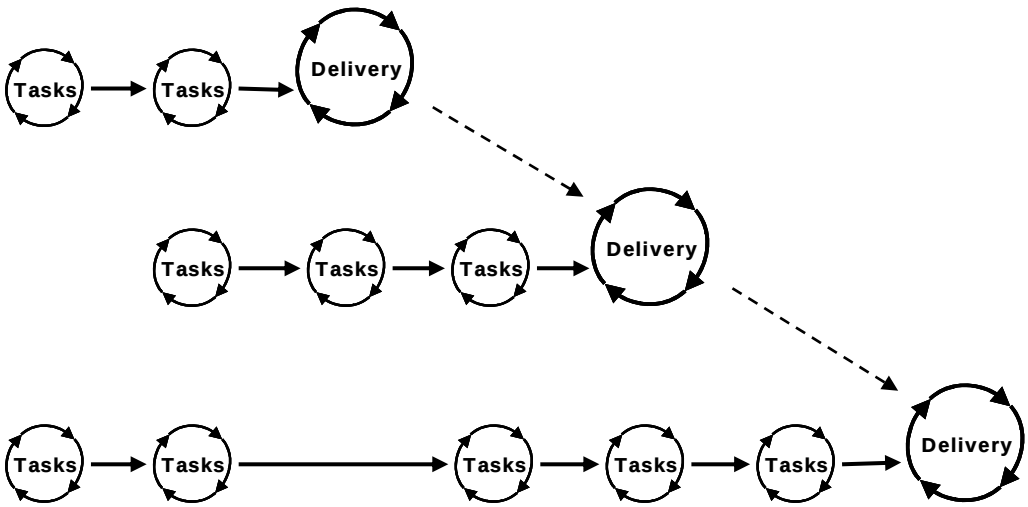
Niels Malotaux

Evolutionary Project Management Methods

How to deliver *Quality On Time* in Software Development and Systems Engineering Projects



Niels Malotaux

Evolutionary Project Management Methods

How to deliver *Quality On Time* in Software Development and Systems Engineering Projects

Software developers systematically fail to manage projects within the constraints of cost, schedule, functionality and quality. Solutions have been developed during the past 35 years, with important results published already some 15 years ago. Still, in practice not much has changed. The challenge is to find ways to catch the practical essence of solutions and ways to get the developers to use these solutions.

In this booklet, we show methods and techniques, which do enable software developers and management to successfully managing projects within the constraints of cost, schedule, functionality and quality. These methods are taught and coached in actual development projects with remarkable results: typically, projects can be done in 30% shorter time.

While software development results were usually delivered late, the delays in other disciplines (like hardware and mechanical development) seemed to be non-existent. Now that we have taught Software Development to deliver Quality On Time (the right results at the right time and within budget), the delays in the other disciplines become exposed. The methods and techniques described in this booklet are obviously not limited to just software development. For those projects where delivering Quality On Time is important it is about time that we are going to apply the techniques at the Systems Development level. Therefore the next target for Evolutionary Development Methods will be Systems Engineering.

Note that the contents of this booklet describe ongoing developments. The methods are being used by the author with various clients and are continuously being optimised based on results found.

Keywords – **Evolutionary delivery, Software Process Improvement, Project management, Development, Risk management, On Time delivery, Systems Engineering.**

1. INTRODUCTION

Software developers systematically fail to manage projects within the constraints of cost, schedule, functionality and quality. More than half of ICT users still is not content with the performance of ICT suppliers [Ernst&Young, 2001]. This is known for some 35 years. Solutions have been developed during the past 35 years, with impressive results published already years ago (e.g. Mills, 1971 [1], Brooks, 1987 [2], Gilb, 1988 [3]). Still, in practice not much has changed. An important step in solving this problem is to accept that *if developers failed to improve their habits*, in spite of the methods presented in the past, *there apparently are psychological barriers in humans, preventing adoption of these methods*. The challenge is to find ways to catch the *practical essence of the solutions* to manage projects within the constraints of cost, schedule, functionality and quality and *ways* to get the developers to use these solutions.

The importance of solving the problem is mainly economical:

- Systematically delivering software development results within the constraints of cost, schedule, functionality and quality saves unproductive work, both by the developers and the users (note Crosby, 1996: the Price Of Non-Conformance [4]).
- Prevention of unproductive work eases the shortage of IT personnel.
- Enhancing the quality level of software developments yields a competitive edge.
- Being successful eases the stress on IT personnel, with positive health effects as well as positive productivity effects.

In this booklet, we show methods and techniques, labelled “Evo” (from Evolutionary), which enable software developers and management to deliver “Quality On Time”, which is short for successfully managing projects within the constraints of cost, schedule, functionality and quality. These methods are taught and coached in actual development projects with remarkable results.

The contents is based on practical experiences and on software process improvement research and development and especially influenced by Tom Gilb (1988 [3], later manuscripts [5] and discussions).

2. HISTORY

Most descriptions of development processes are based on the Waterfall model, where all stages of development follow each other (Figure 1). Requirements must be fixed at the start and at the end we get a Big Bang delivery. In practice, hardly anybody really follows this model, although in reporting to management, practice is bent into this model. Management usually expects this simple model, and most development procedures describe it as mandatory. This causes a lot of mis-communication and wastes a lot of energy.

