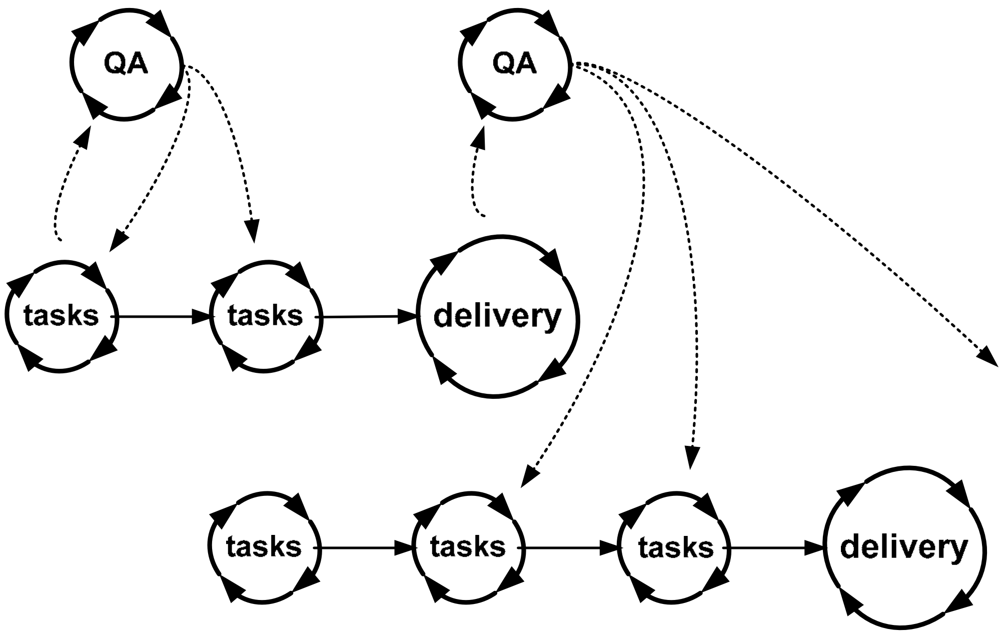
Niels Malotaux

Optimizing Quality Assurance for Better Results



Optimizing Quality Assurance for Better Results

1 Introduction

We know all the stories about failed and partly failed projects, only about one third of software projects delivering according to their original goal [1]. Failures are not limited to software projects: also system level projects suffer from many smaller and larger failures, resulting in time and cost overruns or even complete systems failure.
Apparently, despite all the efforts for doing a good job, too many defects are generated by developers, and too many remain undiscovered by checkers, causing still too many problems to be experienced by users. It seems that people are taking this state of affairs for granted, accepting it as a nature of development. After all, people make mistakes, don't they? Solutions are mostly sought in technical means like process descriptions, metrics and tools. If this really would have helped, it should have shown by now.
Oddly enough, there is a lot of knowledge about how to significantly reduce the generation and proliferation of defects and deliver the right solution quicker. Still, this knowledge is ignored in the practice of many development organizations. In papers and in actual projects I've observed that the time spent on testing and repairing is quoted as being up to 60 to 80% of the total project time. That's a large budget and provides excellent room for a lot of savings.
In 2004, I published a booklet: *How Quality is Assured by Evolutionary Methods* [2], describing practical implementation details of how to organize projects using this knowledge, making the project a success. In an earlier booklet: *Evolutionary Project Management Methods* [3], I described issues to be solved with these methods and my first practical experiences with the approach. Tom Gilb published already in 1988 about these methods [4].
In this paper we'll extend the Evo methods to QA, in order to optimize the contribution of QA to project success.
Important ingredients for success are: a change in attitude, taking the Goal seriously, which includes working towards defect-free results, focusing on prevention rather than repair, and constantly learning how to do things better.

2 The Goal

Let's define as the main goal of our software development efforts:

> *Providing the customer with what he needs, at the time he needs it,*
> *to be satisfied, and to be more successful than he was without it …*

If the customer is not satisfied, he may not want to pay for our efforts. If he is not successful, he *cannot* pay. If he is not more successful than he already was, why should he invest in our work anyway? Of course we have to add that what we do in a project is:

> *… constrained by what the customer can afford and what we mutually*
> *beneficially and satisfactorily can deliver in a reasonable period of time.*

Furthermore, let's define a Defect as:

> *The cause of a problem experienced by the stakeholders of the system.*

If there are no defects, we'll have achieved our goal. If there are defects, we failed.

3 The knowledge

Important ingredients for significantly reducing the generation and proliferation of defects and delivering the right solution quicker are:

- **Clear Goal**: If we have a clear goal for our project, we can focus on achieving that goal. If management does not set the clear goal, we should set the goal ourselves.
- **Prevention attitude**: Preventing defects is more effective and efficient than injecting-finding-fixing, although it needs a specific attitude that usually doesn't come naturally.
- **Continuous Learning**: If we organize projects in very short Plan-Do-Check-Act (PDCA) cycles, constantly selecting only the most important things to work on, we will most quickly learn what the real requirements are and how we can most effectively and efficiently realize these requirements. We spot problems quicker, allowing us more time to do something about them. Actively learning is sped up by expressly applying the Check and Act phases of PDCA.

