Optimizing the Contribution of Testing to Project Success

Niels Malotaux

N R Malotaux Consultancy

+31-30-228 88 68 niels@malotaux.nl www.malotaux.nl

1

Niels Malotaux

Niels Malotaux is an independent Project Coach and expert in optimizing project performance. He has 35 years experience in designing electronic and software systems, at Delft University, in the Dutch Army, at Philips Electronics and 20 years leading his own systems design company. Since 1998 he devotes his expertise to helping projects to deliver Quality On Time: being predictable while delivering what the customer needs, when he needs it, to enable customer success. To this effect, Niels developed an approach for effectively teaching Evolutionary Project Management (Evo) Methods, Requirements Engineering, and Review and Inspection techniques. Since 2001, he taught and coached well over 100 projects in 25+ organizations in the Netherlands, Belgium, China, Germany, India, Ireland, Israel, Japan, Romania, South Africa and the US, which led to a wealth of experience in which approaches work better and which work less in the practice of real projects. He is a frequent speaker at conferences and published several booklets around the topic of the presentation (see www.malotaux.nl/Booklets).

Optimizing the Contribution of Testing to Project Success

Niels Malotaux

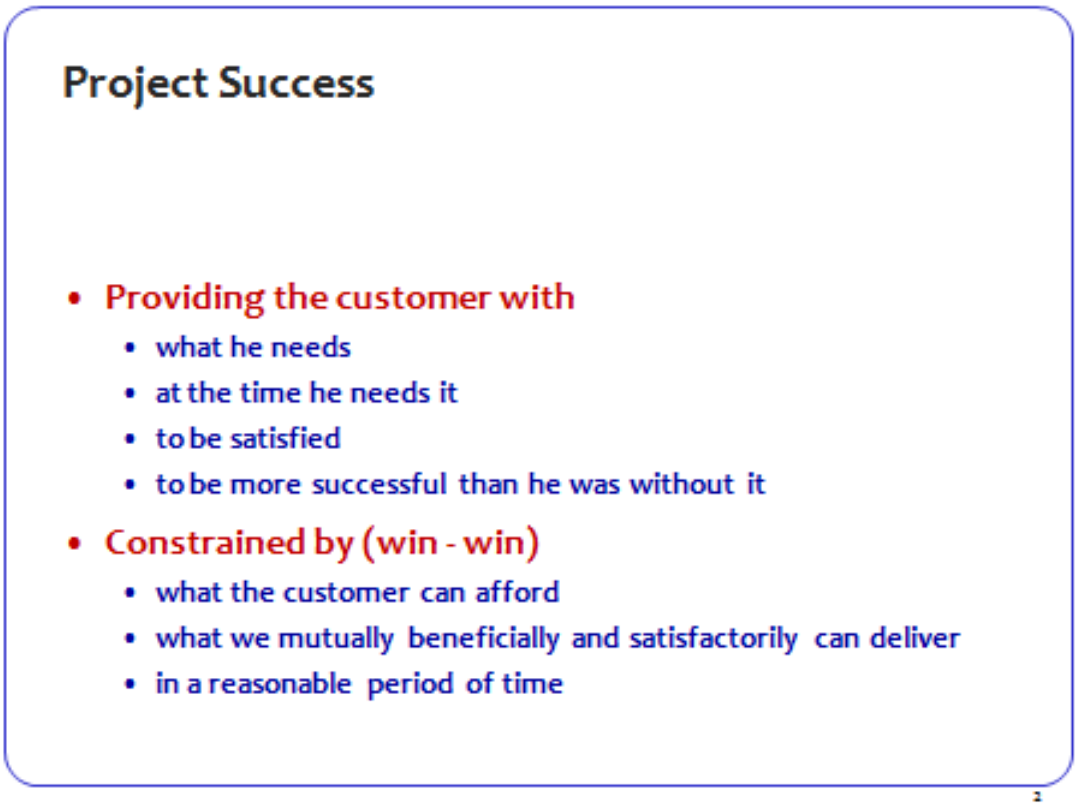
N R Malotaux Consultancy

+31-30-228 88 68 niels@malotaux.nl www.malotaux.nl

1

I'm going to tell a story that a CEO of a test house once called: "Quite philosophical and controversial". He felt that if this were true, he'd get out of work. I assured him that that would be nice but not easily the case and that once there are hardly any bugs left, testing really becomes a challenge, namely to *prove the absence* of bugs (as Dijkstra once said). Still, our customers probably wouldn't mind at all if there would be no bugs any more. The techniques to (asymptotically) come quite near this goal are known, but not much applied.

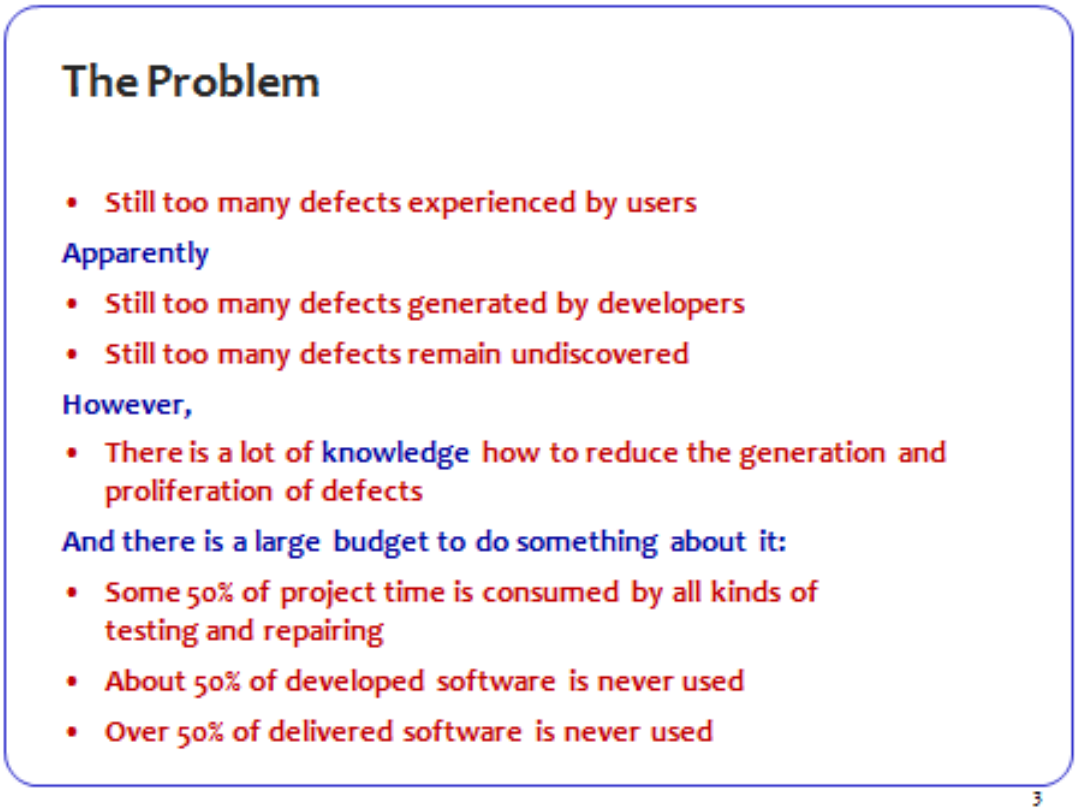
During my talk I expect to hear a lot of "Yes, but…"s. If those people simply deliver flawless software, then I'll keep my mouth shut. But if with their current way of working they do not produce flawless software, it would be better to keep listening, because there is a lot of knowledge how to improve a lot on the current state of software delivery. One reason why this knowledge is ignored is probably that a lot of it is counter-intuitive. Intuition is a very strong mechanism in people, causing improvement not to happen automatically.