Early descriptions of Evolutionary delivery, then called Incremental delivery, are described by Harlan Mills in 1971 [1] and F.P. Brooks in his famous "No silver bullet" article in 1987 [2]. Evolutionary delivery is also used in Cleanroom Software Engineering [6]. A practical elaboration of Evolutionary development theory is written by Tom Gilb in his book "Principles of Software Engineering Management" in 1988 [3] and in newer manuscripts at Tom Gilb's web-site [16]. Incremental delivery is also part of eXtreme Programming (XP) [15, 17], however, if people claim to follow XP, we hardly see the Evo element practiced as described here.

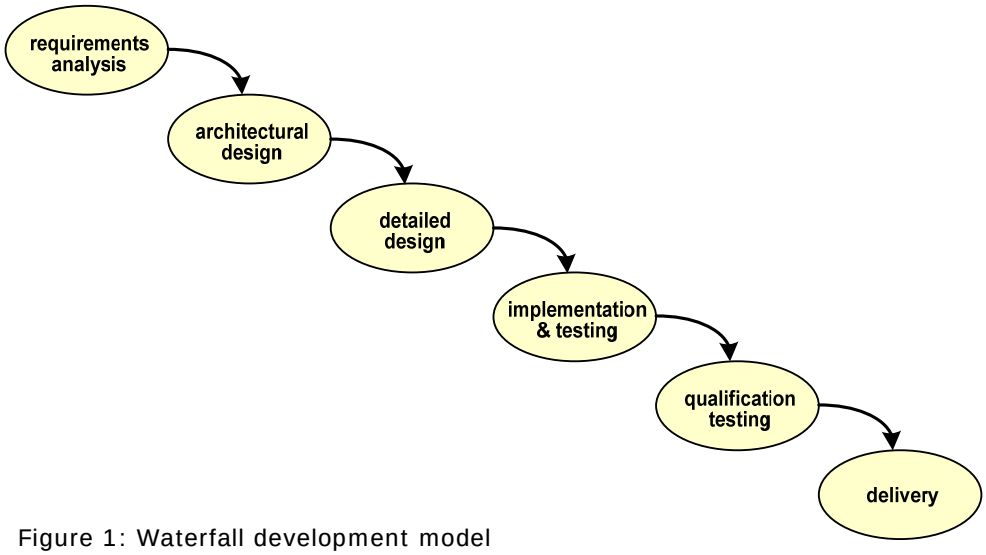


Figure 1: Waterfall development model

We prefer using the expression Evolutionary delivery, or Evo, as proposed by Tom Gilb, because not all Incremental delivery is Evolutionary. Incremental delivery methods use cycles, where in each cycle, part of the design and implementation is done. In practice this still leads to Big Bang delivery, with a lot of debugging at the end. We would like to reserve the term Evolutionary for a special kind of Incremental delivery, where we address issues like:

- Solving the requirements paradox.
- Rapid feedback of estimation and results impacts.
- Most important issues first.
- Highest risks first.
- Most educational or supporting issues for the development first.
- Synchronising with other developments (e.g. hardware development).
- Dedicated experiments for requirements clarification, before elaboration is done.
- Every cycle delivers a useful, completed, working, functional product.
- At the fatal end day of a project we should rather have 80% of the (most important) features 100% done, than 100% of all features 80% done. In the first case, the customer has choice to put the product on the market or to add some more bells and whistles. In the latter case, the customer has no choice but to wait and grumble.

In Evolutionary delivery, we follow the waterfall model (Figure 1) repeatedly in very short cycles (Figure 2).

3. Issues Addressed by Evo

A. Requirements Paradoxes

The **1st Requirements Paradox** is:

- Requirements must be stable for reliable results.
- However, the requirements always change.

Even if you did your utmost best to get complete and stable requirements, they will change. Not only because your customers change their mind when they see emerging results from the developments. Also the developers themselves will get new insights, new ideas about what the requirements should really be. So, requirements change is a *known risk*. Better than ignoring the requirements paradox, use a development process that is designed to cope with it: Evolutionary delivery.

Evo uses rapid and frequent feedback by stakeholder response to verify and adjust the requirements to what the stakeholders really need most. Between cycles there is a short time slot where stakeholders input is allowed *and requested* to reprioritise the list.

This is due to the **2nd Requirements Paradox**:

- We don't want requirements to change.
- However, because requirements change now is a *known risk*, we try to *provoke* requirements change as early as possible

We solve the requirements paradoxes by creating stable requirements *during* a development cycle, while explicitly reconsidering the requirements *between* cycles.

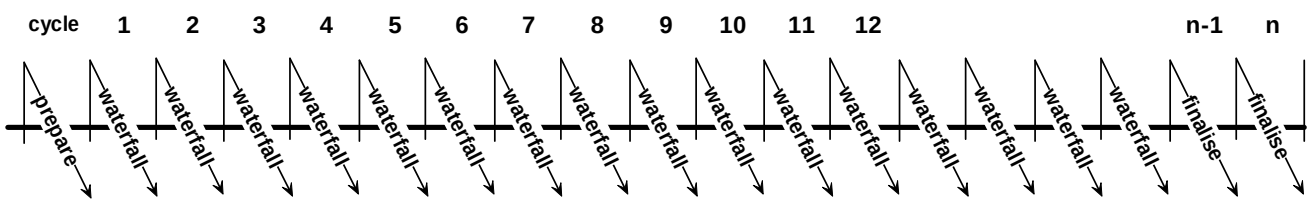


Figure 2: Evolutionary delivery uses many waterfalls

B. Very short cycles

Actually, few people take planned dates seriously. As long as the end date of a project is far in the future (Figure 3), we don't feel any pressure and work leisurely, discuss interesting things, meet, drink coffee, ... (How many days before your last exam did you really start working...?). So at the start of the project we work