4 Evo

Evolutionary Project Management (*Evo* for short) uses this knowledge to the full, combining Project-, Requirements- and Risk-Management into Result Management. The essence of Evo is actively, deliberately, rapidly and frequently going through the PDCA cycle, for the product, the project *and* the process, constantly reprioritizing the order of what we do based on Return on Investment (ROI), and highest value first. In my experience as project manager and as project coach, I observed that projects seriously applying the Evo approach, are routinely successful on time, or earlier [5].

Evo is not only iterative (using multiple cycles) and incremental (breaking the work into small parts), but above all Evo is about *learning*. We proactively anticipate problems before they occur and work to prevent them. We may not be able to prevent all the problems, but if we prevent most of them, we have a lot more time to cope with the few problems that slip through.

5 Something is not right

Satisfying the customer and making him more successful implies that the system we deliver should show no defects. So, all we have to do is delivering a result with no defects. As long as a lot of projects deliver their results with defects and late (which I consider a defect as well), apparently something is not right.

Customers are also to blame, because they keep paying when the system is not delivered as agreed. If they would refuse to pay, the problem could have been solved long ago. One problem here is that it often is not obvious what was agreed. However, as this is a *known problem*, there is no excuse if this problem is not solved within the project, well before the end of the project.

6 Defects found are symptoms

Many defects are symptoms of deeper lying problems. Defect prevention seeks to find and analyze these problems and doing something more fundamental about them.

Simply repairing the apparent defects has several drawbacks:

- Repair is usually done under pressure, so there is a high risk of imperfect repair, with unexpected side effects.
- Once a bandage has covered up the defect, we think the problem is solved and we easily forget to address the real cause. That's a reason why so many defects are still being repeated.
- Once we find the underlying real cause, of which the defect is just a symptom, we'll probably do a more thorough redesign, making the repair of the apparent defect redundant.

As prevention is better than cure, let's move from *fixation-to-fix* to *attention-to-prevention*.

Many mistakes have a repetitive character, because they are a product of certain behavior of people. If we don't deal with the root causes, we will keep making the same mistakes over and over again. Without feedback, we won't even know. With quick feedback, we can put the repetition immediately to a halt.

7 Defects typically overlooked

We must not only check whether functions are correctly implemented (verification) as documented in the requirements, but also, a level higher, whether the requirements adequately solve the needs of the customer according to the goal (verification by the project to make sure they do deliver the right things; validation by or for the customer, for acceptance).

Some types of defects that are commonly overlooked are:

- **Functions that won't be used** (superfluous requirements, no Return on Investment)
- **Nice things** (not required, added by designers or programmers, usefulness not checked, not paid for)
- **Missing quality levels** (should have been in the requirements)
  e.g.: response time, security, dependability, maintainability, usability, learnability
- **Missing constraints** (should have been in the requirements)
- **Unnecessary constraints** (not required)
- **Being late or over budget** (few people learnt to treat these as *defects*)

Another problem that may negatively affect our goal is that many projects end at "Hurray, it works!". If the goal of our project is to make the customer more successful, our responsibility goes further: we have to make sure that the increase in success is *going to happen*. This awareness will stimulate our understanding of quality requirements like "learnability" and “usability”. Without it, these requirements don’t have much meaning for developers. It’s a defect if success is not going to happen.

8 Are defect free results possible?

Many people think that we inevitably make mistakes and that the mere idea of Zero Defects is a fallacy. This is probably caused by lack of under-standing about what Zero Defects really means. Think of it as an asymptote (Figure 1). We know that an asymptote never reaches its target, but we can do our best to approach the target level as closely as possible. However, if we put the bar at an *acceptable level* of defects, we’ll asymptotically approach that level. Only if we put the bar at zero defects, we can asymptotically approach Zero Defects.

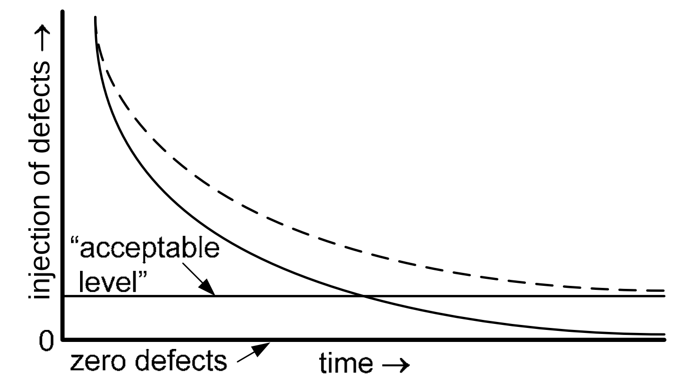


Figure 1: Zero Defects is an asymptote. The curve is also called: “learning curve”

Philip Crosby writes [6]:

> Conventional wisdom says that error is inevitable. As long as the performance standard requires it, then this self-fulfilling prophecy will come true. Most people will say: People are humans and humans make mistakes. And people do make mistakes, particularly those who do not become upset when they happen. Do people have a built-in defect ratio? Mistakes are caused by two factors: lack of knowledge and lack of attention. Lack of attention is an attitude problem.