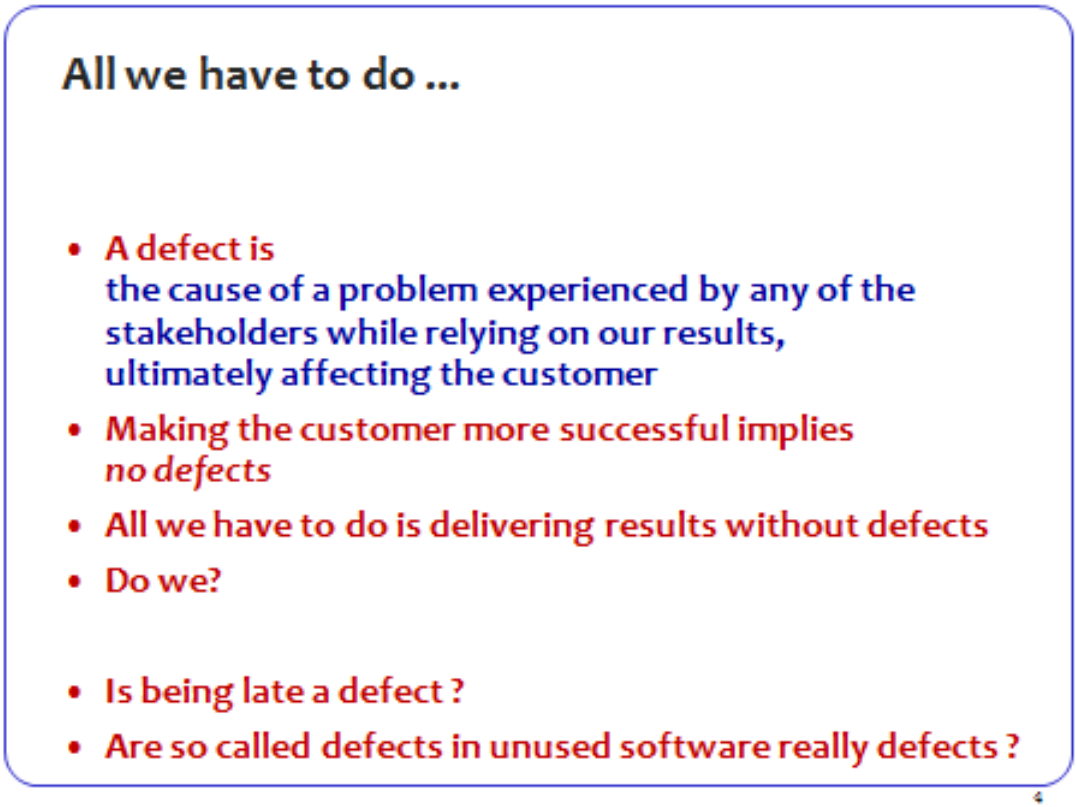
Let's first define the top level requirement of any project:

To provide the customer (usually through users and other stakeholders)

- what he needs (is usually not what he says)
- at the time he needs it (is usually earlier or later that he says)
- to be satisfied (then he *wants* to pay)
- to be more successful than he was without it (if he's not successful, he *cannot* pay; if he's not *more* successful, why *should* he pay)
- what the customer can afford (what the customer asks, he cannot afford; if we try to deliver that, failure is assured)
- what we mutually beneficially and satisfactorily can deliver in a reasonable period of time (it should be win-win)



A colleague in a SPIder working group once laughingly said: "We have a new manager. She said that from now on, she'd expect that whatever we deliver to the business works problem-free for at least the first two weeks of deployment. Ha-ha, what a joke!"
I replied: "Finally a manager who knows how to set requirements! I think this is a normal requirement that can very well guide what we are supposed to do."
This shows the difference between the prevailing attitude in software development and testing, and what I want to tell you in this presentation.



Short definition: A defect is the cause of a problem for the users
If we cause a problem by being late, it is a defect (by the above definition)
If the software isn't being used (over 50% of delivered software), the defects in that part of the software aren't defects according to this definition. The only defect is the fact that that part of the software was made in the first place.
This urges us to determine what software we are going to make that eventually won't be actually used, so that we can refrain from making it, saving a lot of time. Whether that's easy or not is beside the point.

The process of defect injection

Conventional software development:

