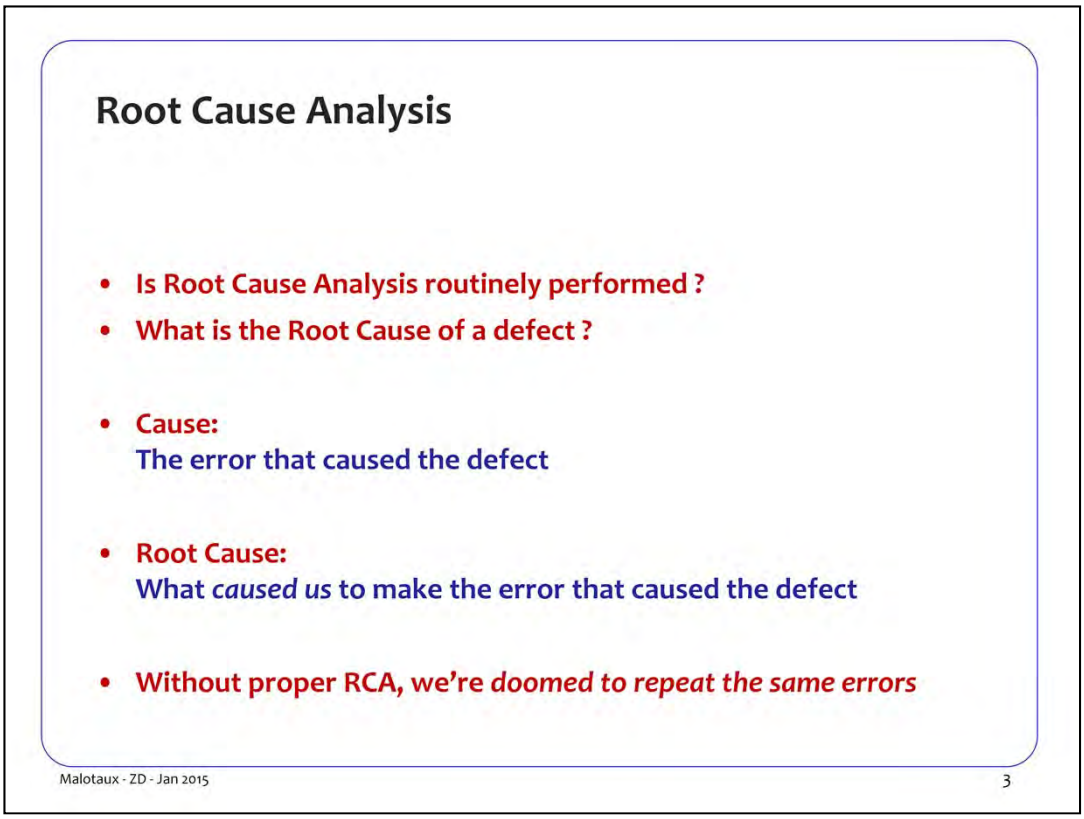

A colleague found this remark in one of my tutorials: "In my experience the 'zero defects' attitude results in 50% less defects almost overnight" and asked me to explain this to a project team he was coaching. I thought that my experience with Zero Defects might be interesting for more people than just this project team.

Note after the presentation at the London Test Management Forum (http://uktmf.com), 28 Jan 2015:

Several people in the audience confirmed: Every time the software was first properly designed before coding, we experienced better quality in less time. It's as simple as that.




As many people think that even talking about ZD is useless, I'd like first to discuss some questions with the audience.



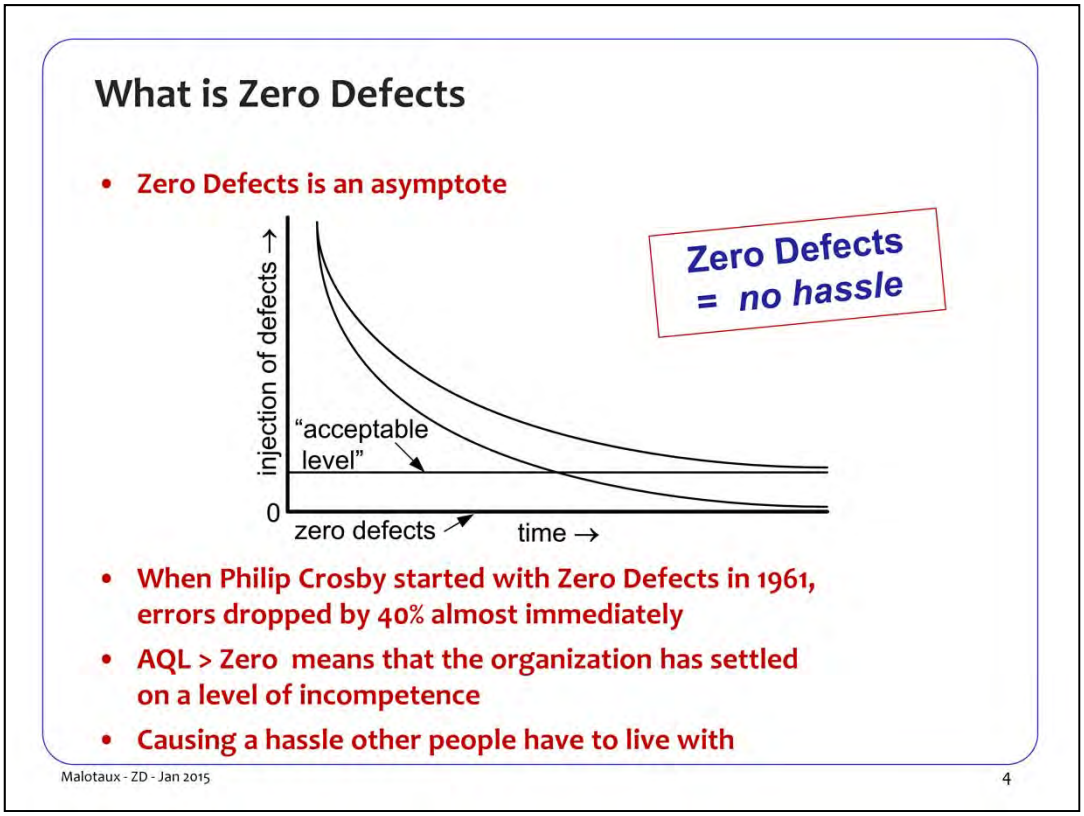

Years ago I suggested to add a box for the 'Root Cause' and for the 'Root Cause Suggested Solution' in a bug-tracking system. When I later checked how people were using this, I found that in the Root Cause box they documented the cause of the bug and in the Root Cause Suggested Solution box the suggestion how to repair the bug.

Apparently, they didn't see the difference between 'Cause' and 'Root Cause':

- The Cause of a defect is the error that caused the defect
- The Root Cause is what caused ***us*** to make the error that caused the defect

In another project I asked the project manager what they do with the results of the code reviews. "People repair the bugs" he said. I asked: "Don't you do Root Cause Analysis, in order to learn how to prevent this type of error from now on?" The response was: "On every issue we found??? We have no time for that!"

Apparently they have no time to learn to prevent, and rather spend a lot of time to find and fix(?). No wonder that projects take more time than they hoped for.



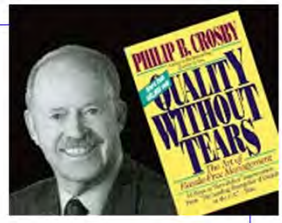

When I actively started using the Zero Defects (ZD) concept in software projects, defects made decreased by at least 50% almost immediately. It took about 2 weeks before the developers understood that I was dead serious about it. Then the testers came to me saying: "Niels, something weird is going on: we don't find issues anymore! It simply works!" I said: "Isn't that exactly what we want to see? Now testing is becoming a real challenge, namely proving that there are no errors."

So, even if you don't believe that this can be true, if two people (Crosby and me) did it and showed a huge decrease of errors *made*, only by adopting the attitude, isn't it at least worth a try, especially if you realize that about half of most projects is spent on finding and fixing defects. That's a huge budget. Any savings on that is probably well worth trying.