When Crosby first started to apply Zero Defects as performance standard in 1961, the error rates dropped 40% almost immediately [6]. In my projects I’ve observed similar effects.

Experience: No defects in the first two weeks of use

A QA person of a large banking and insurance company I met in a SPIN metrics working group told me that they got a new manager who told them that from now on she expected that any software delivered to the (internal) users would run defect free for at least the first two weeks of use. He told me this as if it were a good joke. I replied that I thought he finally got a good manager, setting them a clear requirement: “No defects in the first two weeks of use”. Apparently this was a target they had never contemplated before, nor achieved. Now they could focus on how to achieve defect free software, instead of counting function points and defects. Remember that in bookkeeping being one cent off is already a capital offense, so defect free software should be a normal expectation for a bank. Wouldn’t it be for *any* environment?

Zero Defects is a performance standard, set by management. In Evo projects, even if management does not provide us with this standard, we’ll assume it as a standard for the project, because we know that it will help us to conclude our project successfully in less time.

9 Attitude

As long as we are convinced that Zero Defects is impossible, we’ll keep producing defects, failing our goal. As long as we are accepting defects, we are endorsing defects. The more we talk about them, the more normal they seem. It's a self-fulfilling prophecy. It will perpetuate the problem. So, let’s challenge the defect-cult and do something about it:

From now on, we don’t want to make mistakes any more. We get upset if we make one. We feel the failure. If we don't feel failure, we don’t learn. Then we work to find a way not to make the mistake again. If a task is finished we don’t *hope* it’s ok, we don’t *think* it’s ok, no, we’ll *be sure* that there are no defects and we’ll be genuinely surprised when there proves to be any defect after all.

In my experience, this attitude prevents half of the defects in the first place. Because we are humans, we can study how we operate psychologically and use this knowledge to our advantage. If we can prevent half of the defects overnight, then we have a lot of time for investing in more prevention, while still being more productive. This attitude is a crucial element of successful projects.

Experience: No more memory leaks

My first Evo project was a project where people had been working for months on software for a hand-held terminal. The developers were running in circles, adding functions they couldn't even test, because the software crashed before they arrived at their newly added function. The project was already late and management was planning to kill the project. We got six weeks to save it.

The first goal was to get stable software. After all, adding any function if it crashes within a few minutes of operation is of little use: the product cannot be sold. I told the team to take away all functionality except one very basic function and then to make it stable. The planning was to get it stable in two weeks and only then to add more functionality gradually to get a useful product.

I still had other business to finish, so I returned to the project two weeks later. I asked the team "Is it stable?". The answer was: "We found many memory leaks and solved them. Now it's much *stabler*". And they were already adding new functionality. I said: "Stop adding functionality. I want it stable, not *almost* stable". One week later, all memory leaks were solved and stability was achieved. This was a bit of a weird experience for the team: the software didn't crash any more. Actually, in this system there was not even a need for dynamically allocatable memory and the whole problem could have been avoided. But changing this architectural decision wasn't a viable option at this stage any more.

Now that the system was stable, they started adding more functions. We got another six weeks to complete the product. I made it very clear that I didn't want to see any more memory leaks. Actually that I didn't want to see any defects. The result was that the testers suddenly found hardly any defect any more and from now on could check the correct functioning of the device. At the end of the second phase of six weeks, the project was successfully closed. The product manager was happy with the result.

Conclusion: after I made it clear that I didn't want to see any defects, the team hardly produced any defects. The few defects found were easy to trace and repair. The change of attitude saved a lot of defects and a lot of time. The team could spend most of its time adding new functionality instead of fixing defects. This was Zero Defects at work. Technical knowledge was not the problem to these people: once challenged, they quickly came up with tooling to analyze the problem and solve it. The attitude was what made the difference.

10 Plan-Do-Check-Act

I assume the Plan-Do-Check-Act (PDCA- or Deming-) cycle [7] is well known (Figure 2). Because it's such a crucial ingredient, I'll shortly reiterate the basic idea:

- We ***Plan*** *what* we want to accomplish and *how* we think to accomplish it best.
- We ***Do*** according to the plan.
- We ***Check*** to observe whether the result from the *Do* is according to the *Plan*.
- We ***Act*** on our findings. If the result was good: what can we do better. If the result was not so good: how can we make it better. *Act* produces a renewed strategy.

The key-ingredients are: planning before doing, doing according to the plan, systematically checking and above all *acting*: doing things *differently*. After all, if you don't do things differently, you shouldn't expect a *change* in result, let alone an *improvement* in result.