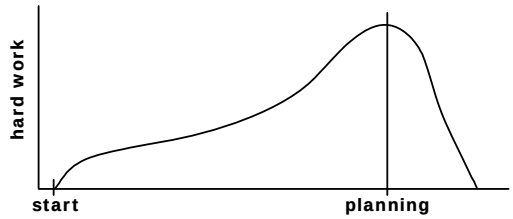


Figure 3: We only start working harder when the pressure of the delivery date is near. Usually we are late.

relatively slowly. When the pressure of the finish date becomes tangible, we start working harder, stressing a bit, making errors causing delays, causing even more stress. The result: we do not finish in time. We know all

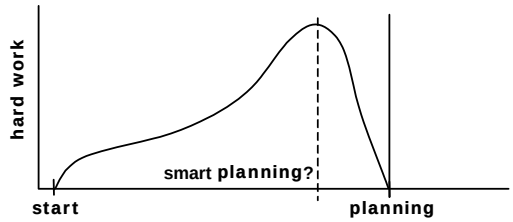


Figure 4: To overcome the late delivery problem, a smart project manager sells his team an earlier delivery date. Even smarter developers soon will know.

the excuses, which caused us to be late. It's never our own fault. This is not wrong or right. It's human psychology. That is how we function. So don't ignore it. Accept it and then think what to do with it.

Smart project managers tell their team an earlier date (Figure 4). If they do this cleverly, the result may be just in time for the real date. The problem is that they can do this only once or twice. The team members soon will discover that the end date was not really hard and they will loose faith in milestone dates. This is even worse.

The solution for coping with these facts of human psychology is to plan in very short increments (Figure 5). The duration of these increments must be such that:

- The pressure of the end date is felt right the first day.
- The duration of a cycle must be sufficient to finish real tasks.

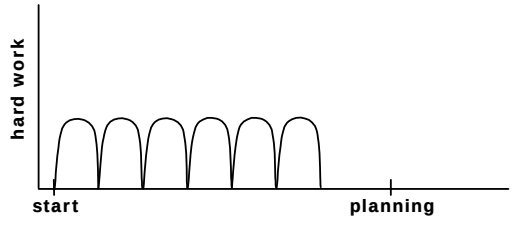


Figure 5: The solution: choose short, realistic "delivery dates". Satisfaction, motivation, fast feedback.

Three weeks is too long for the pressure and one week may be felt as too short for finishing real tasks. Note that the pressure in this scheme is much healthier than the real stress and failure at the end of a Big Bang (delivery at once at the end) project. The experience in an actual project, where we got only six weeks to finish completely, led to using one-week cycles. The results were such, that we will continue using one-week cycles on all subsequent projects. If you cannot even plan a one-week period, how could you plan longer periods ...?

C. Rapid and frequent feedback

If everything would be completely clear we could use the waterfall development model. We call this *production* rather than development. At the start of a new development, however, there are many uncertainties we have to explore and to change into certainties. Because even the simplest development project is too complex for a human mind to oversee completely (E. Dijkstra, 1965: "The competent programmer is fully aware of the limited size of his own skull" [12]) we must iteratively learn what we are actually dealing with and learn how to perform better.

This is done by "think first, then do", because thinking costs less than doing. But, because we cannot foresee everything and we have to assume a lot, we constantly have to check whether our thoughts and assumptions were correct. This is called feedback: we plan something, we do it as well as we can, then we check whether the effects are correct. Depending on this analysis, we may change our ways and assumptions. Shewhart already described this in 1939 [13]. Deming [14] called it the Shewhart cycle (Figure 6). Others call it the Deming cycle or PDCA (Plan-Do-Check-Act) cycle.

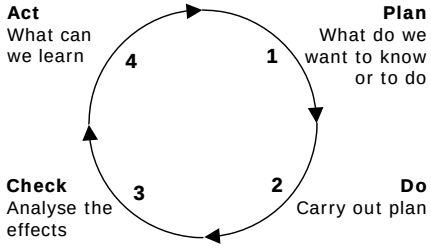


Figure 6: Shewhart cycle, Deming cycle, PDCA cycle.

In practice we see that if developers do something (section 2 of the cycle), they sometimes plan (section 1), but hardly ever explicitly go through the analysis and learn sections. In Evo we do use all the sections of the cycle deliberately in rapid and frequent feedback loops (Figure 7, next page):

- *The weekly task cycle*
  In this cycle we optimise our estimation, planning and tracking abilities in order to better predict the future. We check constantly whether we are *doing* the right things in the right order to the right level of detail for the moment.
- *The frequent stakeholder value delivery cycle*
  In this cycle we optimise the requirements and check our assumptions. We check constantly whether we are *delivering* the right things in the right order to the

right level of detail for the moment. Delivery cycles may take 1 to 3 weekly cycles.

- *The strategic objectives cycle*
  In this cycle we review our strategic objectives and check whether what we do still complies with the objectives. This cycle may take 1 to 3 months.
- *The organisation roadmap cycle*
  In this cycle we review our roadmap and check whether our strategic objectives still comply with what we should do in this world. This cycle may take 3 to 6 months.

In development *projects*, only task cycles and delivery cycles are considered. In a task cycle, tasks are done to feed the current delivery, while some other tasks may be done to make future deliveries possible (Figure 8).

