Niels Malotaux

Help ! We have a QA Problem !

Niels Malotaux

Help ! We have a QA Problem !

1 The problem

I got a phone call from a R&D manager: "We have a QA problem! Can you help?" In most cases this means that they think they have a testing problem, and this case was not different: One senior tester just had left the company because he had complained about his salary, and the remaining senior tester was starting to complain as well. This may be difficult for others to understand, but engineers in general like their work, and if they start complaining about the salary, something is very wrong in the organisation. The senior tester, with only one junior tester to assist, was paralysed by the pile of work in front of him. Some 15 developers producing hardware, firmware and software caused the pile to grow faster than the remaining testers could handle. Customers were waiting too long for solutions to their problems, becoming really impatient, and started abandoning this supplier in favour of the competition. As often is the case, the testers were blamed for the delay in deliveries to the customers.

1.1 What did we do about it

Switching on the LCD projector, using Excel as a structured notepad, we started analysing the extent of the problem, listing the work-packages waiting in the pile. I asked the senior tester to estimate the number of days he would need to complete the required testing of all the packages in the pile, focusing on his part of the work being the bottleneck. We added up all his estimates and arrived at 106 days of work (Table 1).

This would mean that some customers would have to wait for about half a year before getting the solution to their problem, while during this time the developers would produce an even bigger pile, worsening the situation even further. This was clearly unacceptable. Indeed, there was a problem! The tester was sitting there, feeling not happy at all. Instead of complaining about a problem, we'd better do something about it. So, this is what we did:

Line	Activity	Estim	Alternative	Junior tester	Develop	Customer	Will be done (now=22Feb)
1	Package 1	17	2	17	4	HT	
2	Package 2	8	5		10	Chrt	
3	Package 3	14	7	5	4	BMC	
4	Package 4 (wait for feedback)	11				McC?	
5	Package 5	9	3		5	Ast	
6	Package 6	17	3	10	10	?	
7	Package 7	4	1		3	Cli	
8	Package 8.1	1	1			Sev	
9	Package 8.2	1	1			?	
10	Package 8.3	1	1			Chrt	24 Feb
11	Package 8.4	1	1			Chrt	
12	Package 8.5	1.1	1.1			Yet	28 Feb
13	Package 8.6	3	3			Yet	24 Mar
14	Package 8.7	0.1	0.1			Cli	After 8.5 OK
15	Package 8.8	18	18			Ast	
	totals	106	47	32	36		

Table 1: Slightly simplified and anonymised image of the actual spreadsheet how we deal with the "QA problem". Objectifying and quantifying the problem is a first step to the solution.

- We made it clear to the senior tester that he still had the responsibility to sign-off for delivery to a customer, only if he was sure that the customer would be made happy with the delivery. No dilution of quality!
- We decided that the developers were to stop developing, and that 'the whole company', especially the developers would be at the tester's disposal, as necessary. If he'd need the CEO to do anything for him, we would make the CEO available
- We asked the senior tester to imagine what the developers could do for him, like test automation, making test scripts, testing or whatever. The aim was to relieve the senior tester, being the bottleneck, from as much work as possible. He would still have to oversee the work of the others, making sure that they would be doing the right things, and checking their results
- We now asked him to estimate again: how much time would he need for the various packages and how much time did he estimate the developers would need (not to make the developers a bottleneck)

Adding up his estimates showed that he still would need 47 days, or about 10 weeks.

1.2 Some refinement

Until now, we had only worked with work-packages of about 10 days each. As an example for more detailed planning, I asked which package had the most pressing customers waiting. We split this package into smaller elements, estimated these elements, and listed which customer was waiting for which components of this package (Table 1, Package 8).

The table shows the (slightly simplified) spreadsheet that emerged, the numbers being real, but the actual names of the packages and of the customers anonymised. Note that, strange as it may sound, the exactness and even the correctness of all of these numbers is not important at this stage: Adding numbers averages out variance, and 0^{th} order approximation (ballpark figures) is usually sufficient for decision making. If more detail or 'exactness' doesn't yield a better decision, we shouldn't waste time on the extra detail. The actualisation of the numbers happened in the subsequent weekly plannings.

1.3 Planning and result

Now we could start planning what to do in which order, systematically making customers happy, one by one. Note that we don't have to provide every customer with their full solution immediately. After all, customers need time to digest what they get, so we could plan to dose component by component to selected customers in a regular fashion, based on customers' real needs.

The basic plan I showed, was bi-weekly deliveries as shown in Figure 1. Within two weeks, one customer could be made happy. Two weeks later, two more customers, and so on. In reality, customers were made happy even faster, because useful test results came out much more often. Based on our planning, we would send the customers a message: "We'll have your solution at that date. Will you be ready for it?", checking the eagerness and preparedness of the customer for the delivery. We were optimising our delivery process, and if customers were not activated appropriately at the same time, our improvements would not make much sense.