In Evo we constantly go through multiple PDCA cycles, deliberately adapting strategies in order to learn how to do things better all the time, actively and purposely speeding up the evolution of our knowledge. As a driver for moving the evolution in the right direction, we use Return on Investment (ROI): the project invests time and other resources and this investment has to be regained in whatever way, otherwise it's just a hobby. So, we'll have to constantly be aware whether all our actions contribute to the value of the result. Anything that does not contribute value, we shouldn't do.

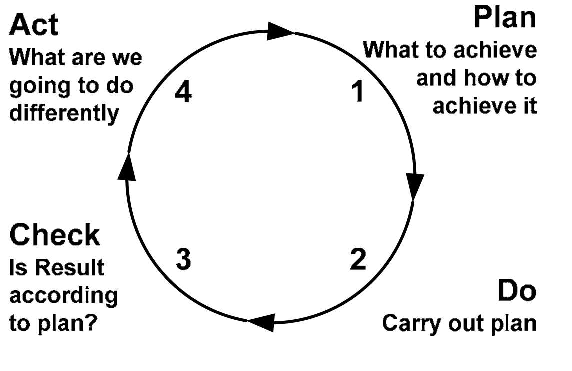

Act
What are we going to do differently
4
Plan
What to achieve and how to achieve it
1
Check
Is Result according to plan?
3
2
Do
Carry out plan

Figure 2: PDCA or Deming cycle

Furthermore, in order to maximize the ROI, we have to do the *most important things* first. In practice, priorities change dynamically during the course of the project, so we constantly reprioritize, based on what we learnt so far. Every week we ask ourselves: "What are the most important things to do. We shouldn't work on anything less important." Note that priority is molded by many issues: customer issues, project issues, technical issues, people issues, political issues and many other issues.

11 How about Project Evaluations

Project Evaluations (also called Project Retrospectives, or Post-Mortems - as if all projects die) are based on the PDCA cycle as well. At the end of a project we evaluate what went wrong and what went right (things that went right may have gone right accidentally).

Doing this only at the end of a project has several drawbacks:

- We tend to forget what went wrong, especially if it was a long time ago.
- We put the results of the evaluation in a write-only memory: do we really remember to check the evaluation report at the very moment we need the analysis in the next project? Note that this is typically one full project duration after the fact. So there is not much benefit for the next project.
- The evaluations are of no use for the project just finished and being evaluated.
- Because people feel these drawbacks, they tend to postpone or forget to evaluate. After all, they are already busy with the next project, after the delay of the previous project.

In short: the principle is good, but the implementation is not tuned to the human time-constant.

In Evo, we evaluate *weekly* (in reality it gradually becomes a way-of-life), using PDCA cycles, and now this starts to bear fruit (Figure 3):

- Not so much happens in one week, so there is not so much to evaluate.
- It's more likely that we remember the issues of the past five days.
- Because we most likely will be working on the same kind of things during the following week, we can immediately use the new strategy, based on our analysis.
- One week later we can check whether our new strategy was better or not, and refine.
- Because we immediately apply the new strategy, it naturally is becoming our new way of working.
- The current project benefits immediately from what we found and improved.

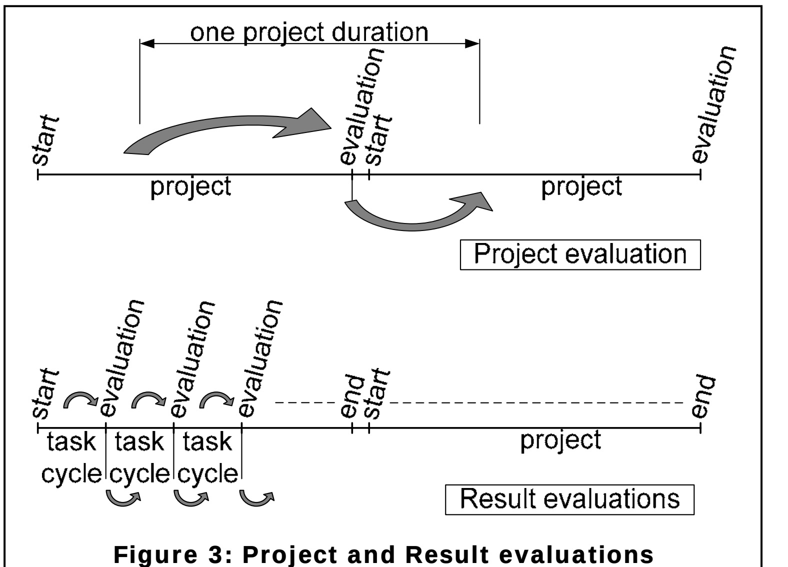


Figure 3: Project and Result evaluations

Evaluations are good, but they must be tuned to the right cycle time to make them really useful. The same applies to all verification activities, as these are also evaluations.