"No Hassle" proved to be easier to use than ZD: Don't cause a hassle. No hassle to yourself, to your peers, to your organization, to your customers.

.

## Crosby (1926-2001) - Absolutes of Quality




- Conformance to requirements
- Obtained through prevention
- Performance standard is zero defects
- Measured by the price of non-conformance (PONC)

Philip Crosby, 1970

- The purpose is customer success
  (not customer satisfaction)

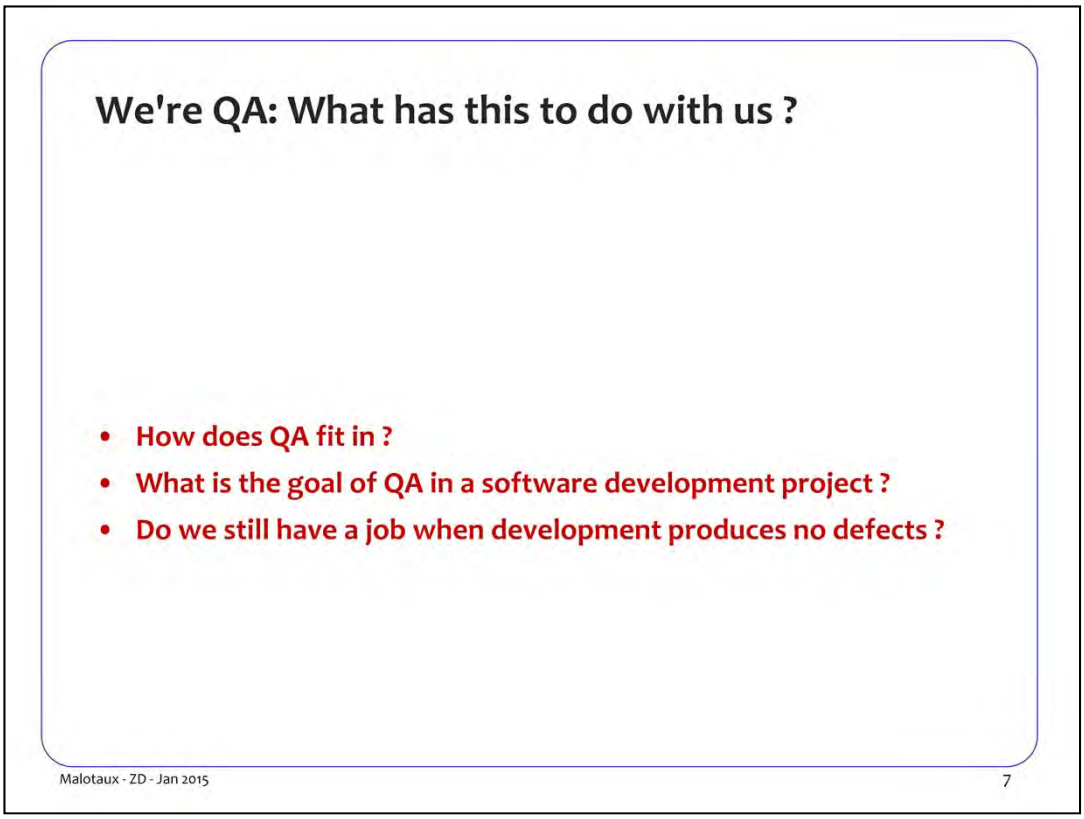
Added by Philip Crosby Associates, 2004


Malotaux - ZD - Jan 2015 5

Philip Crosby defined the four 'Absolutes of Quality'. When I started as a coach in a company recently, I gave his book "Quality without tears" to the CEO for homework: "Next week I'll check that you read it!". He did and it immediately had an impact on his behaviour. He delayed a major release to first get rid of the hassles that we were going to deliver to the costumers. He also calculated the 'Price of Non-Conformance' (PONC), to be at least a quarter of a million Euro in the past year.
Phil Crosby's organization later added a 5th Absolute: Customer Success. I agree completely. But I don't agree with them adding "… not customer satisfaction". After all, the customer should be successful, but satisfied as well, as we'll see on the next slide.


This is to me the top-level requirement for any project or any work we do.

- The customer is the entity that orders and pays. The customer, however, in many cases doesn't use the result of our project himself. He gets the benefit through the users of the result.
- What the customer says he wants is usually not what he really needs
- The time he needs it may be earlier or later than he says
- If the customer isn't satisfied, he doesn't want to pay
- If the customer isn't successful with what we deliver, he cannot pay
- If he's not more successful, why would he pay?
- What the customer wants, he cannot afford. If we try to satisfy all customer's wishes, we'll probably fail from the beginning. We can do great things, given unlimited time and money. But neither the customer nor we have unlimited time and money. Therefore: The requirements are what the Stakeholders require, but for a project: the requirements are what the project is planning to satisfy.
- The customer is king, but we aren't slaves. Both sides should benefit and be happy with the result.
- We will get the best result in the shortest possible time, but not shorter than possible. The impossible takes too much time.




Many people equate QA with Testing. Testing actually is just one of the quality measuring instruments of QA and hence only a small part of QA. If people use QA when they mean testing, I take the freedom to assume that testing should do QA. Let's first shortly discuss what QA actually is and who the customer of QA is.

## Who is the (main) customer of Testing and QA ?

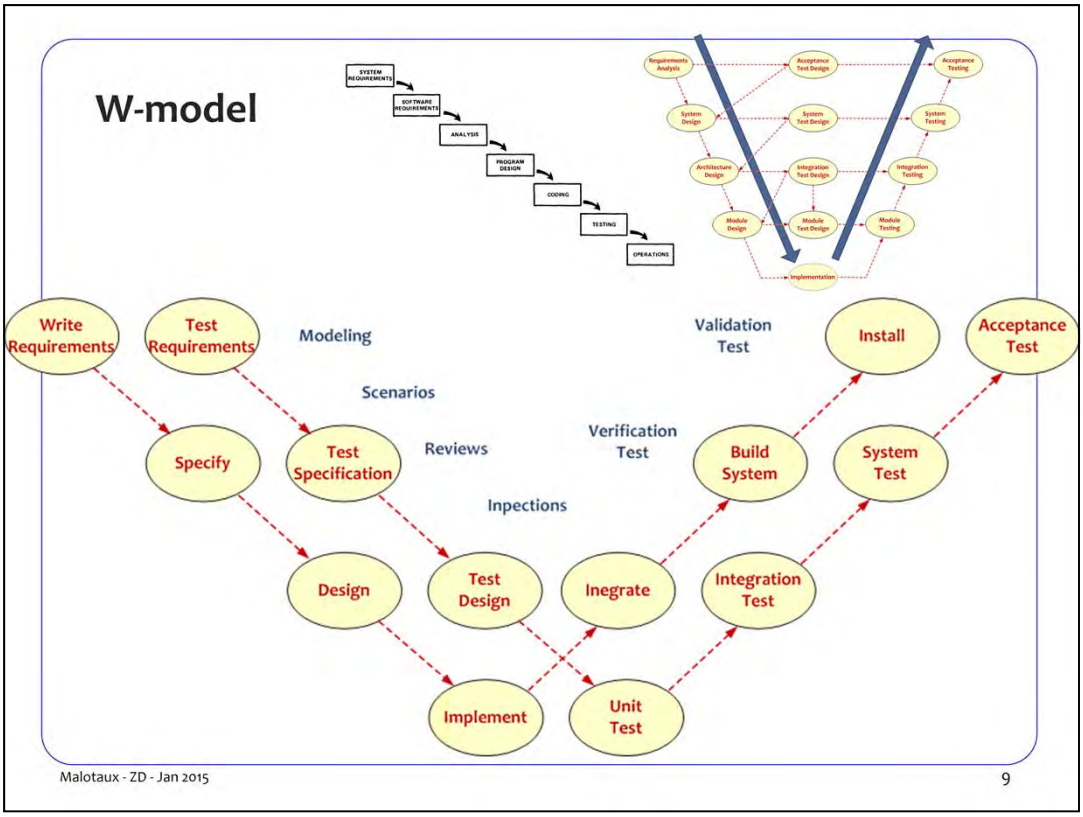
- **Deming:**
  - Quality comes not from testing, but from *improvement of the development process*
  - Testing does *not* improve quality, nor guarantee quality
  - It's too late
  - The quality, good or bad, is already in the product
  - You cannot test quality into a product
- **Who is the main customer of Testing and QA ?**
- **What do we have to deliver to these customers ?**
  **What are they *waiting for* ?**
- **Testers and QA are *consultants* to development**
- **Testing and QA *shouldn't delay* the delivery - How ?**

Deming
(1900-1993)

Malotaux - ZD - Jan 2015 8

I experienced that to most testers this quote from Deming is quite a paradigm shift and usually comes as a shock. But usually it's a shock of recognition! It will change their attitude for the better forever.