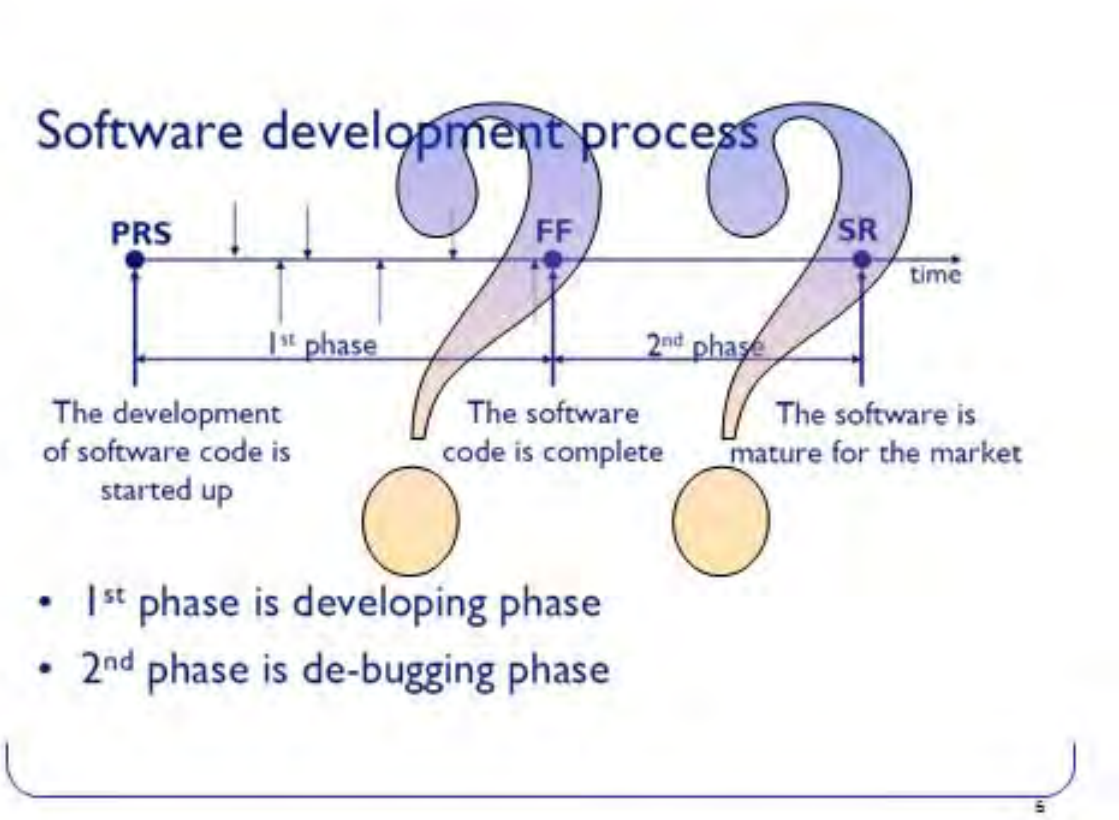
1. Development phase: inject bugs
2. Debugging or Testing phase: find bugs and fix bugs

Can't we do better ?

Does anybody mind ?

5

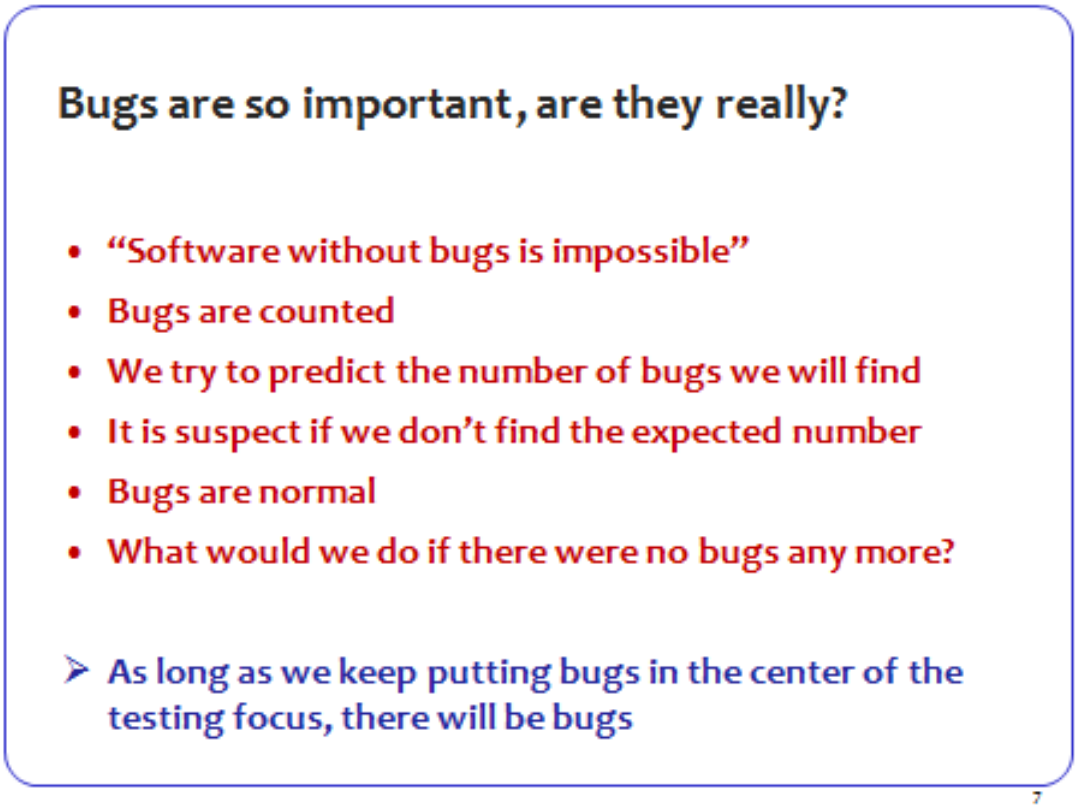
This is a situation we see in so many organizations. Awful, once you know how bad this is. But if nobody minds, there is not much we can do about it. Still, once you know how bad this is, especially because there is so much knowledge how to improve on this, how dare you not do something about this and still expect a salary!



A University PhD student showed this picture as being the official development process at a well known large company in Holland. I've seen a similar picture in a presentation from a well known large software company in the US.

The 2nd phase usually takes 50(±30)% of the total time. How can we call it "Code Complete" if it's full of bugs?

This is a very bad and costly process. However, because it's so widely practiced, many people think that this is how it should be. They should know better. Probably deficiency of the educational system, because the solution is known for decades.



Bug and *debug* are dirty words, to be scratched from our dictionary. If you want to know how to do that, we can talk about it.
Exaggerating the significance of bugs conveys a very bad message to the developers, namely that bugs are expected and that it's normal to produce bugs. However, if the customer shouldn't find bugs, our goal should be to prevent bugs, not to count them. (There is some reason to do some counting but that's another story and, at least the psychological effect of the counting should be recognized and adequately handled)

Defects found are symptoms of deeper lying problems

Repairing apparent defects creates several risks:

- Repair is done under pressure
- We think the problem is solved
- We introduce scars
- After finding the real cause, the redesign may make the repair redundant: time lost
- We keep repeating the same problems

8

The first effect of finding issues should be feedback to development to feed the prevention process. Repairing bugs found is only a secondary goal. After all, testing is always taking a sample (even if we could check all possible paths through the software, we cannot do this with all combinations of data, therefore it will always be a sample!). If we take a sample and repair the defects we happen to have found in that sample, the issues outside of the sample are still there. Besides, repairing issues does usually add other issues. This implies that the quality level of the software is hardly an order of magnitude improved by the results of testing, so what's the point of repairing those issues we happen to have found?
Example: 100 issues in the software, 50 found, 10 inappropriately "repaired". Result: 60 still there.