The senior tester started to plan all the other packages in some more detail in a similar fashion as we did in the example, putting them on the timeline while synchronising with the developers for their share, and customers for their acceptance ability, aiming at optimum customer satisfaction. They started based on this plan, and 9 weeks later the pile was gone. Customers were amazed about the change, got more confident of our capabilities, and started ordering more products. One year later people told me that sales had increased by 70%.

The senior tester felt empowered and revived. He kept planning the testing activities in the same fashion ever since, now making sure that the testers kept up with development. Two years later, he got promoted to the position of product manager, still coaching his successors in the planning technique. An interesting by-product of the exercise was that the developers, having actually been involved with testing, now were much more aware to improve the testability of their fruits of work.

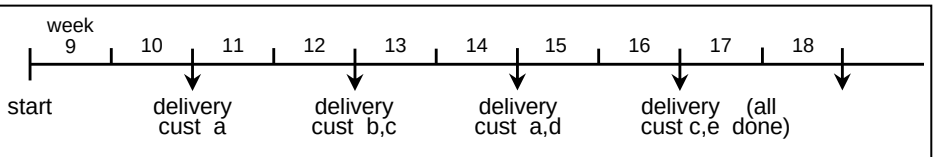

Figure 1: Basic idea of the TimeLine plan, later detailed into more Deliveries

2 What did we do?

In order to achieve the result described, we used what we call Evolutionary Planning techniques. The Evolutionary technique is based on constant improvement of whatever we do, using the Plan-Do-Check-Act or Deming cycle. In the Plan phase we decide what we should achieve, and how we most efficiently and effectively will achieve it. In the Do phase we follow the plan. In the Check phase we check whether the result was as planned, and whether the way we achieved the result was as planned. If yes, we think how we can do it better the next time. If no, we think how we can do it better the next time. The Act phase is the crucial and mostly forgotten one: deciding what to do differently the next time, because if we keep doing things the same way, the result will *not* be different, let alone better. By creating *mutations* in how we do things, we *provoke* evolution. And because as humans we can imagine the impact of the changes we introduce, we can move the evolution quickly to improvement, rather than random change. In Evolutionary Planning, we currently use (note that the process is also evolutionary, so it may change based on evolving experience!) [Mal09], [Mal10]:

- TaskCycles to organize the work and to continuously improve the way we spend our time
- DeliveryCycles to deliver to stakeholders either to make them happy early, or to find out what will make them happy. This is to check the (perceived) requirements and the assumptions, many of which are often incorrect. In the DeliveryCycle we aim to get feedback to find out whether we are on the right track to success, and to find out as quickly as possible when we are not on the right track. This way, we have to redo as little as possible, wasting as little time as possible
- TimeLine to get and keep control over longer periods of time: predicting what will happen if we don't change our ways and to find alternative strategies to do better things, and to do things better

2.1 TimeLine

In the case of the "QA problem" we started doing a Check phase, first studying the current situation and what would happen if we just would continue unaltered. We made a list of what we thought we had to do, and made rough estimates (in this case activities between 5 to 15 days). Before we started, the testers and their manager had a *feeling* that there was a lot of work to do, more than they could handle in an acceptable period of time. Once we quantified the problem, we *knew* (*sufficiently* accurately) how much work there was, showing the nature and the size of the bottleneck and not liking what we saw. We realised that going on unaltered was an unacceptable option. We had to do something differently, in this case using the developers as a temporary extension of the testing department. We quantified this scenario and arrived at a much more acceptable strategy.

Summarizing the TimeLine technique:

- Cutting what we think we have to do into up to 20 chunks (packages, activities) and estimating these chunks. Adding up the estimates usually provides sufficient evidence that we need more time than we have available. At this point, most projects decide that they simply need more time, or complain that management is imposing impossible deadlines
- With Evolutionary Planning, however, we don't stop here, but think of alternative strategies of doing things, doing different things, or doing things differently. We estimate the impact on the result and choose the optimum strategy. Now we have well-founded arguments to explain management why things will take as much as they still will do
- Now the chosen strategy is planned, focused on the optimum order of implementing the optimum solution, still being aware that "optimum" gradually may change by advancing understanding. It's of no use continuing an initial plan once we see that it should be changed. That's why we have to continuously keep using the Plan-Do-Check-Act technique, with the Business Case as a reference
- Now we can start predicting what will be done when, based on the estimates and subsequent calibration to reality. This provides the business with quite reliable predictions, allowing them to provide reliable predictions to their customers

Table 2 shows a simplified example of a TimeLine table, stating the Activity-description, the estimate, the time already spent and the time still to spend, the ratio of real and estimated time, the calibration factor (ratio of total real time and estimated time during a past period), the resulting calibrated ('real') time still to spend and the resulting dates. If in this example the project has to be concluded on 5 June, we now can say that Activities 17 and 18 won't be done at that deadline, unless we do

something differently. This way, we can very early in a project predict what will be done when, and take the consequence of the prediction, rather than sticking our head in the sand until reality hits us somewhere.