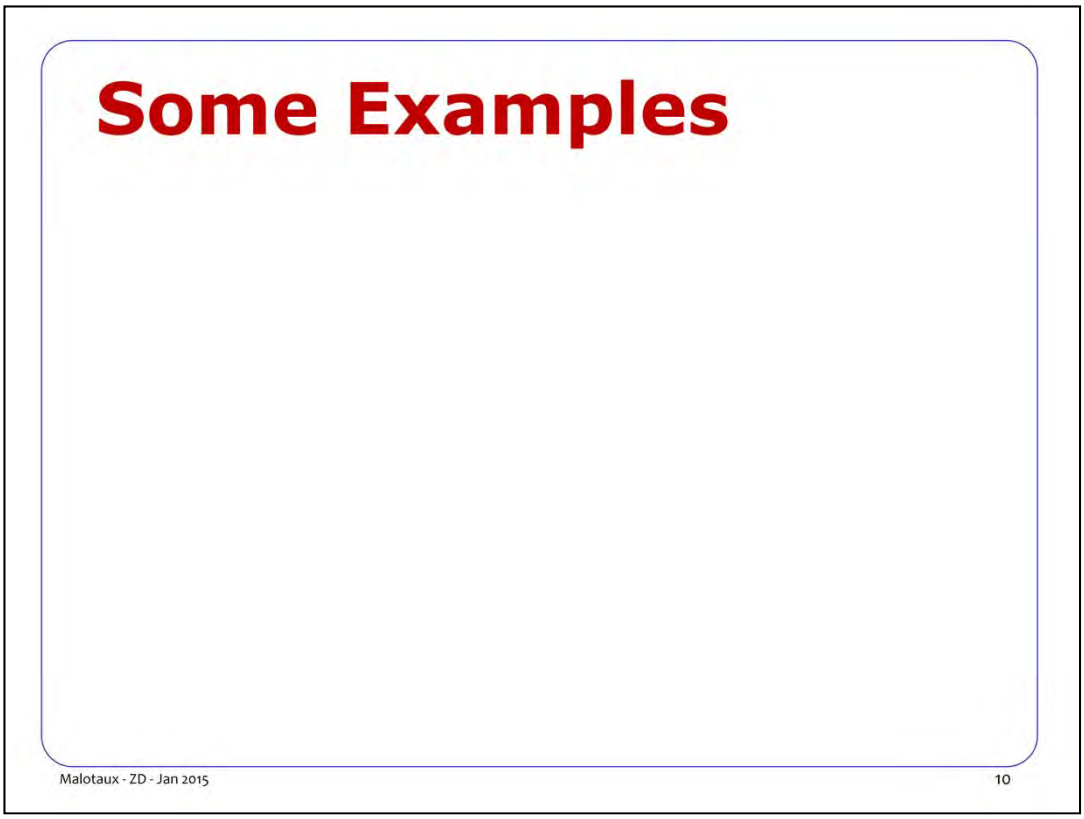
Now let's see how we can optimize our contribution as consultants to development.



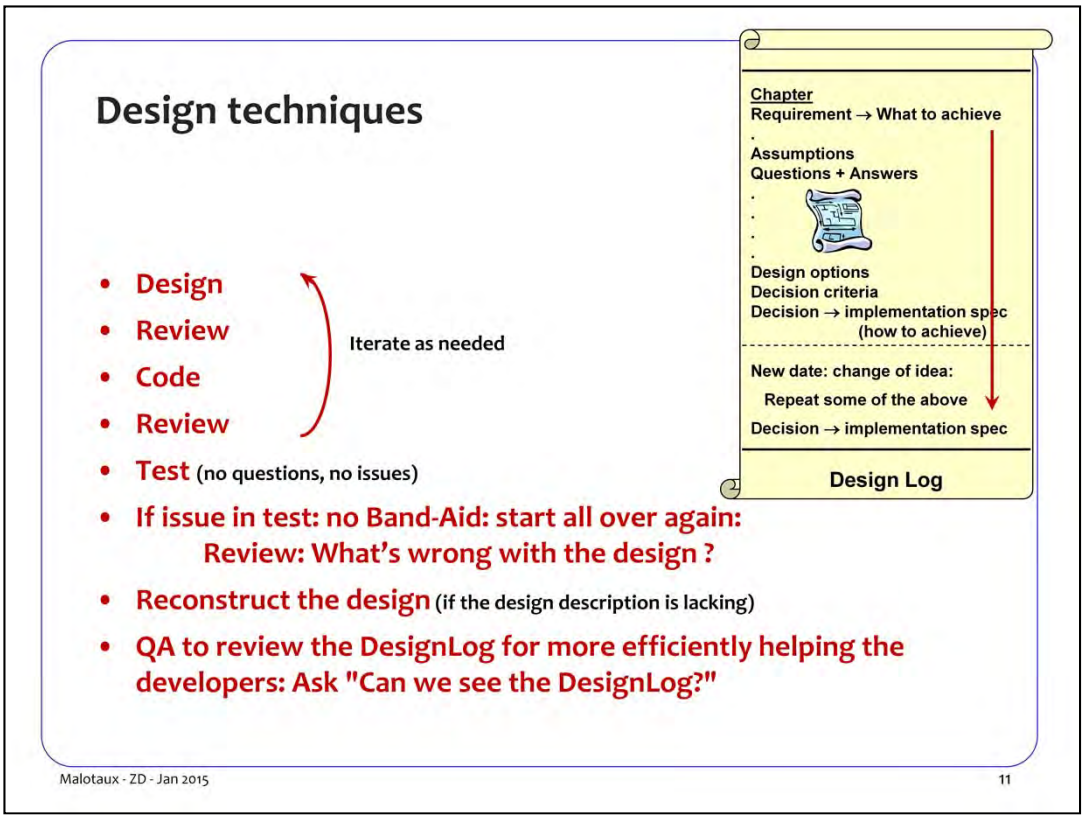

In the Agile world, the Waterfall and derived models are often seen as bad. However, these models are still valid for every sprint. After all, we have to determine what value we should deliver (requirement), how we can and are going to realize it (design), how we implement it (coding), integrate it, test it, and deliver. For QA the challenge is not to test only the code and find bugs, but to help development to prevent the bugs in the first place. By reviewing the requirement, the design and the implementation, so that the final test can conclude that it simply works as it is supposed to work.

The V-model is actually a folded Waterfall, where the most expensive issues (requirements issues) are found at the end. The W-model shows that we should find the issues once they are created, rather than when they cause trouble later. All models are wrong, some are useful. If they're useful, we should use them.

The requirement, the design, the code: they're all different manifestations of the same product. However, only the code can be run to check that it does what it is supposed to do. That's what we usually call 'testing'. Before we have code, there are other techniques to check that the product is right: like Modelling, Scenarios, Reviewing, Inspecting. In these areas QA can prove its value as well.