12 Current Evo Testing

In conventional development mode, most verification is still executed in Waterfall mode: developers are first allowed to inject defects (in drawings, designs, pieces of code, or pieces of hardware), then testers and checkers are supposed to find the defects injected, after which the developers are supposed to repair the defects found. In reality, testers and checkers find only part (30 – 80%) of the defects injected (testers and checkers are human as well). In Evo, we humbly admit that we probably don't know the real requirements, that we have to check our assumptions and that we are prone to making mistakes. Evo Quality Assurance assists the development people to reach their goal successfully. This includes verification of all phases of the development process and ploughing back the findings to the developers for optimizing the product, the project and the process.

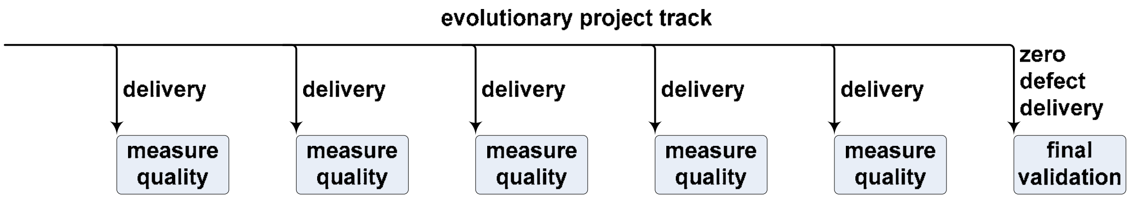


Figure 4: Checking of early deliveries helps the developers to get ready for zero-defect final delivery

Developers design the order of Deliveries in such a way that, in case they made an erroneous assumption or a downright error, it will be found as quickly as possible (figure 4). This way, most of any undiscovered defects will be caught before the final delivery and, more importantly, be exploited for prevention of further injection of similar defects. Evo projects do not need a separate verification (sometimes called "debugging") phase and hardly need repair after delivery. If a delivery is ready, it is complete. Anything is only ready if it is completely done, *not to worry about it any more*. That includes: no defects. I know we are human and not perfect, but remember the importance of attitude: we *want* to be perfect. Because all people in the project aim for Zero Defects delivery, the developers and QA work together in their quest for perfection. Note that perfection means: *freedom from fault or defect*. It does *not* mean: *gold plating*.

13 Further improvement

In the original Evo concept we gained a lot by preventing the injection of defects, because people learn during the work: if a designer has to produce several documents (plans, drawings, designs, pieces of code) or pieces of hardware, he can learn from his mistakes made in the first item, and avoid making similar mistakes in subsequent similar work.
As long as single pieces of work are still made in waterfall mode, first "completed" and only subsequently checked, we are still waiting for the designers to inject defects first, hoping that we can find and fix all these defects afterwards. In order to drive prevention further, why don't we contemplate checking the result of designers before the first item is completed, so that they can prevent mistakes immediately, avoiding the waterfall-syndrome even on single pieces of work. This may seem overkill in case of a first small document of a large set of documents. It makes a lot of sense, however, in case of one single, or a relatively large document.

We can also extend the Evo project management techniques to the QA process itself and exploit the PDCA paradigm even further:

- QA focuses on a clear goal. Finding defects is not the main goal. After all, we don't want defects. (Of course, if there are defects, we hope to find them. The problem is that we *know* we won't find them all).
- QA will select and use any method appropriate for optimum feedback to development, be it testing, review or inspection, or whatever more they come up with.
- QA checks work in progress *even before* it is delivered, to feedback issues found, allowing the developers to abstain from further producing these issues in the remainder of their work.
  "Can I check some piece of what you are working on now?" "But I'm not yet ready!" "Doesn't matter. Give me what you have. I'll tell you what I find, if I find anything". Checkers have a different view, seeing things the developer doesn't see. Developers don't naturally volunteer to have their intermediate work checked. Not because they don't like it to be checked, but because their attention is elsewhere. QA can help by asking. Initially the developers may seem a little surprised, but this will soon fade. *If* QA plays the game well.
- Similarly, QA can solve a typical problem with planning reviews and inspections. Developers are not against reviews and inspections, because they very well understand the value. They have trouble, however, planning them in between of their design work, which consumes their attention more. If we include QA directly in the process, QA can recognize when which types of review, inspections or tests are needed and organize these accordingly. This is a natural part of their work helping the developers to minimize rework by minimizing the injection of defects and minimizing the time slipped defects stay in the system.

In general: organizing QA the Evo way means entangling the verification process more intimately with the development process.

14 Cycles in Evo

In the Evo development process, we use several learning cycles (see [2] and [3] for explanations of terms):

- The TaskCycle [8] is used for organizing the work, optimizing estimation, planning and tracking. We constantly check whether we are doing the right things in the right order to the right level of detail. We optimize the work effectiveness and efficiency. TaskCycles never take more than one week.
- The DeliveryCycle [9] is used for optimizing the requirements and checking the assumptions. We constantly check whether we are moving to the right product results. DeliveryCycles focus the work organized in TaskCycles. DeliveryCycles normally take not more than two weeks.
- TimeLine [10] is used to keep control over the project duration. We optimize the order of DeliveryCycles on the TimeLine in such a way that we approach the product result in the shortest time, with as little rework as possible.

