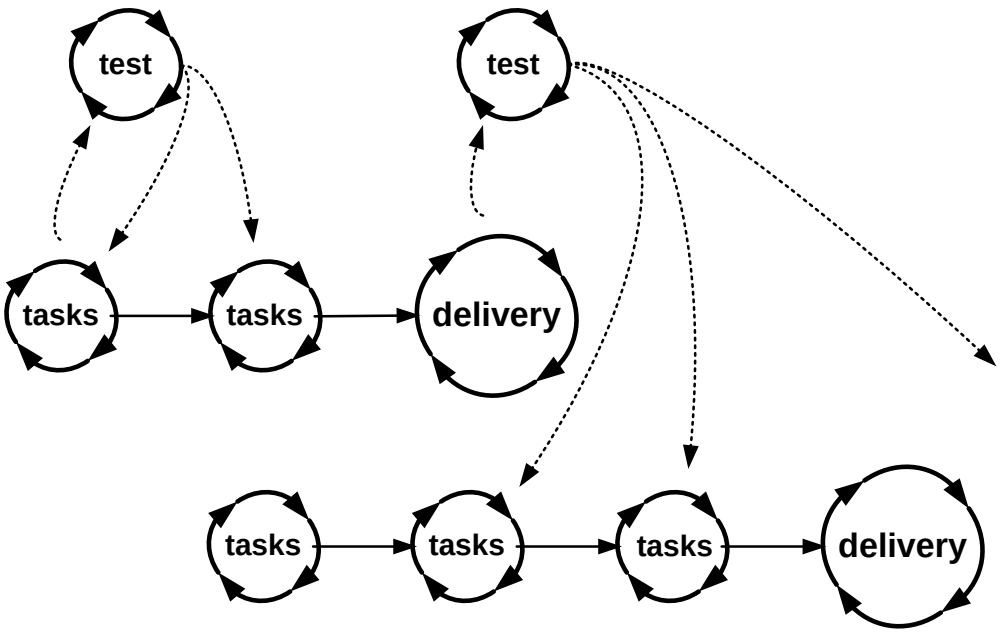
Niels Malotaux

Optimizing the Contribution of Testing to Project Success



Optimizing the Contribution of Testing to Project Success

1 Introduction

We know all the stories about failed and partly failed projects, only about one third of the projects delivering according to their original goal [1].

Apparently, despite all the efforts for doing a good job, too many defects are generated by developers, and too many remain undiscovered by testers, 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 software development. A solution is 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 software development organizations. In papers and in actual projects I've observed that the time spent on testing and repairing (some people call this *debugging*) is quoted as being 30 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 booklet we'll extend the Evo methods to the testing process, in order to optimize the contribution of testing 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 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

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 those projects who seriously apply the Evo approach, are routinely successful on time, or earlier [5].

Evo is not only iterative (using multiple cycles) and incremental (we break the work into small parts), like many similar Agile approaches, 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.

4 Something is not right

Satisfying the customer and making him more successful implies that the software 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 software is delivered 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 software 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.

5 The problem with bugs

In a conventional software development process, people develop a lot of software with a lot of defects, which some people call *bugs*, and then enter the *debugging* phase: testers testing the software and developers repairing the *bugs*.
Bugs are so important that they are even counted. We keep a database of the number of bugs we found in previous projects to know how many bugs we should expect in the next project. Software without bugs is even considered suspect. As long as we put bugs in the centre of the testing focus, there will be bugs. Bugs are normal. They are needed. What should we do if there were no bugs anymore?

This way, we *endorse* the injection of bugs. But, does this have anything to do with our goal: making sure that the customer will not encounter any problem?

Personally, I dislike the word bug. To me, it refers to a little creature creeping into the software, causing trouble beyond our control. In reality, however, people make mistakes and thus cause defects. Using the word *bug*, subconsciously defers responsibility for making the mistake. In order to prevent defects, however, we have to actively take responsibility for our mistakes.

6 Defects found are symptoms

Many defects are symptoms of deeper lying problems. Defect prevention seeks to find and analyse 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 behaviour 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 test whether functions are correctly implemented as documented in the requirements, but also, a level higher, whether the requirements adequately solve the needs of the customer according to the goal. Typical defects that may be overlooked are:

- Functions that won't be used (superfluous requirements, no Return on Investment)
- Nice things (added by programmers, usefulness not checked, not required, not paid for)
- Missing quality levels (should have been in the requirements)
  e.g.: response time, security, maintainability, usability, learnability
- Missing constraints (should have been in the requirements)
- Unnecessary constraints (not required)

Another problem that may negatively affect our goal is that many software projects end at "Hurray, it works!" If our software is supposed 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 development. It's a defect if success is not going to happen.