Let's discuss some examples of techniques that helped me and others to move towards Zero Defect deliveries. QA should be aware of these and other techniques in order to do a proper consulting job to development.




There are many techniques known to approach ZD faster. One of them is what I call the DesignLog.

When I started my career at Philips Electronics in 1976 (at the same time Philips started to sell its first microprocessor), we got a notebook to note our thoughts, experiments and findings chronologically. It was difficult, however, to retrieve an idea I had several weeks before, because it was buried in many pages of hardly readable handwriting.

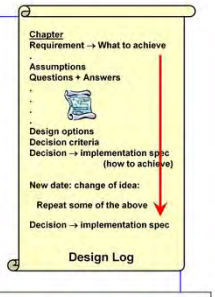
Nowadays we can use a word processor, add pictures, organize by subject rather than chronologically, and search through the text. We log our thoughts in chapters, which start with what we have to achieve (requirement), end with how we think we will achieve it (implementation specification), with in between the reasoning, assumptions, questions and answers, possible solutions, decision criteria and the selected solution (design).

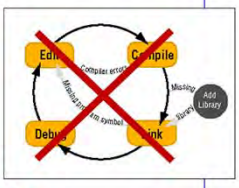
If I see design documentation, this often only shows what people decided to do, rather than also recording why and how they arrived at this decision.

The DesignLog should be reviewed to find possible issues before we start the implementation. Because the choices and design are well documented, in the maintenance phase (often a the largest portion of the cost of deployment of software!) minimum time is lost. One of the requirements for the DesignLog is: "If someone has to change something in the software one year later, he should be up and running within one or at most two days."

When QA asks development to review the DesignLog, if there is one they can review and also use this information to define and optimize their test-cases. If there is none, this is a good time to introduce the concept. See next slide.

## In the pub




**James:**

*Niels, this is Susan*

*Susan, this is Niels, who taught me about DesignLogging*

*Tell what happened*

**Susan:**

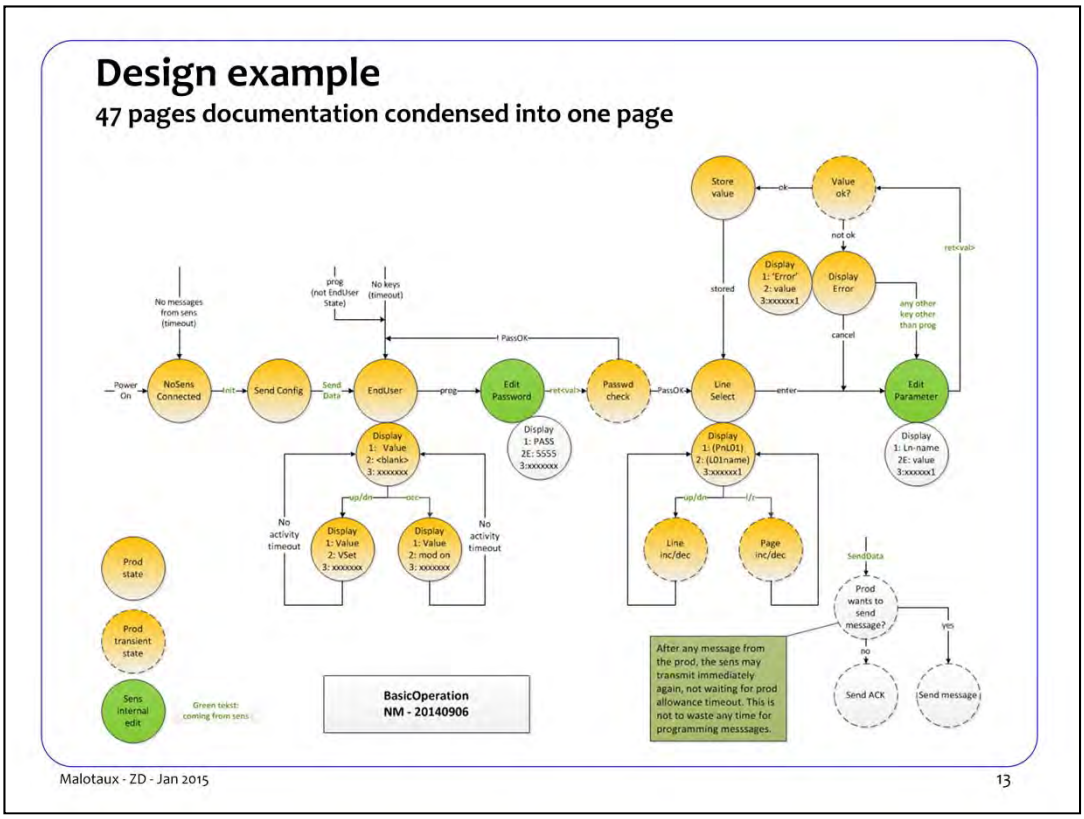
- *We had only 1.5 week to finish some software*
- *We were working hard, coding, testing, coding, testing*
- *James said we should stop coding and go back to the design*
- *"We don't have time!" - "We've only 7 days!"*
- *James insisted*
- *We designed, found the problem, corrected it, cleaned up the mess*
- *Done in less than 7 days*
- *Thank you!*



This happened just a few months ago. It's always nice to experience that the techniques that worked for me and for many others in the past, still work today. Many old techniques never get out of date.

We see, however, that it's not so easy to convince people to do something that seems counter-intuitive: going back to the design rather than grinding on in code and leaving a lot of dangerous scars in the process.

Delivering quality often needs counter-intuitive measures.



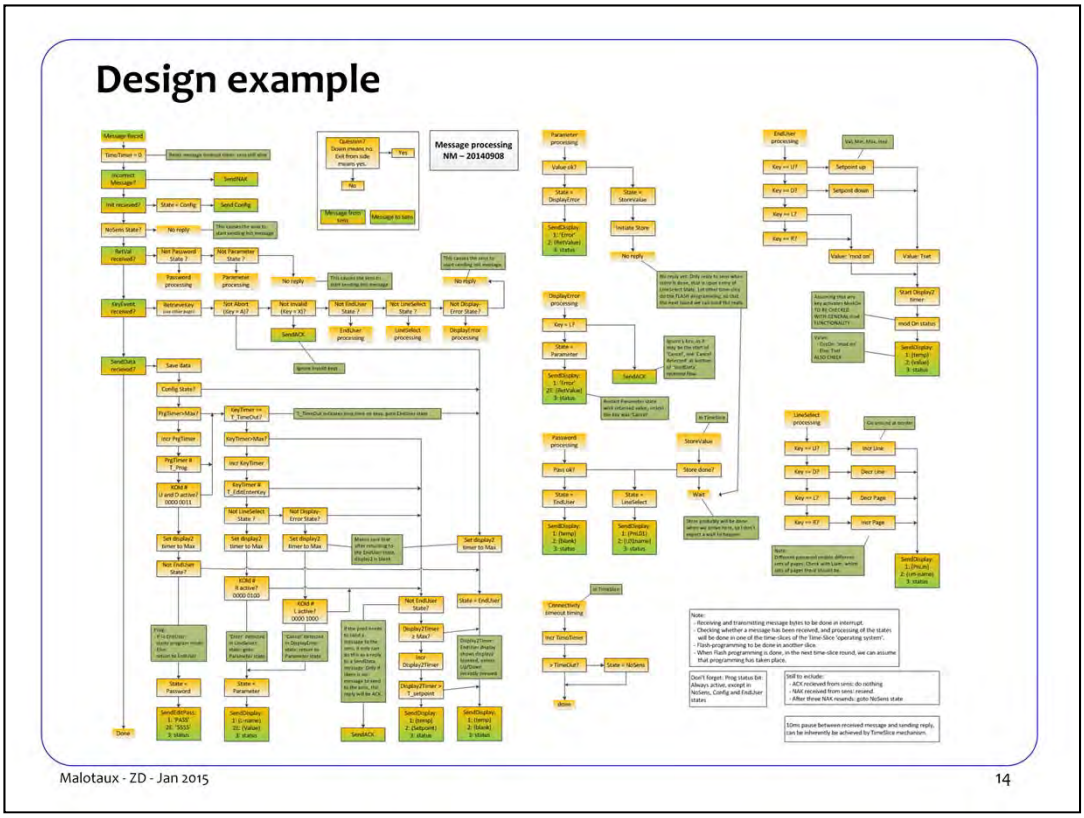

If I see documentation at all, it is usually just text. Sometimes a lot of text. One of my mantra's is: "Where are the pictures?"