During these cycles we are constantly optimizing:

- **The product** [11]: how to arrive at the best product (according to the goal).
- **The project** [12]: how to arrive at this product most effectively and efficiently.
- **The process** [13]: finding ways to do it even better. Learning from other methods and absorbing those methods that work better, shelving those methods that currently work less effectively.

If we do this well, by definition, there is no better way.

15 Evo cycles for testing

Extending Evo to QA adds cycles (figure 5) for feedback from QA checking and testing to development, as well as cycles for organizing and optimizing the verification activities themselves:

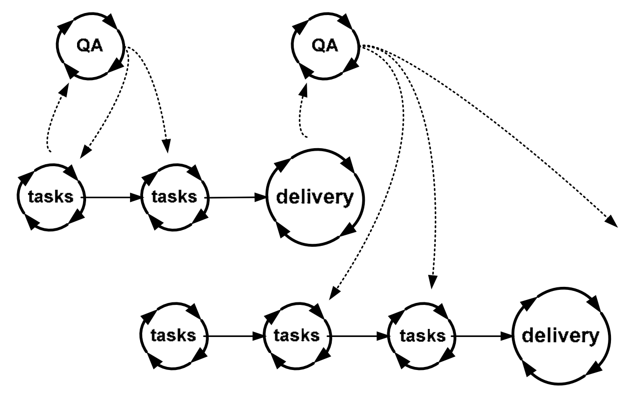


Figure 5: Adding QA-cycles to an Evo project

- QA organizes their own work in weekly, or even shorter TaskCycles.
- The DeliveryCycle of QA is the QA-feedback cycle: in very short cycles checkers take intermediate results from developers, check for defects in all varieties and feed back optimizing information to the developers, while the developers are still working on the same results. This way the developers can avoid injecting defects in the remainder of their work, while immediately checking out their prevention ideas in reality.
- QA use their own TimeLine, synchronized with the development TimeLine, to control that they plan the right things at the right time, in the right order, to the right level of detail during the course of the project and that they conclude their work in sync with development.

During these cycles the checkers are constantly optimizing:

- **The product**: how to arrive at the most effective product.
  Remember that their product goal is: providing their customer, in this case *the developers*, with what they need, at the time they need it, to be satisfied, and to be more successful than they were without it.
- **The project**: how to arrive at this product most effectively and efficiently.
  This is optimizing in which order they should do which activities to arrive most efficiently at their result.
- **The process**: finding ways to do it better. Learning from other methods and absorbing those methods that work better, shelving those methods that currently work less effectively.

Checkers are part of the project and participate in the weekly 3-step procedure [14] using about 20 minutes per step:

1. Individual preparation.
2. 1-to-1's: Modulation with and coaching by Project Management .
3. Team meeting: Synchronization and synergy with the team.

Project Management in step 2 is now any combination, as appropriate, of the following functions:

- The Project Manager or Project Leader, for the project issues.
- The Architect, for the product issues.
- The QA Manager, for the QA issues.

There can be only one captain on the ship, so the final word is to the person who acts as Project Manager, although he should better listen to the advice of the others.

QA people participate in requirements discussions. They communicate with developers in the unplannable time [15], or if more time is needed, they plan tasks for interaction with developers. If the priority of an issue is too high to wait for the next TaskCycle, the interrupt procedure [16] will be used. If something is unclear, an Analysis Task [17] will be planned. The Prevention Potential of issues found is an important factor in the prioritizing process.

In the team meeting checkers see what the developers will be working on in the coming week and they synchronize with that work. There is no ambiguity any more about which requirements can be tested and to which degree, because the checkers follow development, and they design their contribution to assist the project optimally for success.

In Evo QA, we don't wait until something is thrown at us. We actively take responsibility. Prevention doesn't mean sitting waiting for the developers. It means to decide *with* the developers how to work towards the defect free result together. Developers doing a small step. Checkers checking the result and feeding back any imperfections before more imperfections are generated, closing the very short feedback loop. Developers and checkers quickly finding a way of optimizing their cooperation. It's important for the whole team to keep helping each other to remind that we don't want to repair defects, because repair costs more. If there are no defects, we don't have to repair them.

In many cases, the deadline of a project is defined by genuine external factors like a finite market-window. Then we have to predict which requirements we can realize before the deadline or "FatalDate". Therefore, we still need to estimate the amount of work needed for the various requirements. We use the TimeLine technique to regularly predict what we will have accomplished at the FatalDate and what not, and to control that we will have a working product well before that date. QA people in the project use their own TimeLine to control that they will complete whatever they have to do in the project, in sync with the developers.