Testing is very expensive

- You can prove the existence of a defect (if you found one)
- You cannot prove the absence of defects (if you didn't find any)

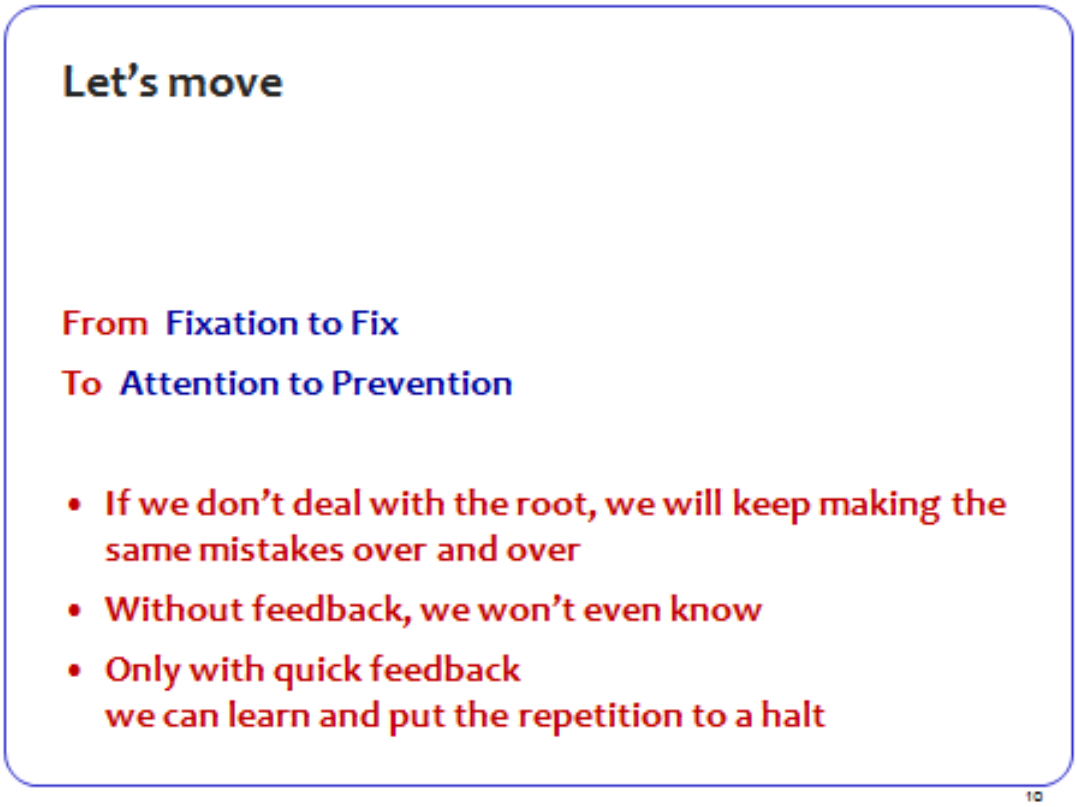
- Proving the absence of defects is difficult
- Proving the existence of defects is also difficult

- Why do we put so much emphasis on finding defects?
- While what we want is *no defects*

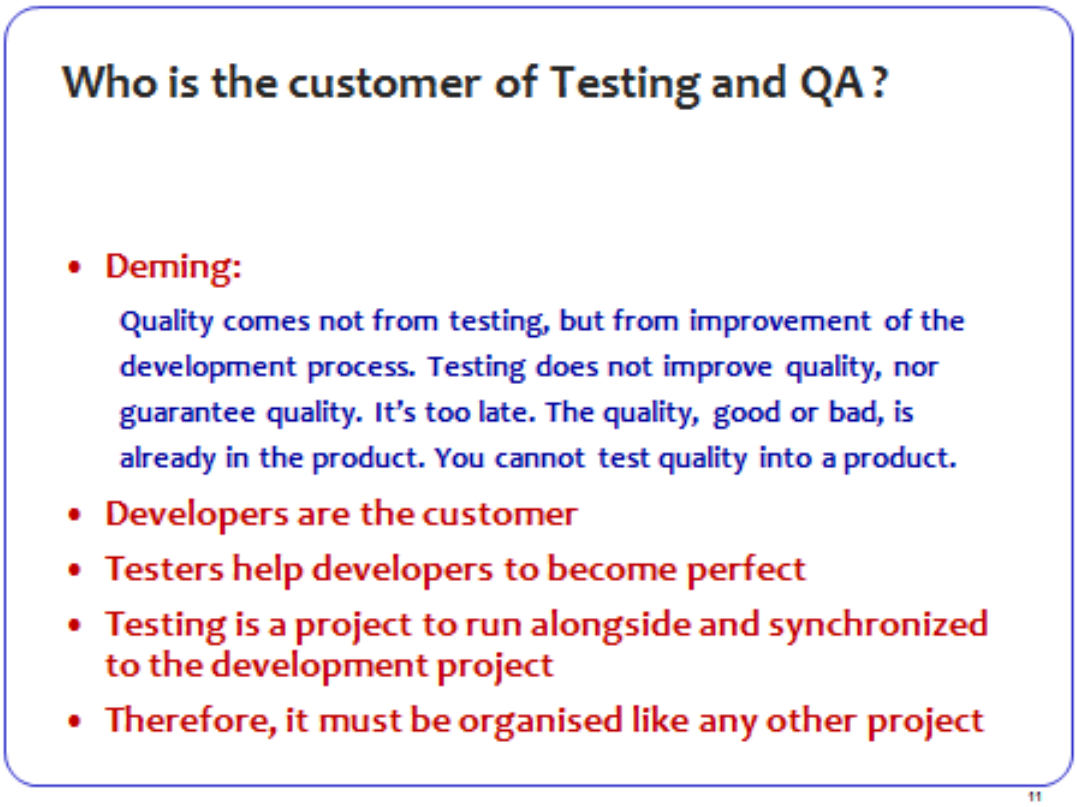
- Testers should learn better how to prove the absence of defects while
- The developers should learn better how to avoid defects
- Testers can help, showing which types of defects are still made

9

Dijkstra:
Testing can show the existence of defects, but it is *highly inadequate* to show their *absence*.
Note: *No defects* is *cheaper* than first producing defects, then trying to find them (we find only about half) and to fix them (fixing often uncovers more defects). Crosby wrote a book "Quality is free". I know (by my own experience and because of what others did) that Quality is *cheaper*.
One problem is that most people don't believe this is true. Therefore they don't even try to improve.