This and the next slide are an example of some design I made recently (anonymised). You don't have to check the text and what it actually does. It's just to show some examples of concisely documenting functionality in a way most people, with a bit of understanding, can follow immediately.

This slide shows a design of the communication between some controller and a remote user interface. It was documented in a 47 page document by an 'architect'. 47 pages of interface description is almost impossible to oversee by humans, hence it contained a lot of inconsistencies and the people who had to implement it actually ignored it.

Once I made this one page overview, we could discuss, ease out the inconsistencies, make decisions, agree, and everyone knew exactly what to do. Conclusion: just documenting isn't enough. We have to learn how to document for usefulness.

QA can ask a developer to explain how the interface should work. If the developer only shows code to review, we know we have a problem. If the QA person doesn't understand the explanation, the explanation apparently isn't clear enough, which is a big risk for the quality of the result. If it's only text, it won't work either.

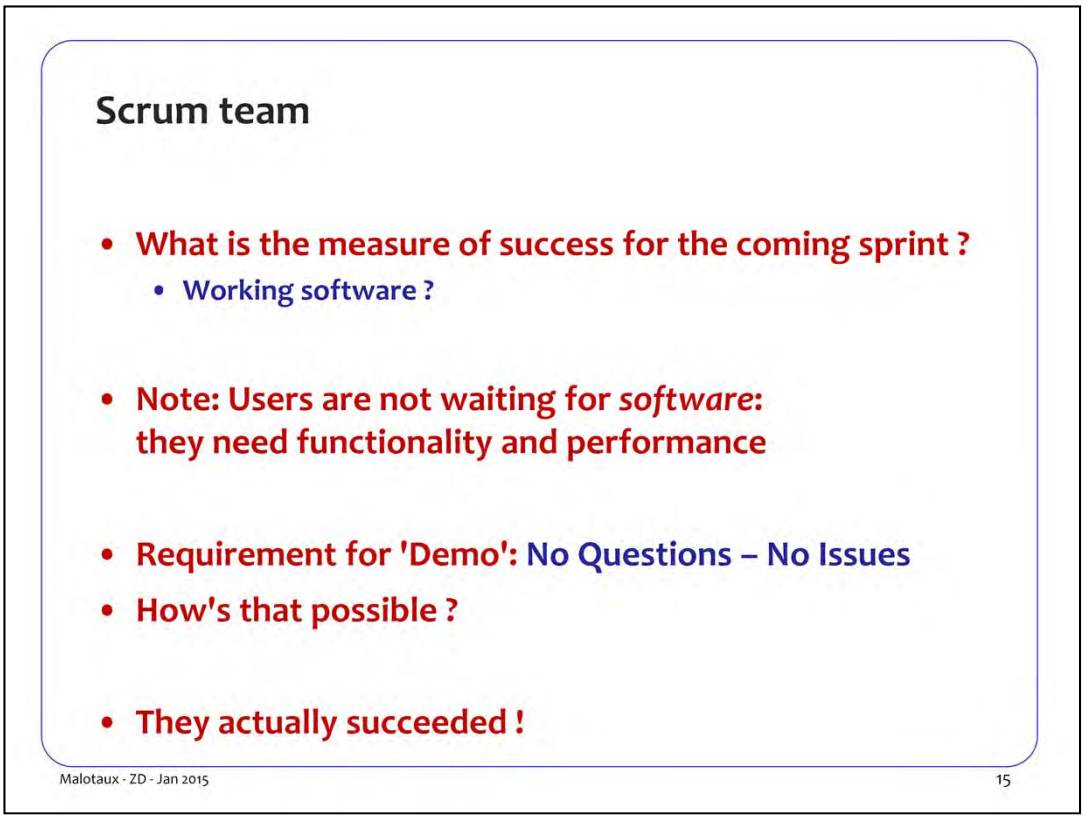



This is how I implemented the communication design based on my discussions with the suppliers of the remote user interface. The design was made to be reviewed and then it could readily be implemented based on this design. If I see how much I moved and reshuffled before I was content that this was right, I cannot imagine how this could be done properly in code without having this design. Like in the Cleanroom Approach to Software Development I designed down to a level of some 3 lines of code per design element. Sorry, I have no time now to go into detail, but the Cleanroom Approach routinely delivered an order of magnitude less defects in shorter time. Making changes in the code is not allowed before we have updated the design. The code should always be derived from the current design. Reviews of code should always check that the code does what the design says

These were just examples. The challenge is every time again to find the right representation that is easiest to comprehend

Of course the projects the audience is working in usually do these things properly. But I still see too often that the 'design' is only in the mind of the developer who writes the code, or just a rough sketch, with devastating effects in software quality and delivery time.

If as a QA person you encounter these effects, think what you should do about it.




I came in an project of some 70 people, with 3 Scrum teams of some 12 people each. We know 12 is too many, but that's another story.

At a Sprint Planning meeting I asked one of the teams: "What would be the measure of success for this Sprint?"

They looked at me: "What a strange question. We're Agile, so we deliver working software. Don't you know?"

I asked: "Shouldn't we have a measure of success, to know that we really did a good job?" and suggested: "No questions, No Issues". That's easy to measure: one question or one issue and we know we failed. No question and no issue and we know we were successful.

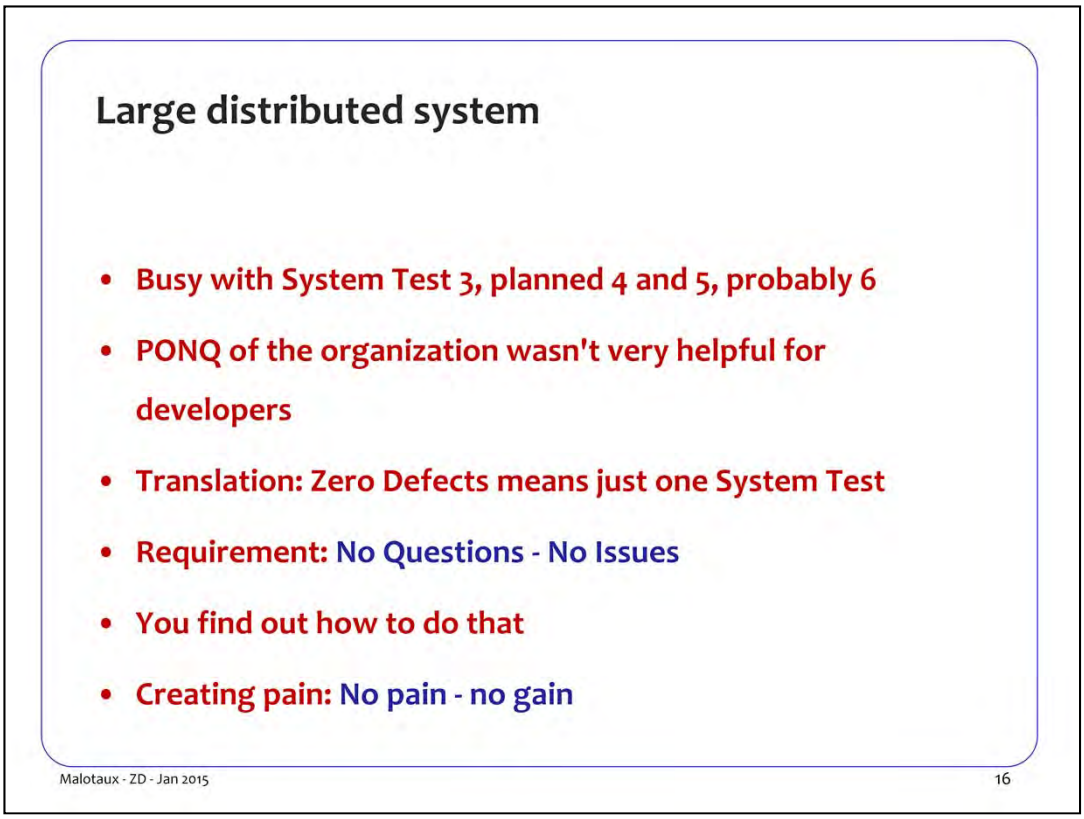
Their first reaction was: That's impossible! Surely there will be some questions when we deliver and there are always some issues.