Doesn't all of this take a lot of time? No. My experience with many projects shows that it saves time, projects successfully finishing well before expected. At the start it takes some more time. The attitude, however, results in less defects and as soon as we focus on prevention rather than continuous injection-finding-fixing, we soon decrease the number of injected defects considerably and we don't waste time on all those defects any more.

16 Database for Change Requests, Problem Reports and *Risk Issues*

Most projects already use some form of database to collect defects reported (Problem Report/PR: development pays) and proposed changes in requirements (Change Request/CR: customer pays).

If we are seriously in Prevention Mode, striving for Zero Defects, we should also collect Risk Issues (RI): issues which better be resolved *before* culminating into CR's or PR's.

With the emphasis shifted from repair to prevention, the RI/CR/PR database will have to provide additional space for the collection of data to specifically support the prevention process like:

- Follow-up status
- When, where and why found
- Where caused and root cause
- Where should it have been found earlier
- Why didn't we find it earlier
- Prevention plan
- Analysis task defined and put on the Candidate Task List [18]
- Prevention task(s) defined and put on the Candidate Task List
- Check lists updated for finding this issue easier, in case prevention doesn't work yet.

Analysis tasks may be needed to sort out the details. The analysis, prevention and repair tasks are put on the Candidate Task List and will, like all other candidate tasks, be handled when their time has come: if nothing else is more important. Analysis tasks, prevention tasks and repair tasks should be separated, because analysis and prevention usually have priority over repair. We better first stop the leak, to make sure that not more of the same type of defect is injected.

17 How about metrics?

In Evo, the time to complete a task is estimated as a TimeBox [19], within which the task will be 100% done. This eliminates the need for tracking considerably. The estimate is used during the execution of the task to make sure that we complete the task on time. We experienced that people can quite well estimate the time needed for completing tasks, if we are really serious about time.

Note that exact task estimates are not required. Planning at least 4 tasks in a week allows some estimates to be a bit optimistic and some to be a bit pessimistic. All we want is that, at the end of the week, people have finished what they promised. As long as the average estimation is OK, all tasks can be finished at the end of the week. As soon as people learn not to overrun their (average) estimates any more, there is no need to track or record overrun metrics. The attitude replaces the need for the metric. So, we do use metrics and measurements in Evo, but we are very reluctant to accumulate a lot of measurement data because of the limited use of the data for project success. We rather use the data to immediately learn. Once we have learnt, the old data has no meaning any more.

It can be useful to know the average time of designing a circuit board or parts of a building of a given size and complexity. It can also be useful to know the average time to build some hardware or parts of a building. We can optimize these times, but these times will not become zero: there is always a finite time needed to complete these tasks. These metrics are useful to predict the cost of the development.

For defects, however, the goal is Zero Defects. And when there are no defects, there is no cost-of-defects (cost of non-quality) involved. So, what’s the use of “knowing” the number of defects “to be expected”?

Several typical testing metrics become irrelevant when we aim for defect free results, for example:

- **Defects per Page, per Drawing, per kLoC**
  Counting defects condones the existence of defects, so there is an important psychological reason to discourage counting them.
- **Incoming defects per month**, found by test, found by users
  Don’t count incoming defects. Do something about them. Counting conveys a wrong message. We should better make sure that the user doesn’t experience any problem.
- **Defect detection effectiveness** or **Inspection yield**
  (found by test / (found by test + customer))
  There may be some defects left, because perfection is an asymptote. It’s the challenge for checkers and testers to find them all. Results in practice are in the range of 30% to 80%. Testers and checkers apparently are not perfect either. That’s why we must strive towards zero defects *before* final test. Whether that is difficult or not, is beside the point.
- **Cost to find a defect**
  The less defects there are, the higher the cost to find the few defects that slip through from time to time, because we still have to spend the time to test, to see that the result is OK. This was a bad metric anyway.
- **Numbers and types of issues resolved and unresolved, age of open customer found issues**
  Whether, when and how an issue is closed or not depends on the Evo prioritizing process. Every week any issues are handled, appropriate tasks are defined and put on the Candidate Task List, to be handled when their time has come. It seems that many metrics are there because we don’t trust the developers to take appropriate action. In Evo, we do take appropriate action, so we don’t need policing metrics.
- **When are we done with testing?**
  Examples from conventional projects: if the number of defects found per day has declined to a certain level, or if the defect backlog has decreased to zero. In some cases, curve fitting with early numbers of defects found during the testing phase is used to *predict* the moment the defect backlog will have decreased to zero. Another technique is to predict the number of defects *to be expected* from historical data. In Evo projects, the project will be ready at the agreed date, or earlier. That includes all appropriate testing being done.

Instead of *improving* non-value adding activities, including various types of metrics, it is better to *eliminate* them. In many cases (but not all!), the attitude, and the assistance of the Evo techniques replace the need for metrics. Other metrics may still be useful, like Remaining Defects, as this metric provides information about the effectiveness of the prevention process. Still, even more than in conventional metrics activities, we will be on the alert that whatever we do must contribute value. If people have trouble deciding what the most important work for the next week is, I usually suggest as a metric: "The size of the smile on the face of the customer". If one solution does not get a smile on his face, another solution does cause a smile and a third solution is expected to put a big smile on his face, which solution shall we choose? This proves to be an important Evo metric that helps the team to focus.