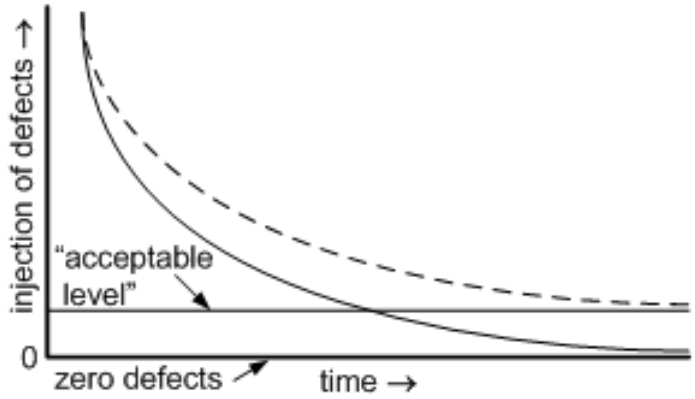
Root cause analysis is the name of the game.
Said a Project Manager "Should we then do root cause analysis with ALL bugs found?"
My answer: "Of course!"
PM answer: "Impossible, we don't have time for that".
Remember the Toyota principle of "Stop the line": Initially, no cars were coming of the production line. However, after some time ONLY GOOD cars were being produced. In contrast, US car manufacturers kept producing errors, which had to be repaired afterwards at high cost.



I experienced that to most testers this 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.

Is defect free software possible?

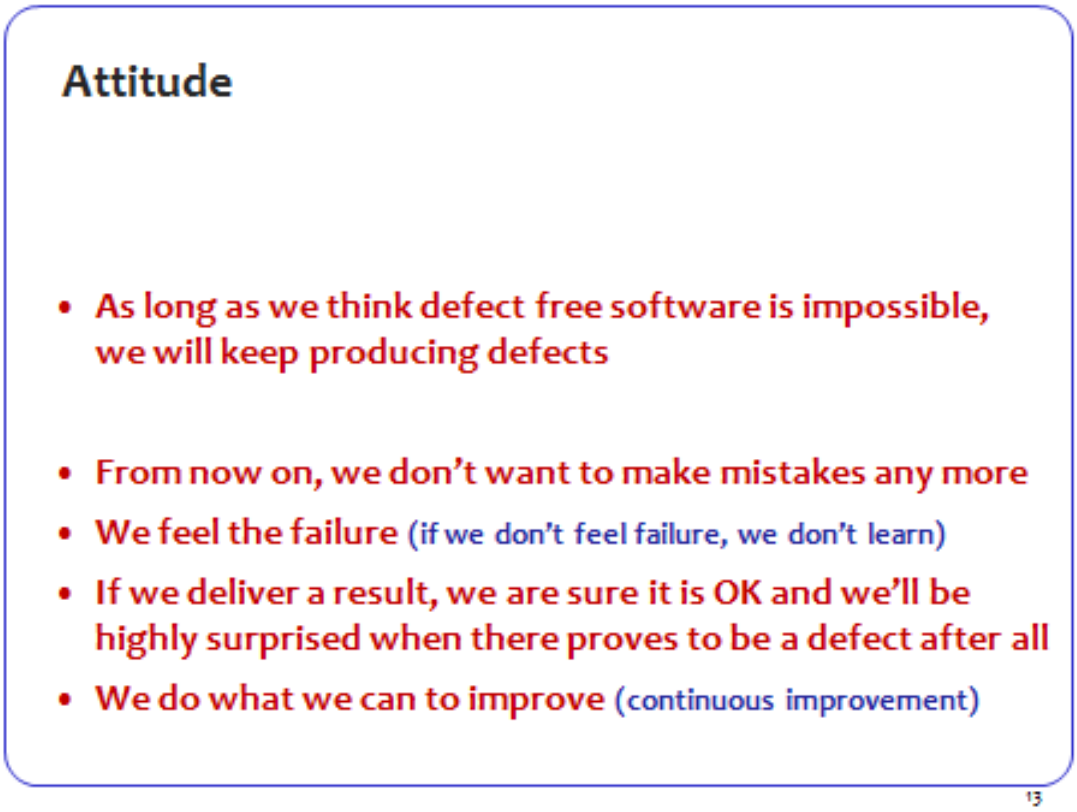
- Zero Defects is an asymptote



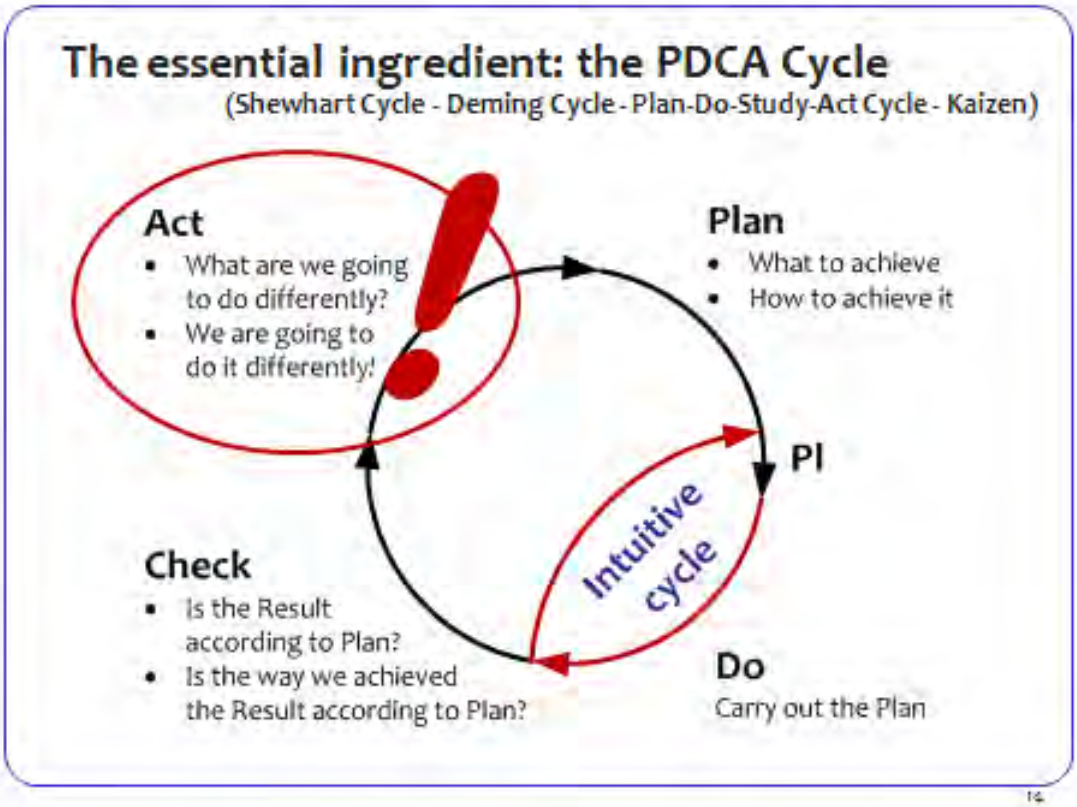
- When Philip Crosby started with Zero Defects in 1961, errors dropped by 40% almost immediately

12

When I actively started using the Zero Defects paradigm in software projects, defects made were reduced 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 errors anymore!" I said: "Keep up the good work. 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 half of the project is spent on finding and fixing defects. That's a huge budget. Any savings on that is probably well worth trying.