I suggested: "You find out how to do it. It's just a simple requirement: "No questions, No Issues".

Interestingly, they immediately started thinking how they could deliver according to this requirement.

For example, someone thought: "Ah. Perhaps halfway the Sprint we ask someone to check it out and to see whether he would have any questions?" I said: "You're on the right track. Just find out how to do it. The requirement is simple."

Actually, I didn't expect them to be successful in this first Sprint, perhaps after a few. Surprisingly, they were successful. I'll tell how.

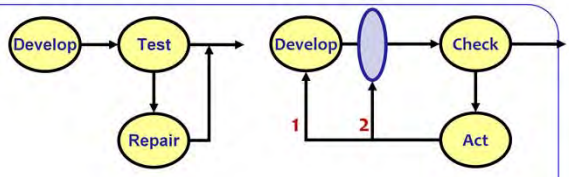



A large distributed system: electronic controllers, hardware and firmware, networking and pc software. When I arrived at this project, they were in 'System Test 3' and had ‘System Test 4’ and ‘System Test 5’ in the planning. Thinking about how to make the concept of ZD operational I thought: "Actually, to be First Time Right (another equivalent of ZD), there should be only *one* System Test, shouldn't there?"

I discussed this with the team and suggested: "No Questions, No Issues" at System Test. Before that people can discuss, ask, review, repair, whatever they like, but at the formal test, things simply work properly. After all, when delivered to users the software should work properly on its own - we're not there to help the system and the users.

I suggested to use ‘System Test 4’ as an exercise and ‘System Test 5’ as the real one. Actually, ‘System Test 5’ still failed within an hour. But gradually the team started to understand the concept, started using DesignLogs and reviews before coding and produce much less defects. ZD is not just turning a switch and then we don't deliver defects anymore. It takes time, but if we're serious about it, we can learn very quickly. If we're not serious about it, we'll never get there

## Zero Defects at test




What we often see

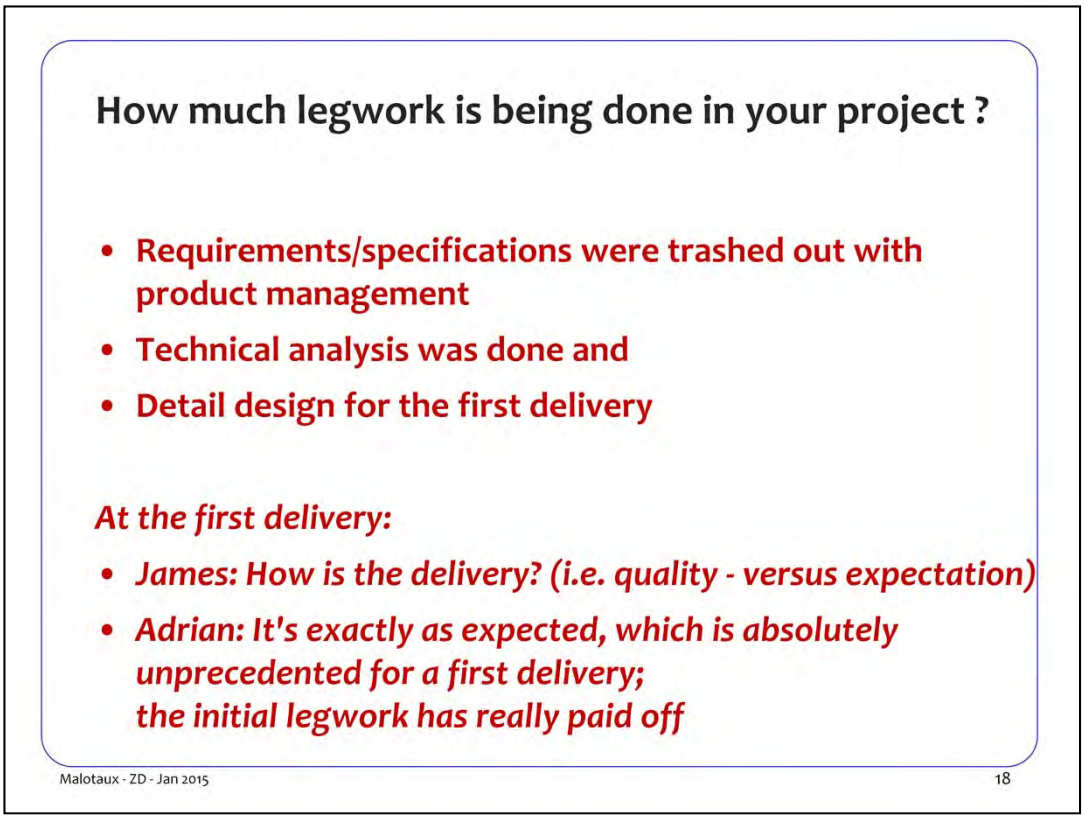
What we should expect

- What is delivered simply works
  If it doesn't work, nothing was delivered
- At the formal test: "No questions, no issues"
- Before that, in daily integrations:
  work together, ask anything, learn anything, improve anything
- Design Log - Review - Code - Review
  iterate as needed *before* test (don't waste testers' time)
- Daily integration ("That's impossible !" Of course it was possible!)

→ The shorter the feedback-loop the better they learn to prevent

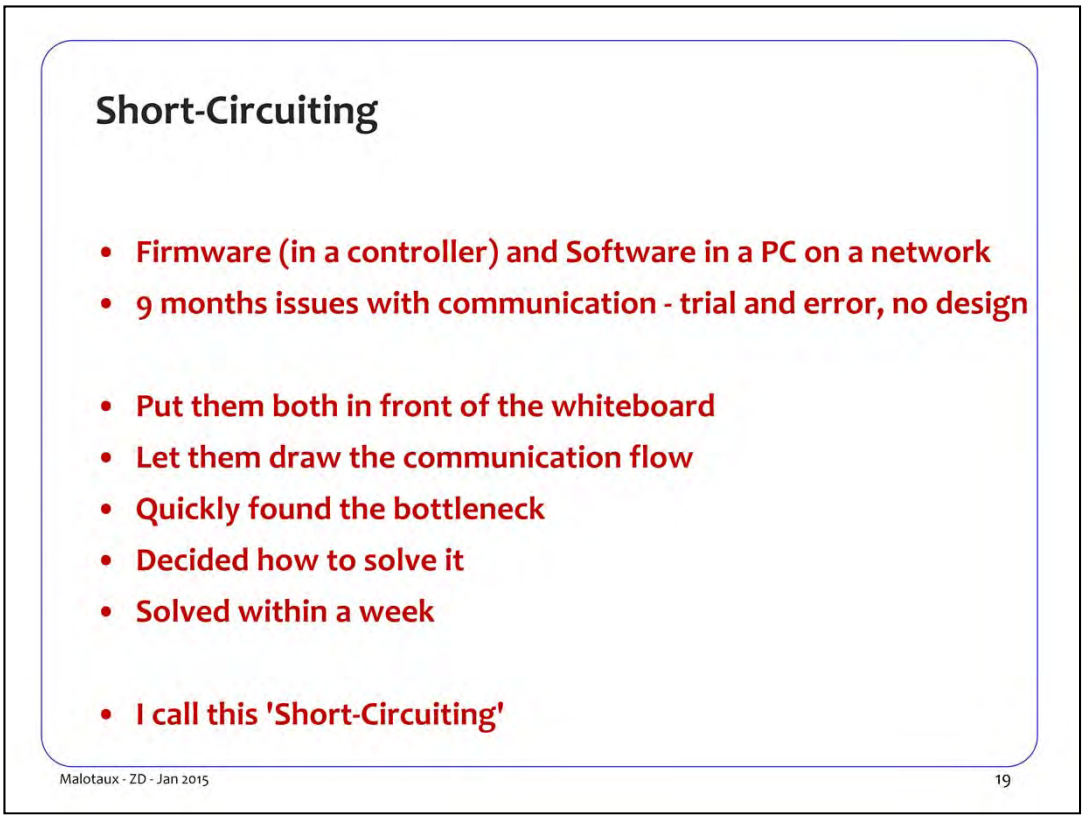


Some conclusions of the previous story.