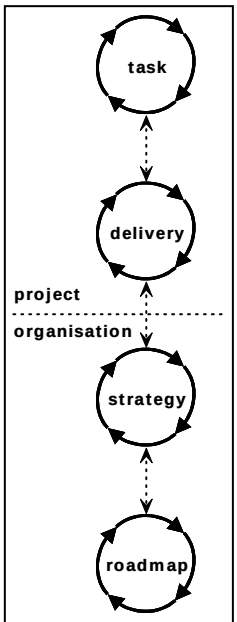


Figure 7: Cycles in Evo

D. Time Boxing

Evolutionary project organisation uses *time boxing* rather than *feature boxing.* If we assume that the amount of resources for a given project is fixed, or at least limited, it is possible to realise either:

- A fixed set of features in the time needed to realise these features. We call this *feature boxing*.
- The amount of features we can realise in a fixed amount of time. We call this *time boxing*.

To realise a fixed set of features in a fixed amount of time with a given set of resources is only possible if the time is sufficient to realise all these features. In practice, however, the time allowed is usually insufficient to realise all the features asked: *What the customer wants, he cannot afford.* If this is the case, we are only fooling ourselves trying to accomplish the impossible (Figure 9). This has nothing to do with lazy or unwilling developers: if the time (or the budget) is insufficient to realise all the required features, they *will not all be realised*. It is as simple as that.

The Evo method makes sure that the customer gets the most and most important features *possible* within a certain amount of time and with the available resources. Asking developers to accomplish the impossible is one of the main energy drains in projects. By wasting energy the result is always less than otherwise possible.

In practice, time boxing means:

- A set number of hours is reserved for a task.
- At the end of the time box, the task should be 100% done. That means really *done*.
- Time slip is not allowed in a time box, otherwise other tasks will be delayed and this would lead to uncontrolled delays in the development.
- Before the end of the time box we check how far we can finish the task. If we foresee that we cannot finish a task, we should define what we know now, try to define what we still have to investigate, define tasks and estimate the time still needed. Preferably, however, we should try whether we could *go into less detail* this moment, actually finishing the task to a *sufficient* level of detail within the time box. A TaskSheet (details see [8]) is used to define:
  - The goal of the task.
  - The strategy to perform the task.
  - How the result will be verified.
  - How we know for sure that the task is really done (i.e. there is really nothing we have to do any more for this task, we can forget about it).

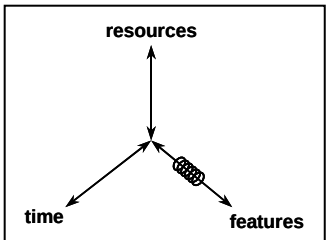


Figure 9: If resources and time are fixed, the features are variable

E. Estimation, planning and tracking

Estimation, planning and tracking are an inseparable trinity*. If you don't do one of them, you don't* need the other two.

- If you don't estimate, you cannot plan and there is nothing to track.
- If you do not plan, estimation and tracking is useless.
- If you do not track, why should you estimate or plan?

So:

- Derive small tasks from the requirements, the architecture and the overall design.
- Estimate the time needed for every small task.
- Derive the total time needed from:
  - The time needed for all the tasks
  - The available resources
  - Corrected for the real amount of time available per resource (nobody works a full 100% of his presence on the project. The statistical average is about 55%. This is one of the key reasons for late projects! [9])

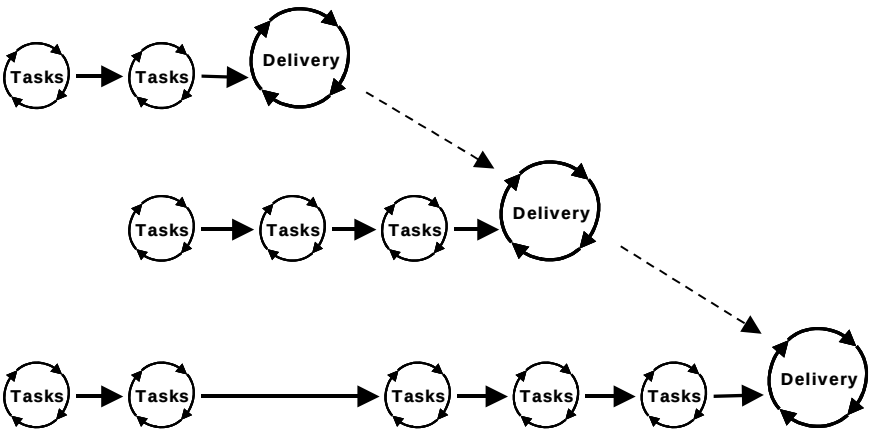


Figure 8: Current tasks feed the current delivery cycle, as well as prepare for future delivery cycles

- Plan the next cycle exactly.
- Be sure that the work of every cycle can be done. That means *really* done. Get commitment from those who are to do the real work.
- Plan the following cycles roughly (the planning may change anyway!).
- Track successes and failures. Learn from it. Refine estimation and planning continuously. Warn stakeholders *well in advance* if the target delivery time is changing because of *any* reason.
- There may be various target delivery times, depending on various feature sets.

If times and dates are not important to you (or to management), then don't estimate, plan, nor track: you don't need it. However, if timing is important, *insist* on estimation, planning and tracking. And it is not even difficult, once you get the hang of it.