8 Is defect free software possible?

Most people think that defect free software is impossible. This is probably caused by lack of understanding about what defect free, or Zero Defects, really means. Think of it as an asymptote (Figure 1). We know that an asymptote never reaches its target. However, if we put the bar at an *acceptable level* of defects, we'll asymptotically approach that level. If we put the bar at zero defects, we can asymptotically approach that level.

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. Why 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.

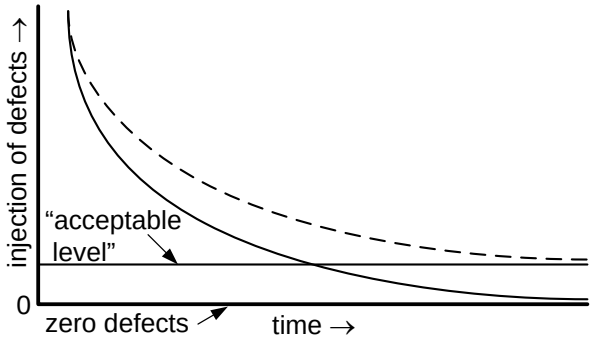


Figure 1: Zero Defects is an asymptote

9 Attitude

As long as we are convinced that defect free software is impossible, we will 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. 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 immediately prevents half of the defects being made. 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.

10 Plan-Do-Check-Act

I assume the Plan-Do-Check-Act (PDCA- or Deming-) cycle [7] is well known (Figure 2, next page). 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 then *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

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, and 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 allocable memory and the whole problem could have been avoided. But changing this architectural decision wasn't a viable option at this stage anymore.

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 anymore 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 analyse the problem and solve it. The attitude was what made the difference.

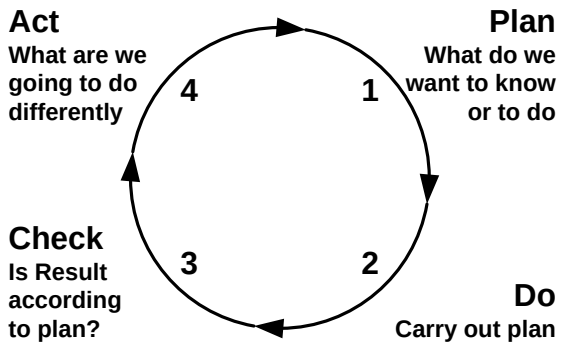


Figure 2: PDCA or Deming cycle

The key-ingredients are: planning before doing, systematically checking and above all *acting*: doing something differently. After all, if you don't do things differently, you shouldn't expect a change 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.

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 moulded 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 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.

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

- 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.

So, evaluations are good, but they must be tuned to the right cycle time to make them really useful. The same applies to testing, as this is also a type of evaluation.

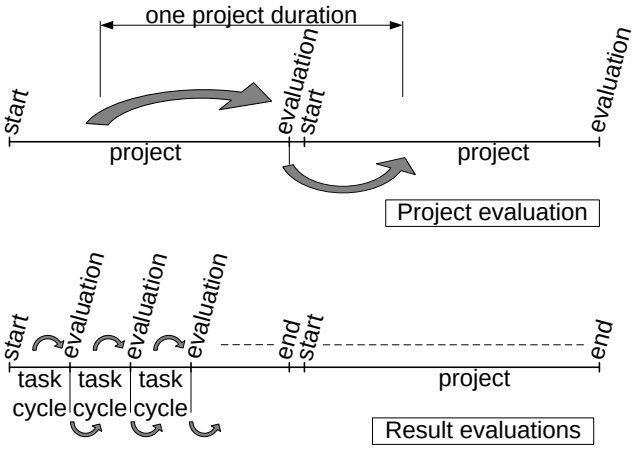


Figure 3: Project and Result evaluations

12 Current Evo Testing

Conventionally, a lot of testing is still executed in Waterfall mode, after the *Code Complete* milestone. I have difficulty understanding this "Code Complete", while apparently the code is not complete, witness the planned "debugging" phase after this milestone. Evo projects do not need a separate debugging phase and hardly need repair after delivery. If code is complete, it is complete. Anything is only ready if it is completely done, *not to worry about it anymore*. That includes: no defects. I know we are human and not perfect, but remember the importance of attitude: we *want* to be perfect (note that perfection means: 'exactly as it should be'. It does not mean: 'gold plating').