This case was an organization with extraordinary bright people. In many projects we have to explain things over and over again, but in this project people needed only half a word to understand and do things better.
James (their new QA person) told me this story. He asked them to prepare well, design properly, and then do the coding. The result: "*It's exactly as expected, which is absolutely unprecedented for a first delivery.*"

He suggested it, they did it, and it worked. It's great for a QA person to work in such a fertile environment!




An example of ‘Short-Circuiting’:

Firmware in a controller has to communicate with a PC over a network.

They had been working for 9 months to get it to work and the communication still didn't work flawlessly.

Every time something went wrong there was a lot of debugging, tracing the traffic on the network to see where things went wrong, often accusing ‘other traffic’ on the network as the cause, which of course isn’t a good excuse at all.

The firmware engineer then suggested some Band-Aid, a retry, or extending some time-out. All these time-outs made the communication unacceptably slow.

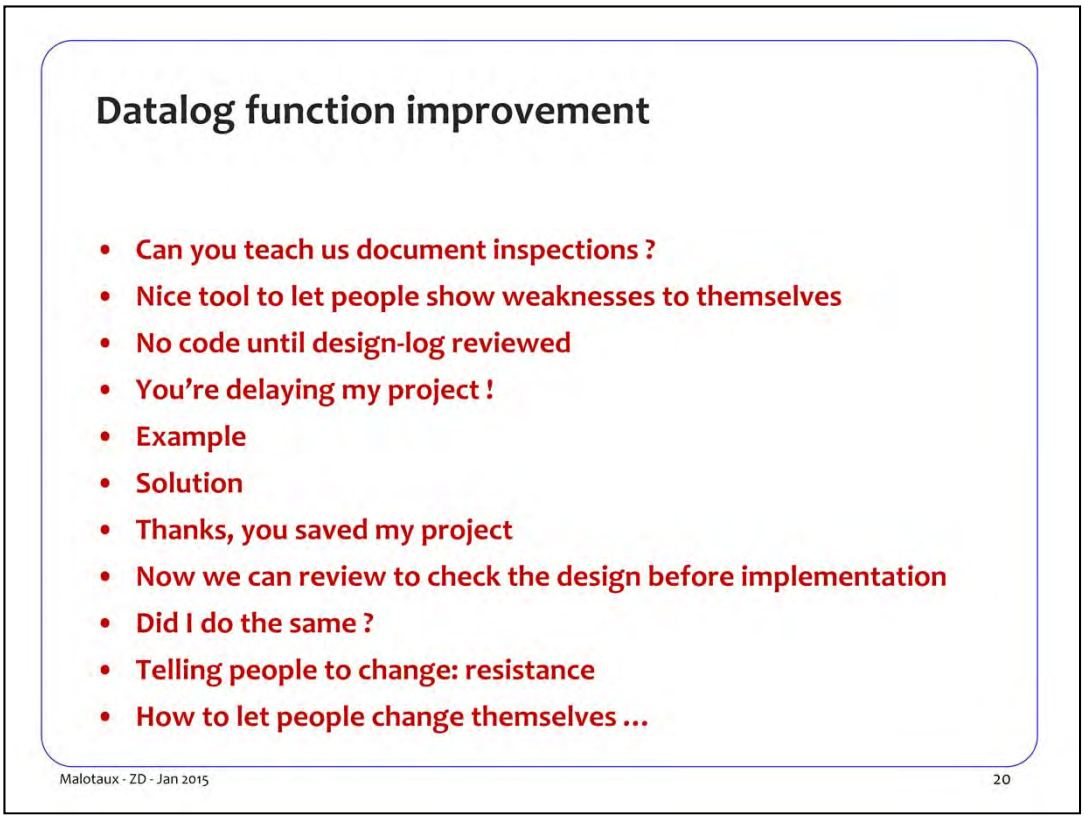
The PC software people and the testers were sick and tired of this trial and error approach, but they couldn't help the firmware guy either.

They were sitting in the same room, but hardly communicated properly

I put them in one room. Asked to make a drawing on the whiteboard that showed the flow of the communication. Within half an hour they found the bottleneck and within a week they solved the communication problem.

Some of my clients complain that I'm expensive as a coach. These people were much more expensive, having wasted so much trial-and-error time.

I call this "Short-Circuiting": if people talk (or complain) *about* each other rather talking *with* each other: put them in one room and let them draw an discuss the design. It always works very quickly. In the beginning you need a moderator, however, to help them to discuss constructively rather than complaining about each other (or worse). Don't talk about what we do wrong. Find out how we can do it right.




I was asked to teach Document Inspections to a group of developers. I gave a short introduction and then we did a baseline review. After all, most people do reviews, don't they?

We selected a design document for some datalog functionality in a controller, took one page out of it, made 20 copies of that page for everyone to review. They started reviewing and after some 10 minutes everyone seemed ready.

I asked about the issues found. Hardly any. Perhaps a typo or two.

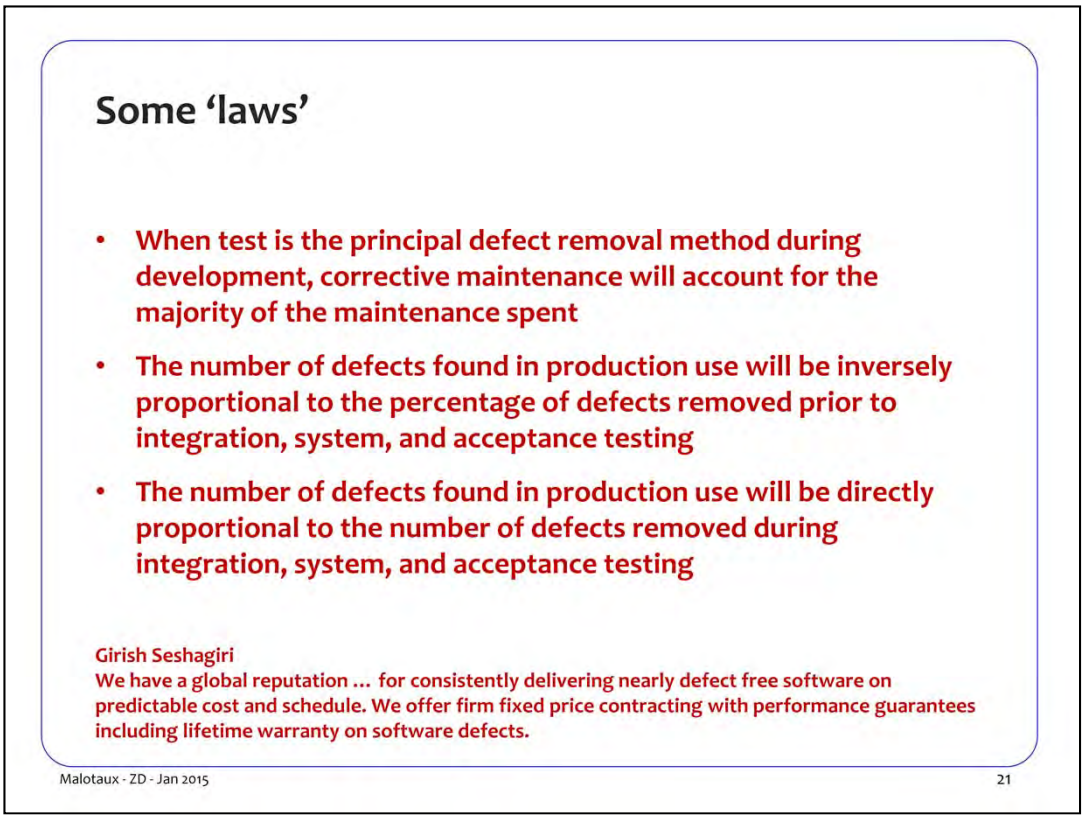
Then I introduced a rule: "If we don't know the requirement of this design, how do we know that the design does what it should do and does not what it should not do?" I asked them to review once more.