3 What does all this have to do with Testing or QA?

Just like development, testing can also improve productivity enormously by using Evolutionary Planning techniques. Testers often complain that at the end of the project they don't get enough time to do proper testing, the developers always being late and the end-date never being adjusted, squeezing the remaining time available for testing. Just like in the above example, testers shouldn't complain about this, but rather think what they can do about it. The solution is simple: don't wait until the end to start with testing, but start testing right from the start. Review the business case, review the requirements, review the architecture and design, review whatever code is being produced as the project progresses, all the time providing quick feedback to the developers, so that the developers can repair the mistakes already made, and learn from them to prevent making these and similar mistakes anymore, saving a lot of time. This way, testing needs hardly any extra time after the developers have finished, minimizing the delay because of testing.

3.1 Who is the customer of Testing and QA?

Deming [Dem86] explained (slightly modified for testing):

"*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.*"

Once we understand this, it's inevitable to recognize that the main customer of QA and of the testers is *development*. For most testers, this is quite a paradigm shift!

The developers are to put the right quality into the product. If the developers are humble enough to admit, that, just like other people, they make mistakes, they can ask the testers to help them finding out where they are still making mistakes, in order to learn how to prevent making these mistakes ever more. The testers of course keep trying to find the remaining mistakes, because feeding these back to development leads to ever better results.

If we recognize that testing should run *along with* development, where the developers are the customer, and the customer has to be supplied with *what they need, at the time they need it, to be*

Line	*Activity*	Estim	Spent	Still to spend	Ratio real/es	Calibr factor	Calibr still to	Date done
1	*Activity 1*	2	2	0	1.0			
2	*Activity 2*	5	5	1	1.2	1.0	1	30 Mar 2009
3	*Activity 3*	1	3	0	3.0			
4	*Activity 4*	2	3	2	2.5	1.0	2	1 Apr 2009
5	*Activity 5*	5	4	1	1.0	1.0	1	2 Apr 2009
6	*Activity 6*	3				1.4	4.2	9 Apr 2009
7	*Activity 7*	1				1.4	1.4	10 Apr 2009
8	*Activity 8*	3				1.4	4.2	16 Apr 2009
↓	↓							
16	*Activity 16*	4				1.4	5.6	2 Jun 2009
17	*Activity 17*	5				1.4	7.0	11 Jun 2009
18	*Activity 18*	7				1.4	9.8	25 Jun 2009

Calibration factor:

$$\frac{\text{Spent+StillToSpend}}{\text{Estimated}} = \frac{21}{15} = 1.4$$

Table 2: A simplified TimeLine sheet, indicating what will be done when based on estimates and a calibrated future. It also shows what will not be done at a certain date, giving us early warnings: on 5 June, based on our current knowledge, Activities 17 and 18 won't be done. The earlier we get a warning, the more time we have to do something about it. Some notes: In this table we don't calibrate 'Still-to-Spend' (using calibration factor 1.0), because of assumed improved insight with Tasks almost done. Activities not yet started are calibrated by the ratio of Spent plus Still to Spend and the original estimates. Apparently, this is a snapshot of 29 March.

satisfied, and to be *more successful* than without us as testers, then testing can also use all the Evolutionary Project Planning techniques that development is already using. TaskCycles to organize and optimize the work, DeliveryCycles to see whether testing is doing the right job, and TimeLine to check that we are keeping in sync with development, not to unnecessarily delay the result. If testing isn't well aware of their actual customer, they are probably doing some things not right.

Looking at the developers' weekly (TaskCycle) planning, the testers know exactly what the developers will have done by the end of any week, so during that week they can plan exactly what and how to test in the following week, immediately upon delivery by the developers, not wasting any time. More explanation in [Mal05].