Because we regularly deliver results, testers can test these intermediate results (Figure 4). They feed back their findings, which we will use for prevention or optimization. Most issues that are not caught by the testers (I suppose testers are human as well) may be found in subsequent deliveries. 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. Because all people in the project aim for Zero Defects delivery, the developers and testers work together in their quest for perfection.

13 Further improvement

To further improve the results of the projects, we can extend the Evo techniques to the testing process and exploit the PDCA paradigm even further:

- Testers focus on a clear goal. Finding defects is not the goal. After all, we don't want defects. Any defects found are only a means to achieve the real goal: the success of the project
- Testers 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
- Testers check work in progress *even before* it is delivered to them, feeding back issues found, helping the developer preventing producing similar issues for the remainder of his 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

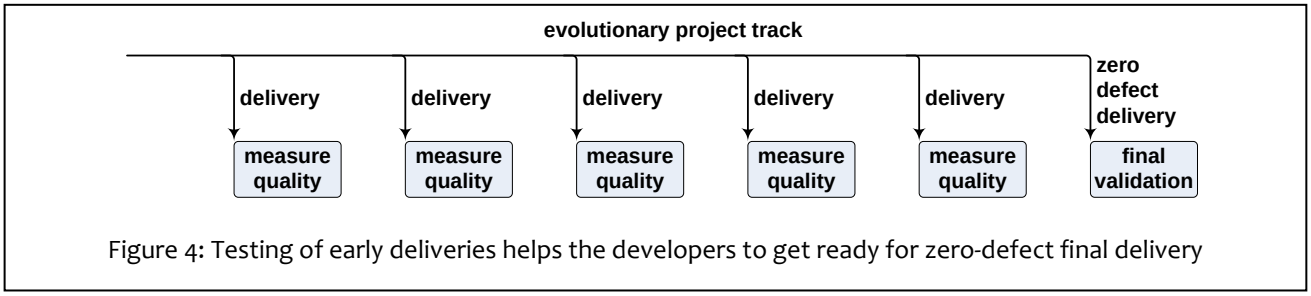


Figure 4: Testing of early deliveries helps the developers to get ready for zero-defect final delivery

find, if I find anything". Testers 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. Testers can help by asking. Initially the developers may seem a little surprised, but this will soon fade

- Similarly, testers 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 the testers in the process, the testers will 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 testing the Evo way means entangling the testing process more intimately with the development process

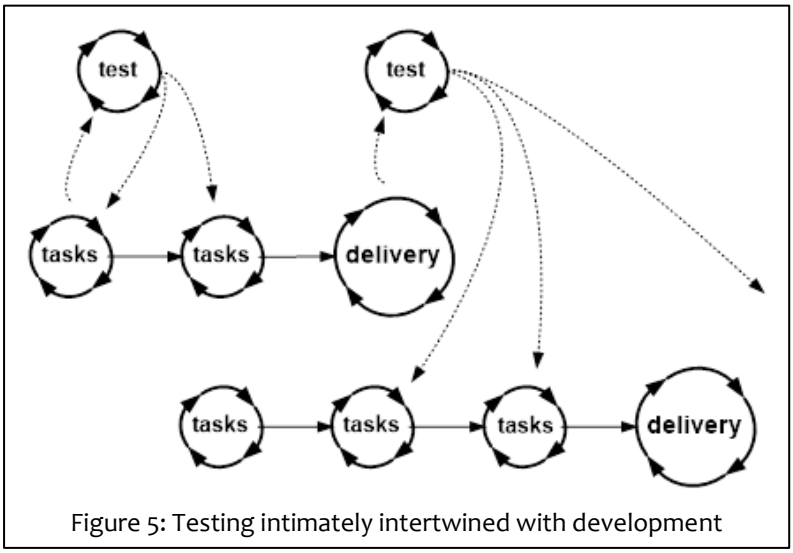


Figure 5: Testing intimately intertwined with development

14 Cycles in Evo

In the Evo development process, we use several learning cycles:

- The TaskCycle [9] 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 [10] 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 [11] is used to keep control over the project duration. We optimize the order of DeliveryCycles 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 [12]: how to arrive at the best product (according to the goal).
- The project [13]: how to arrive at this product most effectively and efficiently.
- The process [14]: 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 testing adds cycles for feedback from testing to development, as well as cycles for organizing and optimizing the testing activities themselves (Figure 5):

- Testers organize their work in weekly, or even shorter TaskCycles
- The DeliveryCycle of the testers is the Test-feedback cycle: in very short cycles testers take intermediate results from developers, check for defects in all varieties and feedback 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
- The Testers 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 testers 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

Testers are part of the project and participate in the weekly 3-step procedure [15] 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 Test Manager, for the testing 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.