Zero Defects isn't an absolute. It doesn't mean that just by adopting ZD we suddenly don't make mistakes any more. People make mistakes and we are people, so if we've done something, probably there will be defects. But once we recognize and admit that, there is a lot we can do about it.
This applies to developers. It applies to testers as well. To continuously improve what they do (their product/goal), how they do it (their project) and how they organize it (the process).

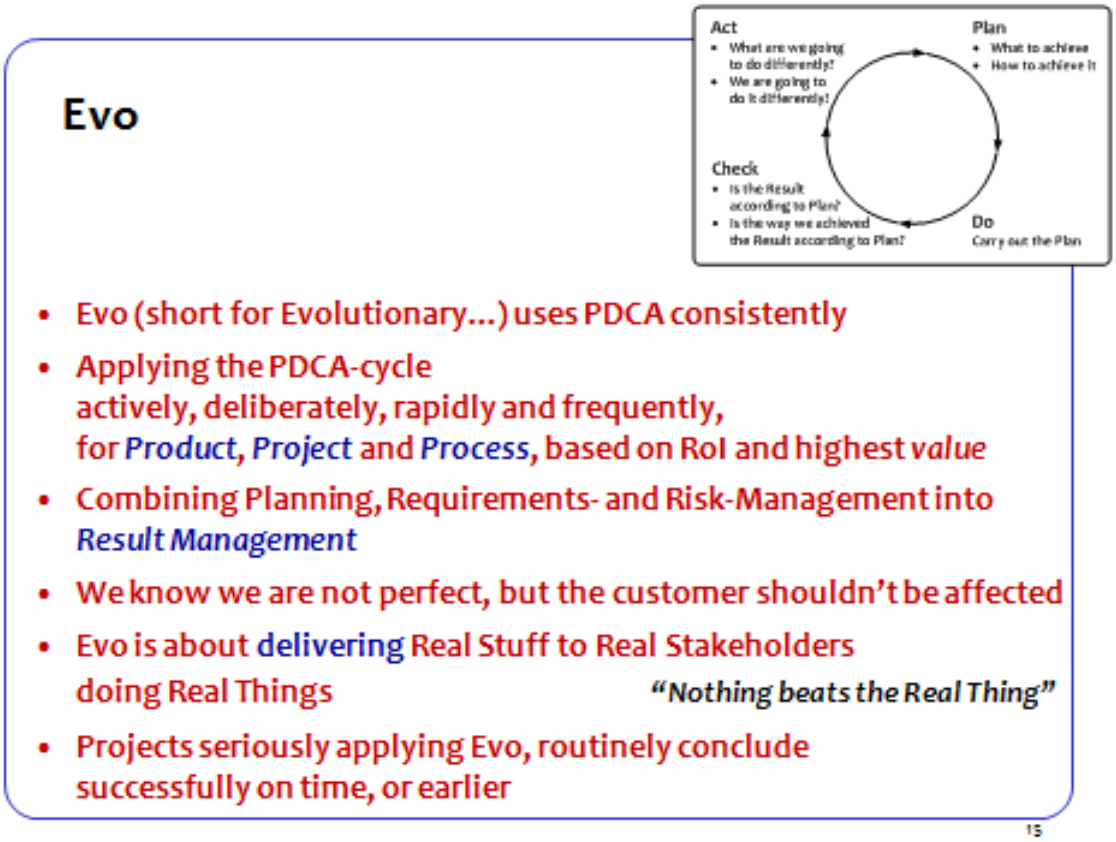


The essential technique for continuous improvement is the Deming or Plan-Do-Check-Act cycle. We Do all the time, Planning we do more or less, usually less and for Check and Act we don't have time.

Many people think they know the Deming cycle, but let's see how it really starts working for us. The intuitive cycle, how we normally work, is the Pl-Do-cycle. I can't call it Plan, so I call it only Pl. "What was the next thing we are supposed to do?" and we are already doing it. If intuition would be perfect, everything would be perfect. Not everything we do is perfect, so apparently our intuition sometimes points us into the wrong direction.

So, let's first Plan what Result we want to achieve and how we think we can most efficiently achieve that (Planning is twofold: the product and the project). Then we Do according to the Plan. This is the first pitfall: the Plan must be doable and we must follow the Plan. Let's assume we did that, then in the Check phase we can Check (Deming also called it Study phase) whether the Result was according to Plan. If it was according to the Plan, we can think: "Can we do it even better the next time?" If it wasn't according to Plan, we can think: "How can we do it better the next time?" Then comes the Act phase: "What are we going to do differently the next time, because if we don't do anything differently, the result will be the same. If we want to improve we have to decide to do something differently, then Plan and Do accordingly and then Check whether the change actually was an improvement. If yes, can we do it better the next time. If not, can we do it better the next time. In the Act phase we introduce a "mutation" in our way of working, hence we call it the "Evolutionary" approach.

This way, we are continuously improving on the Result (the product), the way we realize the Result (the project) and even how we organize all of this (the process). Actually we can stop now, because using the PDCA technique, you can start from scratch and very quickly find out how to continuously do things better. Because we have been doing this already for a long time, we can save you time and give you a flying start.



What is Evo

- **Short for Evolutionary Development/Delivery/Project Management**
  Evo is a label we use for successful methods to deliver Quality On Time. Until now all the successful methods have an Evolutionary aspect in them. So we use the Evolutionary label. In short: Evo.
- **Deliberately going through the PDCA cycle rapidly and frequently, for product, project and process**
  Plan - Do - Check - Act cycle, also called the Shewhart cycle or Deming cycle. Do is what we normally do. Most of us Plan, more, or less. Usually we "have no time" for the Check and Act parts. We use this cycle on everything: the Product (what is really needed *and* possible within the budget), the Project (how to learn to do things better) and even the Process: what doesn't work is discarded: no burocracy.
- **Continuously thinking *what to do*, in *which order*, to *which level of detail for now***
  What we have done until this very moment cannot be changed any more. What we have, we have. What we haven't, we haven't. What we thought last week what we should do does not matter. Based on what we know NOW: What is the best to do NOW, in which order (priority!) to which level of detail for now, because if we do more detail than is necessary NOW, we will have wasted time if we later find out that we should have done something different.
- **Methods for efficiently and effectively running development projects, delivering Quality On Time**
  Evo projects deliver routinely Quality On Time.
- **Delivering what the user needs at the time he needs it**
  That is what pays our salary