If your customer (or your boss) doesn't like to hear that you cannot exactly predict which features will be in at the fatal end day, while you *know* that not all features will be in (at a fixed budget and fixed resources), you can give him two options:

- Either to tell him the day before the fatal day that you did not succeed in implementing all the functions.
- Or tell him now (because you already know), and let him every week decide with you which features are the most important.

It will take some persuasion, but you will see that within two weeks you will work together to get the best possible result. There is one promise you can make: The process used is the most efficient process available. In any other way he will never get more, probably less. So let's work together to make the best of it. Or decide at the beginning to add more resources. Adding resources later evokes Brooks Law [9]: "Adding people to a late project makes it later". Let's stop following ostrich-policy, face reality and deal with it in a realistic and constructive way.

F. Difference between effort and lead-time

If we ask software developers to estimate a given task in days, they usually come up with estimates of lead-time. If we ask them to estimate a task in hours, they come up with estimates in effort. Project managers know that developers are optimistic and have their private multiplier (like 2, $\sqrt{2}$, e or π) to adjust the estimates given. Because these figures then have to be entered in project-planning tools, like MS Project, they enter the adjusted figures as lead-time.

The problem with lead-time figures is that these are a mix of two different time components:

- Effort, the time needed to do the work
- Lead-time, the time until the work is done. Or rather Lead-time minus Effort, being the time needed for other things than the work to be done. Examples of "other things" are: drinking coffee, meetings, going to the lavatory, discussions, helping colleagues, telephone calls, e-mail, dreaming, etc. In practice we use the Effort/Lead-time ratio, which is usually in the range of 50-70% for full-time team members.

Because the parameters causing variation in these two components are different, they have to be kept apart and treated differently. If we keep planning only in lead-time, we will never be able to learn from the tracking of our planned, estimated figures. Thus we will never learn to predict development time. If these elements are kept separately, people can learn very quickly to adjust their effort estimating intuition. In recent projects we found: first week: 40% of the committed work done, second week: 80% done, from the third week on: 100% or more done. Now we can start predicting!

Separately, people can learn time management to control their Effort/Lead-time ratio. Brooks indicated this already in 1975 [9]: *Programming projects took about twice the expected time. Research showed that half of the time was used for activities other than the project*.

In actual projects, we currently use the rule that people select 2/3 of a cycle (26 hours of 39) for project tasks, and keep 1/3 for other activities. Some managers complain that if we give about 3 days of work and 5 days to do the work, people tend to "Fill the time available". This is called Parkinson's Law [10]: "Work expands so as to fill the time available for its completion". Management uses the same reasoning, giving them 6 days of work and 5 days to do it, hoping to enhance productivity. Because 6 days of effort *cannot* be done in 5 days and people have to do, and *will* do, the other things anyway, people will always *fail to succeed* in accomplishing the impossible. What is worse: this causes a constant sense of failure, causing frustration and demotivation. If we give them the amount of work they can accomplish, they will succeed. This creates a sensation of accomplishment and success, which is very motivating. The observed result is that giving them 3 days work for 5 days is *more productive* that giving them 6 days of work for 5 days.

G. Commitment

In most projects, when we ask people whether a task is done, they answer: "Yes". If we then ask, "Is it really done?", they answer: "Well, almost". Here we get the effect that if 90% is done, they start working on the other 90%. This is an important cause of delays. Therefore, it is imperative that we define when a task is really 100% done and that we insist that any task be 100% done. Not 100% is *not* done.

In Evo cycles, we ask for tasks to be 100% done. *No need to think about it any more*. Upon estimating and planning the tasks, effort hours have been estimated. Weekly, the priorities are defined. So, every week, when the project manager proposes any team member the tasks for the next cycle, he should never say "Do this and do that". He should always propose: "Do you still agree that these tasks are highest priority, do you still agree that you should do it, and do you still agree with the estimations?". If the developer hesitates on any of these questions, the project manager should ask why, and *help the developer* to *re-adjust* such that he can give a *full commitment* that he will accomplish the tasks.

The project manager may help the developer with suggestions ("Last cycle you did not succeed, so maybe you were too optimistic?"). He may *never* take over the

responsibility for the decision on which tasks the developer accepts to deliver. This is the only way to get true developer commitment. At the end of the cycle the project manager only has to use the mirror. In the mirror the developer can see himself if he failed in fulfilling his commitments. If the project manager decided what had to be done, the developer sees right through the mirror and only sees the project manager.

It is essential that the project manager coaches the developers in getting their commitments right. Use the sentence: "Promise me to do nothing, as long as *that* is 100% done!" to convey the importance of *completely done*. Only when working with real commitments, developers can learn to optimise their estimations and deliver accordingly. Else, they will *never* learn. Project managers being afraid that the developers will do less than needed and therefore giving the developers more work that they can commit to, will never get what they hope for because without real commitment, people tend to do less.

H. Risks

If there are no risks whatsoever, use the waterfall model for your development. If there are risks, which is the case in any new development, we have to constantly assess how we are going to control these risks. Development is for an important part risk-reduction. If the development is done, all risks should have been resolved. If a risk turns out for worse at the end of a development, we have no time to resolve it any more. If we identify the risks earlier, we may have time to decide what to do if the risk turns out for worse. Because we develop in very short increments of one week the risk that an assumption or idea consumes a lot of development time before we become aware that the result cannot be used is limited to one week. Every week the requirements are redefined, based upon what we learnt before.

Risks are not limited to assumptions about the product requirements, where we should ask ourselves:

- Are we developing the right things right?
- When are things right?