After a short time, everyone's paper was full of remarks: This I cannot judge, that I cannot judge, because I don't know what was required and why this is the best solution. With the review they had found that there was a lack of design knowledge.

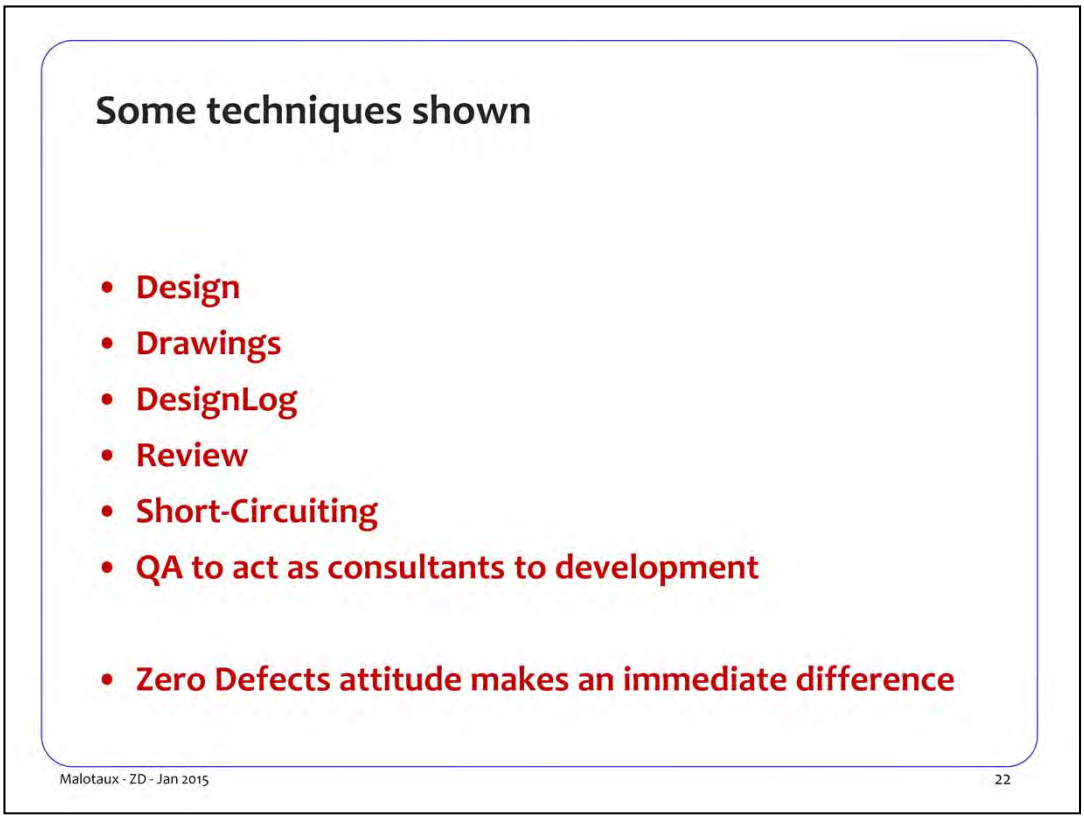
We suggested the designer to make a DesignLog. Not to write even one line of code until the DesignLog was reviewed and found ok.

It took some time until the author understood how to make the DesignLog, but it led to an interesting conclusion: He decided not to implement functionality in the controller firmware, because of some intricacies, which could much better be solved in the PC software at the other side of the network. Imagine what would have happened if he had started coding already. Getting deeper and deeper into trouble, not wanting to stop, because having spent already so much time on coding.

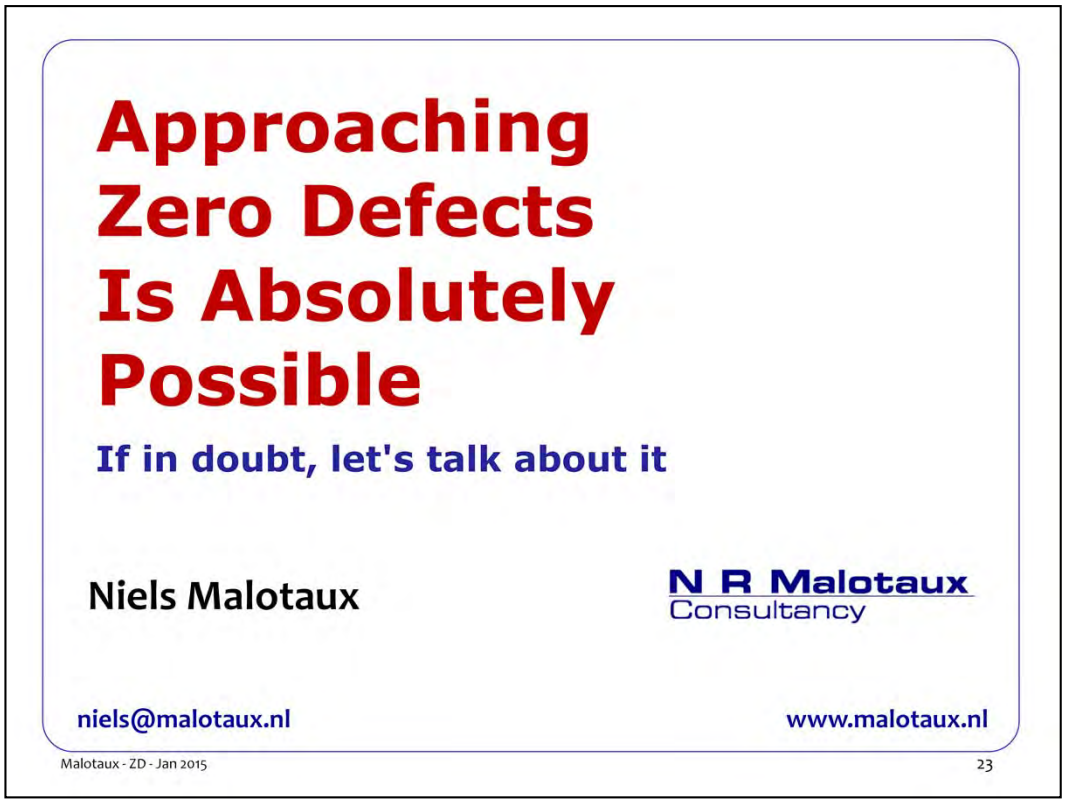
First he complained that I was delaying his project because he had to spend time on design rather than coding. Later he said: Thank you, you saved my project!




Some Laws I read from a guy who is CEO of a company that gives a lifetime warranty on software defects. Would you dare to do the same?




To summarize some of the techniques for ZD. A Zero Defects attitude makes an immediate difference.




In this presentation I didn't only want to bring Zero Defects to you attention. I also wanted to show some examples where people became serious about it in one way or another, illustrating some techniques to help approaching ZD. Note that, as ZD is an asymptote, it's always a matter of approaching it as near as possible. Not to hassle our customers anymore.

If you have any questions or doubts, let’s discuss!


Let's discuss!