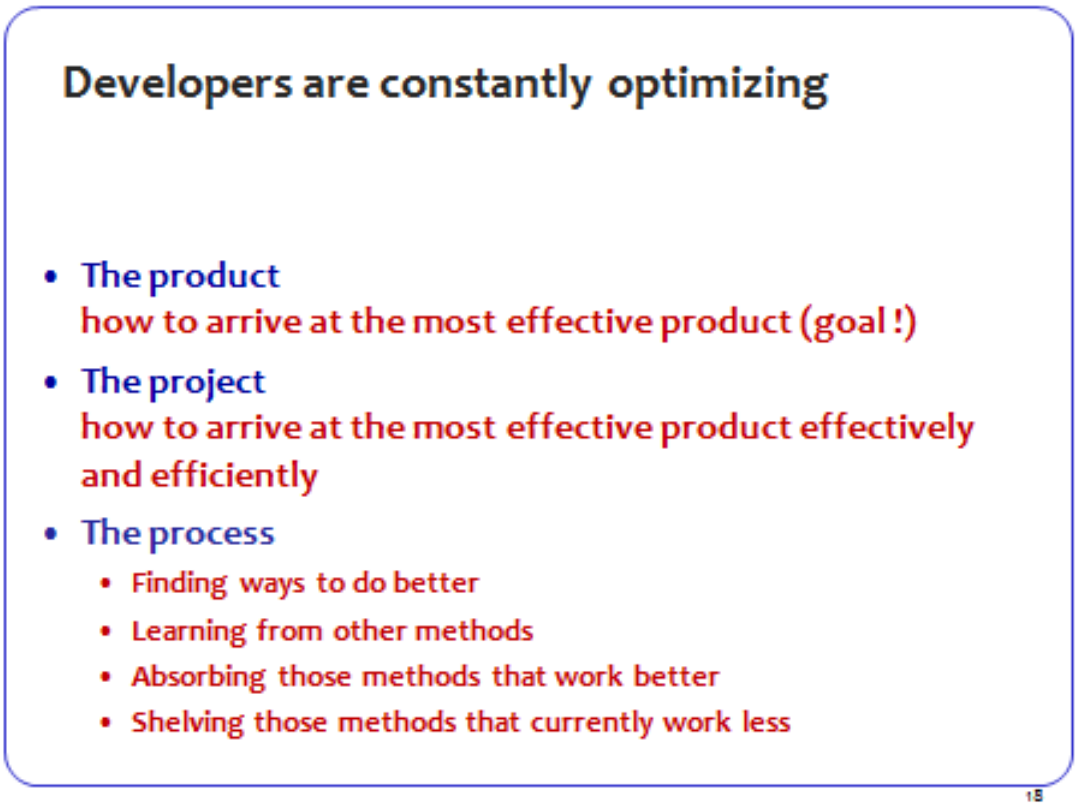
Based on continuously applying the PDCA cycle, we continuously improve. This way we could start from scratch and quickly find the "best" way to do things. However, we can make a flying start if we start with what others already found out and keep improving from there.
This way, "Evo" is a label covering the "best" way of doing things, as far as we know. As soon as we see a better way, we'll Check that way, decide what and how to use it (Act), apply it to our Planning, Do accordingly and then Check whether it actually worked better. If it worked better than how we did it before, we keep the better way.
The following elements have crystallized so far: see slide. Because of the limited time I cannot dwell on all of these much.
Business Case defines why we are doing what we do. It's about RoI. Did you define the Business Case of your current testing project? Can you imagine that your testing work can have a Business Case?
Requirements engineering the Evo way is different from conventional RE: we employ a requirements description language everybody can easily understand. We define "Real" Requirements. We don't just decide what we are supposed to realize, but also how much and what not. For example, for testing, a Requirement could be in the form: "Number of defects produced by development; Now: 13 per kLoC [project x, 21 April 2010], Goal: 6 per kLoC [project x, 1 Oct 2010). I immediately hear testers think "How can we be made responsible for the improvement of the developers?!" We can, but in this presentation unfortunately I don't have enough time to elaborate on that.
The Evo Design process is about finding the "best" compromise between the conflicting requirements. Note that there are always requirements in conflict with other requirements. Think about more performance vs. budget (time/cost). In order to be able to find the best compromise, requirements should not be stated as point requirements, but rather as range requirements (between MUST and GOAL) so that there is *room* for compromise.
Evolutionary Project Planning basically has to do with the notion that we never have enough time to do all we *think* we have to do (proof: most projects are late). Evo projects are not late and the Evo planning techniques help projects how to achieve that.

Conventional Evo Testing

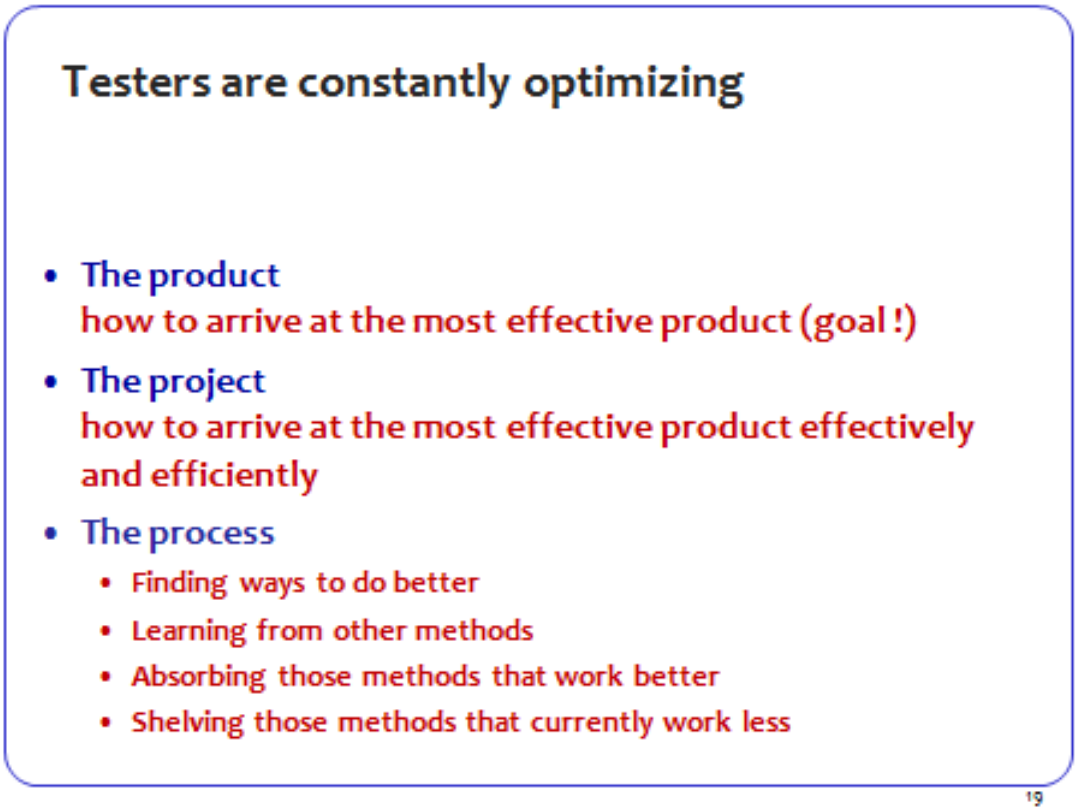
- Final validation shouldn't find any problems
- Earlier verifications mirror quality level to developers: how far from goal and what still to learn
- Evo has *no debugging phase*!