Many risks are also about timing and synchronisation:

- Can we estimate sufficiently accurate?
- Which tasks are we forgetting?
- Do we get the deliveries from others (hardware, software, stakeholder responses, …) in time?

Actually the main questions we are asking ourselves systematically in Evo are: *What* should we do, in *which order*, to *which level of detail* for *now*. Too much detail too early means usually that the detail work has to be done over and over again. May be the detail work was not done wrong. It only later turns out that it should have been done differently.

I. Team meetings

Conventional team meetings usually start with a round of excuses, where everybody tells why he did not succeed in what he was supposed to do. There is a lot of discussion about the work that was supposed to be done, and when the time of the meeting is gone, new tasks are hardly discussed. This is not a big problem, because most participants have to continue their unfinished work anyway. The project manager notes the new target dates of the delayed activities and people continue their work. After the meeting the project manager may calculate how much reserve ("slack time") is left, or how much the project is delayed if all reserve has already been used. In many projects we see that project-planning sheets (MS Project) are mainly used as wallpaper. They are hardly updated and the actual work and the plan-on-the-wall diverge more and more every week.

In the weekly Evo team meeting, we only discuss new work, never past work. We do not waste time for excuses. What is past we cannot change. What we still should do is constantly re-prioritised, so we always work on what is best from this moment. We don't discuss past tasks because they are finished. If discussion starts about the new tasks, we can use the results in our coming work. That can be useful. Still, if the discussion is between only a few participants, it should be postponed till after the meeting, not to waste the others' time.

J. Magic words

There are several "magic words" that can be used in Evo practice. They can help us to doing the *right things* in the *right order* to the *right level of detail for this moment*.

- **Focus**
  Developers tend to be easily distracted by many important or interesting things. Some things may even really be important, however, not at this moment. Keeping focus at the current priority goals, avoiding distractions, is not easy, but saves time.
- **Priority**
  Defining priorities and only working on the highest priorities guides us to doing the most important things first.
- **Synchronise**
  Every project interfaces with the world outside the project. Active synchronisation is needed to make sure that planned dates can be kept.
- **Why**
  This word forces us to define the reason why we should do something, allowing us to check whether it is the right thing to do. It helps in keeping focus.
- **Dates are sacred**
  In most projects, dates are fluid. Sacred dates means that if you agree on a date, you stick to your word. Or tell well in advance that you cannot keep your word. With Evo you will know well in advance.
- **Done**
  To make estimation, planning and tracking possible, we must finish tasks completely. Not 100% finished is *not* done. This is to overcome the "If 90% is done we continue with the other 90%" syndrome.
- **Bug, debug**
  A bug is a small creature, autonomously creeping into your product, causing trouble, and you cannot do anything about it. Wrong. People make mistakes and thus cause defects. The words *bug* and *debug* are dirty words and should be erased from our dictionary.

By actively learning from our mistakes, we can learn to avoid many of them. In Evo, we actively catch our mistakes as early as possible and act upon them. Therefore, the impact of the defects caused by our mistakes is minimised and spread through the entire project. This leaves a bare minimum of defects at the end of the project. Evo projects do not need a special "debugging phase".

- **Discipline**
  With discipline we don't mean imposed discipline, but rather what you, yourself, know what is best to do. If nobody watches us, it is quite human to cut corners, or to do something else, even if we know this is wrong. We see ourselves doing a less optimal thing and we are unable to discipline ourselves. If somebody watches over our shoulder, keeping discipline is easier. So, discipline is difficult, but we can help each other. Evo helps keeping discipline. Why do we want this? Because we enjoy being successful, doing the right things.

4. How do we use Evo in projects

In our experience, many projects have a mysterious start. Usually when asked to introduce Evo in a project, one or more people have been studying the project already for some weeks or even months. So in most cases, there are some requirements and some idea about the architecture. People acquainted with planning usually already have some idea about what has to be done and have made a conventional planning, based on which the project was proposed and commissioned.

A. Evo day

To change a project into an Evo project, we organise an "Evo day", typically with the Project Manager, the architect, a tester and *all* other people of the development team. Stakeholder attendance can be useful, but is not absolutely necessary at the first Evo day, where we just teach the team how to change their ways. During the Evo day (and during all subsequent meetings) a notebook and a LCD projector are used, so that all participants can follow what we are typing and talking about. It is preferable to organise the Evo day outside the company.

The schedule is normally:

Morning

- Presentation of Evo methods [11]: why and how.
- Presentation of the product by the systems architect (people present usually have different views, or even no view, of the product to be developed).

Afternoon

In the afternoon we work towards defining which activities should be worked on in the coming week/cycle. Therefore we do exercises in:

- Defining sub-tasks of max 26 hours.
  In practice, only few activities will be detailed. People get tired of this within 20 minutes, but they did the exercise and anyway we don't have time to do it all in one day.
- Estimating the effort of the sub-tasks, in effort-hours, never in days, see "Difference between effort and lead-time" above.
- Defining priorities.
- Listing the tasks in order of priority.
- Dividing top-priority activities, which have not yet been divided into sub-tasks.
- Estimating effort on top-priority sub-tasks if not yet done.
- The team decides who should do what from the top of the list.
- Every individual developer decides which tasks he will be able to deliver done, really done at the end of the cycle. If a commitment cannot be given, take fewer tasks, until full commitment can be given.

At the end of the day everyone has a list of tasks for the coming week, and a commitment that these tasks will be finished completely, while we are sure that the tasks we start working on have the highest priority.