3.2 Evolutionary project management

Evolutionary Planning is one of the Evolutionary Project Management techniques, which evolutionarily have evolved based on actively and very frequently using the Plan-Do-Check-Act or Deming cycle, which is actually a continuous root-cause-analysis-*plus-consequence* (Act!) technique. Some people fear that these techniques will cost a lot of extra time. Recently a Project Manager said: "Do I have to do root-cause-analysis on *all* defects found? I can't spend that amount of time!" Apparently he thought he did have enough time to repair all the repeated defects that kept coming in, rather than preventing most of them. Experience in numerous projects proves that using these techniques, projects can quickly learn to conclude projects more successful, in *significantly shorter time*. A lot of time can be saved, both in development and testing, but we have to actively start looking for it. Evolutionary Project Management techniques help people doing this.
Elements of these techniques are:

- Plan-Do-Check-Act - the powerful ingredient for continuous learning and success
- Zero-Defects as an attitude - preventing half of the defects overnight [Cro84]
- Business Case - to define why we are doing the project
- Requirements Engineering - to define what we are supposed to achieve and what not, using quantification to define how much better performance we are supposed to achieve [Gil88], [Gil05]
- Architecture and Design - selecting the optimum compromise for the conflicting requirements (requirements are always conflicting: e.g. performance <> budget)
- Early Review & Inspection - measuring quality while doing, quickly learning to prevent injecting defects
- Weekly TaskCycle - short term planning, optimizing estimation, promising what we can achieve, and then living up to our promises
- Bi-weekly DeliveryCycle - optimizing the requirements, and checking the assumptions, soliciting feedback by delivering real results to eagerly waiting stakeholders
- TimeLine - getting and keeping control of Time: predicting the future, doing something with that knowledge, and feeding program/portfolio/ resource management with quite reliable results

More details can be read in [Gil88], [Gil05], [Mal10] and [Mal09]. With this paper I hope to have shown that testing can be planned just as any other project, using the same Evolutionary techniques we developed for development, to improve the performance of the tester's contributions to the success of the project, resulting in happy customers and hence in better revenues for the organization, ultimately for all people involved.

References

[Cro84] P.B. Crosby: Quality Without Tears, McGraw-Hill, 1984, ISBN 0070145113

[Dem86] W.E. Deming: Out of the Crisis, MIT, 1986, ISBN 0911379010

[Gil88] T. Gilb: Principles of Software Engineering Management, Addison-Wesley, 1988, ISBN 0201192462

[Gil05] T. Gilb: Competitive Engineering, Elsevier, 2005, ISBN 0750665076

[Mal05] N.R. Malotaux: Optimizing the Contribution of Testing to Project Success, 2005, www.malotaux.eu/booklets - booklet#3

[Mal09] N.R. Malotaux: Evolutionary Planning or How to Achieve the Most Important Requirement, 2009 www.malotaux.eu/booklets - booklet#7

[Mal10] N.R. Malotaux: Predictable Projects - How to deliver the Right Results at the Right Time, 2009 www.malotaux.eu/booklets - booklet#9

Niels Malotaux

Help ! We have a QA Problem !

This is about a real case of too many developers feeding too few testers, causing a testing backlog of half a year, with many angry customers waiting for too long for solutions to their problems. One senior tester just had left the company. There was only one senior and one junior tester left. They were facing this huge backlog of work and didn't know where to start.

We will show how empowerment of the testers, careful planning, and involvement of the developers allowed the testers to catch up in about 9 weeks, systematically making customers happy one by one along the way. The senior tester learnt how to plan the work of the testers effectively and efficiently in sync with the developers, so that there were no backlogs ever since. Trust by customers who were abandoning the supplier was restored, causing turnover to grow enormously since.

We will first show how we used Evolutionary Planning techniques in this particular case. Then we will discuss in more general terms the elements of this planning technique.

Niels Malotaux is an independent Project Coach specializing in optimizing project performance. Since 1974 he designed electronic hardware 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: 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 over 400 projects in 40+ organizations in the Netherlands, Belgium, China, Germany, India, Ireland, Israel, Japan, Romania, South Africa, Serbia, the UK, 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, see www.malotaux.eu/conferences

Find more booklets at: www.malotaux.eu/booklets

1. Evolutionary Project Management Methods
2. How Quality is Assured by Evolutionary Methods
3. Optimizing the Contribution of Testing to Project Success

3a. Optimizing Quality Assurance for Better Results (same as 3, but now for non-software projects)

4. Controlling Project Risk by Design
5. TimeLine: Getting and Keeping Control over your Project
6. Recognizing and Understanding Human Behaviour
7. Evolutionary Planning (similar to booklet#5 TimeLine, but other order, and added predictability)
8. Help! We have a QA problem! (this booklet)
9. Predictable Projects - How to deliver the Right Results at the Right Time (newer description)

N R Malotaux
Consultancy

Niels R. Malotaux
phone +31-655 753 604
mail niels@malotaux.eu
web www.malotaux.eu

Originally prepared for the Conquest 2009 Conference, Nuremberg, Germany
Version 1.5 (some commas, links checked) - 26 Nov 2025