18 Finally

Many development organizations in the world are working the same way, producing defects and then trying to find and fix the defects found, waiting for the customer to experience the reminder. In some cases, the service organization is the profit-generator of the company. And isn't the testing department assuring the quality of our products?
That's what the car and electronics manufacturers thought until the Japanese products proved them wrong. So, eventually the question will be: can we afford it?
The complexity of our systems is growing exponentially, and the capacity needed to develop these systems meaningfully is growing even faster with it. So, why not better become more productive by not injecting the vast majority of defects. Then we have more time to spend on more challenging activities than finding and fixing defects.
I absolutely don't want to imply that finding and fixing is not challenging. Prevention is just cheaper. And, testers and checkers, fear not: even if we start aiming at defect free systems, we'll still have to learn a lot from the mistakes we'll still be making.

References

[1] The Standish Group: *Chaos Report*, 1994, 1996, 1998, 2000, 2002, 2004. http://www.standishgroup.com/chaos_resources/index.php

[2] N.R. Malotaux: *How Quality is Assured by Evolutionary Methods*, 2004. Also downloadable as a booklet: http://www.malotaux.nl/nrm/pdf/Booklet2.pdf

[3] N.R. Malotaux: *Evolutionary Project Management Methods*, 2001. http://www.malotaux.nl/nrm/pdf/MxEvo.pdf

[4] T. Gilb: *Principles of Software Engineering Management*, 1988. Addison-Wesley Pub Co, ISBN: 0201192462.

[5] See cases: http://www.malotaux.nl/nrm/Evo/EvoFCases.htm

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

[7] W.E. Deming: *Out of the Crisis*, 1986. MIT, ISBN 0911379010. M. Walton: *Deming Management At Work*, 1990. The Berkley Publishing Group, ISBN 0399516859.

[8]	TaskCycle	ref [2] chapter 5.1	ref [3] chapter 3C
[9]	DeliveryCycle	ref [2] chapter 5.1	ref [3] chapter 3C
[10]	TimeLine	ref [2] chapter 5.5 and 6.8	
[11]	Product	ref [2] chapter 4.2	
[12]	Project	ref [2] chapter 4.3	
[13]	Process	ref [2] chapter 4.4	
[14]	3-step procedure	ref [2] chapter 6.9	
[15]	Unplannable time	ref [2] chapter 6.1	
[16]	Interrupt procedure	ref [2] chapter 6.7	
[17]	Analysis task	ref [2] chapter 6.6	ref [3] chapter 8
[18]	Candidate Task List	ref [2] chapter 6.5	ref [3] chapter 8
[19]	TimeBox	ref [2] chapter 6.4	ref [3] chapter 3D

Niels Malotaux

Optimizing Quality Assurance for Better Results

Let's define the Goal of projects as: Providing the customer with what he needs, at the time he needs it, to be more successful than he was without it, constrained by what we can deliver in a reasonable period of time. Furthermore, let's define a defect as the cause of a problem experienced by the users of our product. If there are no problems, we will have achieved our goal. If there are problems, we failed.
We know all the stories about failed and partly failed projects. Apparently, too many defects are generated by developers, and too many remain undiscovered by checkers, causing too many problems to be experienced by users. Solutions are mostly sought in technical means like processes, metrics and tools. If this really would have helped, it should have shown by now.
Oddly enough, there is a lot of knowledge how to reduce the generation and proliferation of defects and deliver the right solution quicker. Still, this knowledge is ignored in many development organizations.
In 2004, I published a booklet: *How Quality is Assured by Evolutionary Methods*, describing how to organize projects using this knowledge successfully. In this paper we'll extend the use of this knowledge to QA, in order to optimize the contribution of QA to project success.
Important ingredients are: a change in attitude, taking the Goal seriously, focusing on prevention rather than repair, and constantly learning how to do things better.

Niels Malotaux is an independent Project Coach specializing in optimizing project performance. He has over 30 years experience in designing hardware and software systems. Since 1998, he devotes his expertise to teaching projects how to deliver Quality On Time: delivering what the customer needs, when he needs it, to enable customer success. Since 2001, he coached more than 40 projects in 14 different organizations, which led to a wealth of experience in which approaches work better and which work less well.

N R Malotaux
Consultancy

Niels R. Malotaux
Bongerdlaan 53
3723 VB Bilthoven
The Netherlands

Phone	+31-30-228 88 68
Fax	+31-30-228 88 69
Mail	niels@malotaux.nl
Web	http://www.malotaux.nl/nrm/English
Evoweb	http://www.malotaux.nl/nrm/Evo

Based on: *Optimizing the Contribution of Testing to Project Success*
Version QA0.1a - 19 Apr 2006