B. Last day of the cycle[1]

The last day of a cycle is special and divided into 3 parts (Figure 10, next page):

- The project manager visits every developer individually and discusses the results of the tasks. If the commitments could not be met, they discuss the causes: Was the effort estimation incorrect or was there a time-management problem. The developer should learn from the results to do better the next time. After having visited all developers, the project manager has an overview of the status of the project.
- The status of the project is discussed with the customer, product manager, or whichever relevant stakeholders. Here the Requirements Paradox is handled: during the week, the requirements were fixed, now is the 1 to 2 hours timeslot that the stakeholders may re-arrange the requirements and priorities. At the end of this meeting, the requirements and priorities are fixed again.
- Finally, the project manager defines task-proposals for the developers and discusses these proposals with them individually. Developers agree that these tasks have the highest priority and commit to finishing these tasks during the cycle.

C. Team meeting

Having prepared the individual task-lists for the next cycle, in the team meeting, at the end of the last cycle day, or the beginning of the first new cycle day, the following is done:

- Experience from the past cycle may be discussed if it could benefit subsequent work.
- The status of the project is discussed. Sub-tasks may be (re-)defined and (re-)estimated if full participation is useful.
- The tasks for the next cycle are formally assigned and committed to. Now all participants hear who is going to do what and may react upon it.
- Discussion may be allowed, if it affects most participants.

[1] See new booklet "How Quality is Assured by Evolutionary Methods" for an updated approach of this part.

- The discussions may cause some reprioritisation and thus reshuffling of tasks to be done.

Weekly team meetings typically take less than 20 minutes. A typical reaction at the end of the first Evo team meeting is: "We never before had such a short meeting". When asked "Did we forget to discuss anything important?", the response is: "No, this was a good and efficient meeting". This is one of the ways we are saving time.

5. CHECK LISTS

There are several checklists being used to help defining priorities and to help to get tasks really finished.
These are currently:

A Task prioritisation criteria
B Delivery prioritisation criteria
C Task conclusion criteria

A. Task prioritisation criteria

To help in the prioritisation process of which tasks should be done first, we use the following checklist:

- Most important issues first (based on current and future delivery schedules).
- Highest risks first (better early than late).
- Most educational or supporting activities first.
- Synchronisation with the world outside the team (e.g. hardware needs test-software, software needs hardware for test: will it be there when needed?).
- Every task has a useful, completed, working, functional result.

B. Delivery prioritisation criteria

To help in the prioritisation process of what should be in the next delivery to stakeholders we use the following checklist:

- Every delivery should have the juiciest, most important stakeholder values that can be made in the least time. Impact Estimation [7] is a technique that can be used to decide on what to work on first.
- A delivery must have *symmetrical* stakeholder values. This means that if a program has a start, there must also be an exit. If there is a delete function, there must be also some add function. Generally speaking, the set of values must be a useful whole.
- Every subsequent delivery must show a *clear difference*. Because we want to have stakeholder feedback, the stakeholder must see a difference to feedback on. If the stakeholder feels no difference he feels that he is wasting his time and looses interest to generate feedback in the future.
- Every delivery delivers the *smallest clear increment*. If a delivery is planned, try to delete anything that is not absolutely necessary to fulfil the previous checks. If the resulting delivery takes more than two weeks, try harder.

C. Task conclusion criteria

If we ask different people about the contents of a defined task, all will tell a more or less different story. In order to make sure that the developer develops the right solution, we use a TaskSheet (details see [8]).
Depending on the task to be done, TaskSheets may be slightly different. First, the developer writes down on the TaskSheet:

- The requirements of the result of the task.
- Which activities must be done to complete the task.
- Design approach: how to implement it.
- Verification approach: how to make sure that it *does* what it *should do* and *does not do* what it should *not do*, based on the requirements.
- Planning (if more than one day work). If this is difficult, ask: "What am I going to do the first day".
- Anything that is not yet clear.

Then the TaskSheet is reviewed by the system architect. In this process, what the developer *thinks* has to be done is compared with what the system architect *expects*: will the result fit in the big picture? Usually there is some difference between these two views and it is better to find and resolve these differences *before* the actual execution of the task than *after*. This simply saves time.

After agreement, the developer does the work, verifies that the result produced not less, but also *not more*, than the requirements asked for. *Nice things* are not allowed: Anything not specified in the requirements is not tested. Nobody knows about it and this is an irresolvable and therefore unwanted risk.