17

Developers are constantly improving (well, at least in the projects I coach)



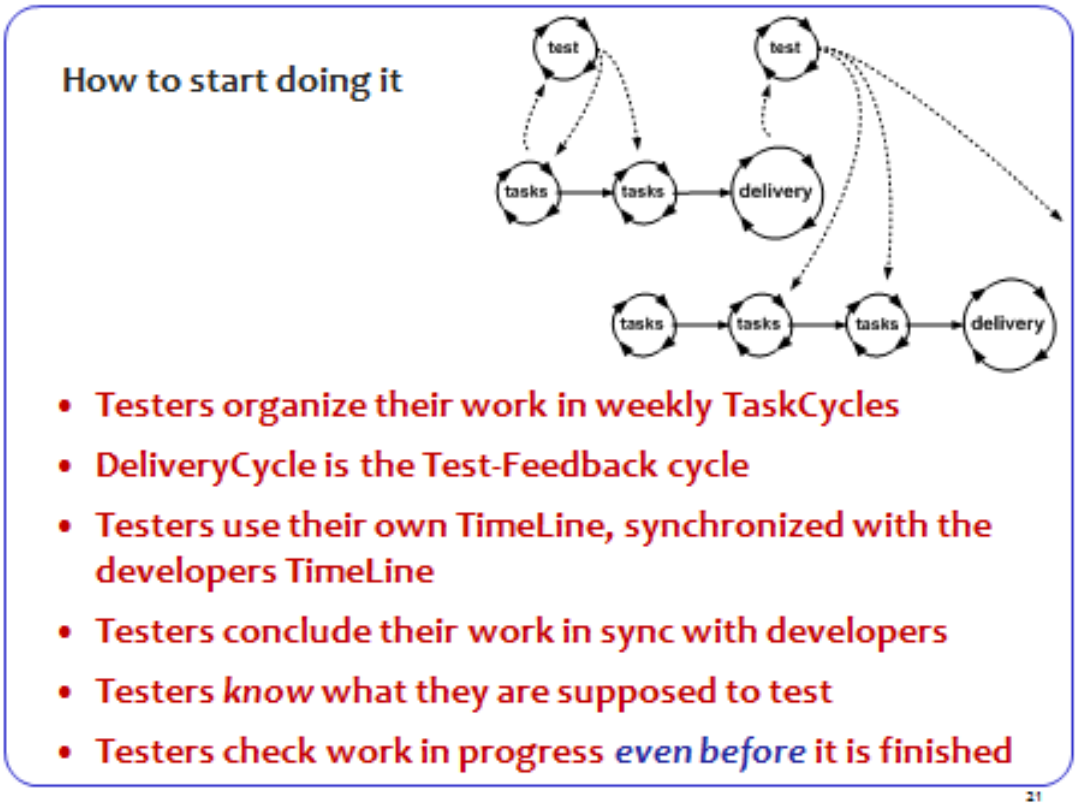
Testing is *checking that it works*. Because Testing statistically only finds about 50% of all defects, the customer will find the other 50%. If you want the customer to find no defects, the system should be without defects *before* the final test.
In Evo, with frequent deliveries, we can regularly ask the testers to tell us "How far are we from defect free delivery?" If the testers tell us what we still are doing wrong, we can learn to prevent injecting defects during the project.
To the developers I regularly say: "Let's starve the testers!" Testers, don't despair! There will still be a lot of testing to be done.
Evo projects have no *debugging phase*.
Note: *Debugging* means finding and fixing *Bugs*. Bugs are defects in the product, caused by errors that the developers have made. After injection, we have to find them, do root cause analysis to feed the prevention process and we may fix the issues found, as well as similar issues that we now can assume are lurking in the remainder of the software. Because we are humans, and humans make mistakes, it is probable that we make some mistakes. However, we can learn to avoid most of these mistakes, if we use rapid and frequent feedback for learning. The words *debug*, *debugging* and *bug* are well known words in software. To me these words should be erased from our dictionaries, because these words are hardly necessary, if we work well. I know that by experience in many projects.



Testers are also constantly improving (well, at least in the projects I coach). Remember what the product of the Testers is! Once the testers realize that Development is their main customer, they can focus the goal of their testing project accordingly and correctly.
The testing project should be organized in parallel with the development project.

Further Improvement

- Tester's customer is "the developers"
- Finding defects is not the goal
- Project Success is
- Testers select and use any method appropriate
- Testers check work in progress *before* it is finished
- Testers solve the Review and Inspection organizing problem
- Testing is organized the Evo way, entangling intimately with the development process



Should we allow developers to inject *all* the errors they will be injecting? Remember: people make mistakes, developers are people, therefore, while they are developing they are injecting defects. Better get the things they are developing from under their hands while they are still busy with it. Quickly feedback the tendencies of defect injection, so that they can repair what they did, and prevent injecting similar issues in the remainder. This is prevention at work. We call this Early Review or Early Inspection.

The aim of Testing

- Being done as soon as the development is done
- Well, almost

- Excuses, excuses, excuses
  - The developers are always late
    (Evo developers live up to their promises)
  - The developers don't take us seriously
    (Evo developers ask testers for help)
  - The developers don't inject enough defects
    (now testing becomes a challenge)
- Helping development to be successful

11

Don't let the product rot on the shelf when it is ready, only because testing is still testing. It is quite possible to have testing be done almost immediately after the final delivery by development.
First people must understand that this is important and possible. Then we can teach them how to do it.
I use the "Bullshit Sticker" when I hear unnecessary excuses. Real professionals know how to handle these issues and hence don't need the excuses. If people don't yet know how to handle issues that *happen in every project*, we call them apprentices or juniors.
I hope that I have put some ideas in your mind to rethink the purpose of testing and that with the principles I mentioned (but unfortunately didn't have enough time to explain more thoroughly) you can improve the contribution of testing to project success. After all, only project success really pays our salaries.

Optimizing the Contribution of Testing to Project Success

Niels Malotaux

N R Malotaux Consultancy

+31-30-228 88 68 niels@malotaux.nl www.malotaux.nl

Questions or comments? Send me an e-mail.