Testers participate in requirements discussions. They communicate with developers in the unplannable time [16], 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 [17] will be used. If something is unclear, an Analysis Task [18] will be planned. The Prevention Potential of issues found is an important factor in the prioritizing process.

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

In Evo Testing, 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. Testers checking the result and feeding back any imperfections before more imperfections are generated, closing the very short feedback loop. Developers and testers 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.

Doesn't this process 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 fewer 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 and 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, this database will, for every RI/CR/PR, have to provide additional space for the collection of data to specifically support the prevention process, like:

- Follow-up status
- When and where 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 [19]
- 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 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 and repair tasks should be separated, because analysis usually has 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 [20], 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 tasks, *if* we are really serious about time.
Note that exact task estimates are not required. Planning at least 5 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 by the end of the week. As soon as people learn not to overrun their (average) estimates anymore, there is no need to track or record overrun metrics. The attitude replaces the need for the metric.

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 "Fatal-Date". 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. Testers use TimeLine to control that they will complete whatever they have to do in the project, in sync with the developers.

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

- ***Defects-per-kLoC* or *Defects-per-page***
  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 testers to find them all. Results in practice are in the range of 30% to 80%. Testers 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 and fix a defect***
  The less defects there are, the higher the cost to find and fix the few defects that slip through from time to time, because we still have to test, to see that the result is OK. This was a bad metric anyway
- ***Closed defects per month*** or ***Age of open customer found defects***
  Whether and how a defect is closed or not, depends on the prioritizing process. Every week any problems 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 bugs 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 debugging 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 testing being done

Instead of *improving* non-value adding activities, including various types of metrics, it is better to *eliminate* them. In many cases, 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 software 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 even 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?

Moore's Law is still valid, implying that the complexity of our systems is growing exponentially, and the capacity needed to fill these systems with meaningful software is growing exponentially 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, fear not: even if we start aiming at defect free software, we'll still have to learn a lot from the mistakes we'll still be making.

Dijkstra [8] said:

> It is a usual technique to make a program and then to test it. But: program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence.

Where we first pursued the very effective way to show the presence of bugs, testing will now have to find a solution for the hopeless inadequacy of showing their absence. That is a challenge as well.

I invite testers from now on to change their focus from finding defects, to working with the developers to minimize the generation of defects in order to satisfy the real goal of software development projects. Experience in many projects shows that this is not an utopia, but that it can readily be achieved, using the Evo techniques described.

References

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

[2] N.R. Malotaux: *How Quality is Assured by Evolutionary Methods*, 2004.
PNSQC 2004 Proceedings.
Also downloadable as a booklet:
www.malotaux.nl/booklets - booklet#2

[3] N.R. Malotaux: *Evolutionary Project Management Methods*, 2001.
www.malotaux.nl/booklets - booklet#1

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

[5] See cases:
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] E. Dijkstra: Lecture: *The Humble Programmer*, 1972.
Reprint in *Classics in Software Engineering*.
Yourdon Press, 1979, ISBN 0917072146.

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

Niels Malotaux

Optimizing the Contribution of Testing to Project Success

Let's define the Goal of development 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 software. If there are no defects, we will have achieved our goal. If there are defects, we failed. We know all the stories about failed and partly failed projects, only about one third of the projects delivering according to the original goal.

Apparently, despite all the efforts for doing a good job, too many defects are generated by developers, and too many remain undiscovered by testers, 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 software development. A solution is 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 software development organizations. In 2004, I published a booklet: *How Quality is Assured by Evolutionary Methods*, describing practical implementation details of how to organize projects using this knowledge, making the project a success.
In this booklet we'll extend these Evolutionary methods to the testing process, in order to optimize the contribution of testing 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.

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.nl/conferences

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

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

3_a. 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!

ETA: Evo Task Administration tool - www.malotaux.nl/?id=downloads#ETA

N R Malotaux
Consultancy

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

Originally prepared for the Annual Pacific Northwest Software Quality Conference, Portland, USA, 2005
Version 1.12 (reformatted, links checked) – 27 Aug 2018