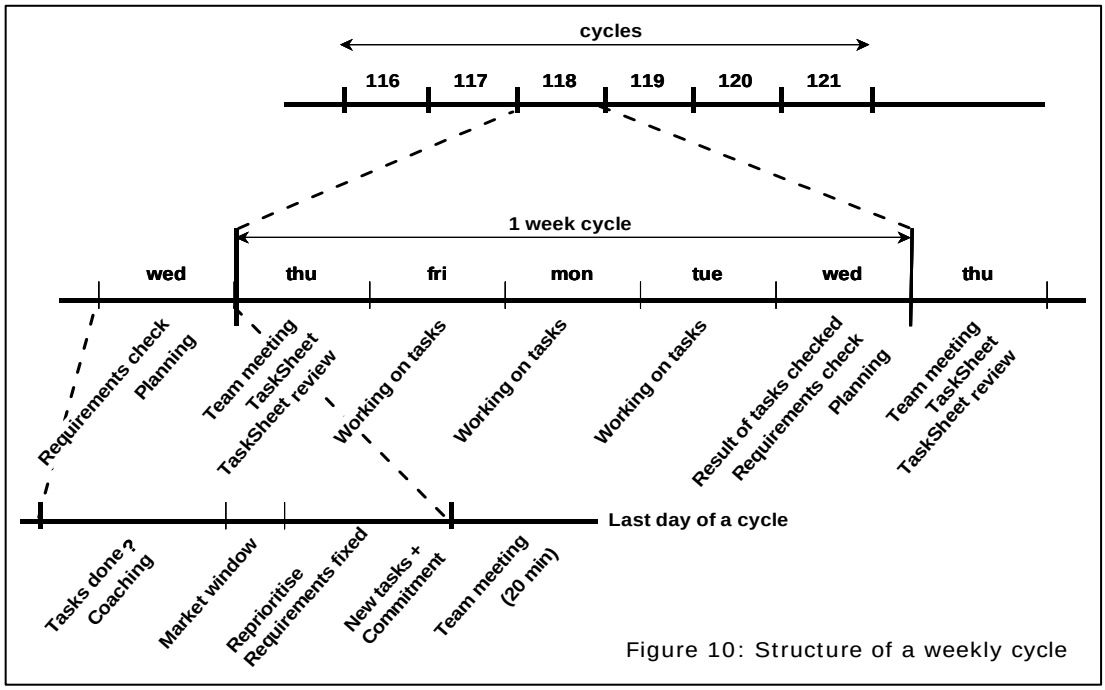


Figure 10: Structure of a weekly cycle

Finally, the developer uses the task conclusion criteria on the TaskSheet to determine that the task is really done. These criteria may be adapted to certain types of tasks. In practical projects, where software code was written we used the following list:

- The code compiles and links with all files in the integration promotion level.
- The code simply does what it should do: no bugs.
- There are no memory leaks.
- Defensive programming measures have been implemented.
- All files are labelled according to the rules agreed.
- File promotion is done.
- I feel confident that the tester will find no problems.

This checklist is to make sure that the task is really done. If all checks are OK, then the work is done. If it later turns out that the work was not completely done, then the checklist is improved.

6. INTRODUCING EVO IN NEW PROJECTS

Many projects where we start introducing Evo are already running. We organise an Evo-day to turn the project into an Evo project. The project has already more or less insight in what has to be done, so this can be estimated, prioritised and selected.

In the case of completely new projects many team members do not yet know what has to be done, let alone how. If team members have no previous Evo experience, they hardly can define tasks and thus estimation and planning of tasks seem hardly possible. Still, the goal of the first Evo day is that at the end of the day the team knows roughly what to do the coming weeks, and exactly what to do the first week. So there is a potential problem.

This problem can be solved by:

- Defining the goal of the project
- Defining the critical success factors and the expectations of the project result from key stakeholders
- Defining what should be done first. What do you think you should be starting on first? Like:
  - Requirements gathering
  - Experiments
  - Collecting information about possible tools, languages, environments
  - Getting to know the selected tools, languages, environments, checking whether they live up to their promise

When we ask how much time the team members are going to spend on these activities, the answer is usually “I don’t know”, “I don’t know what I am going to search for, what I am going to find or going to decide, so I cannot estimate”. This may be true, but should not be used as a licence to freely spend time. Define important things that should be done. Define time boxes, like “Use 10 hours on Requirements collection”, or “Use 8 hours to draw up a tool inventory”. Then put these tasks on the list of Candidate Tasks, define priorities and let everybody take 26 hours of the top of the list, get commitments and that’s it. Then, the next week, based on the findings of the first week, the team already is getting a better idea of what really has to be done. The “fuzzy front end” of projects usually eats up a lot of project time, because the team lacks focus in defining what really has to be done in the project. Evo helps to keep focus and to quickly learn, by evolutionary iterations, what the project really is about.

Still, in some cases the team members cannot set their mind to commit to not-so-clear tasks within a time box. Then, but only as a last resort, team members may do whatever they want, provided that during the work they record what they are doing and how long. This is learning material for the next weeks' meeting. Note that especially if the task is not so clear, it is better first to make it clearer, before spending too much time on it.

These problems can be avoided if we start the new project with people who already have worked the Evo way. Then they know why and how to define tasks, define time boxes, set priorities and finish tasks. This enables them to efficiently start any project, without constantly asking why it has to be done this way. They *know* why and how. Starting using Evo at a completely new project adds up two challenges: learning what the project is all about and learning Evo. It is easier to start learning Evo on a running project, because then the project is already known and only Evo has to be added. However, if there is no Evo experience available when starting a new project, it is still advisable to start using Evo even then, simply because it will lead to better results faster. In this case, a good coach is even more needed to make Evo succeed the first time.

7. TESTING WITH EVO

When developing the conventional way, testing is done at the end of the development, after the Big Bang delivery. Testers then tend to find hundreds of defects, which take a long time to repair. And because there are so many defects, these tend to influence each other. Besides, repairing defects causes more defects.

Software developers are not used to using statistics. If we agree that testing never covers 100% of the software, this means that testing is *taking a sample*. At school we learnt that if we sample, we should use statistics to say something about the whole. So we should get used to statistics and not run away from it.

Statistics tell us that testing is on average 50% effective. Until you have your own (better?) figures, we have to stick to this figure. This means that the user will find the same amount of defects as found in test. Paradoxically this means that the more defects we find in test, the more the user will find. Or, if we do not want the user to find any defects, the test should find no defects *at all*. Most developers think that defect-free software is impossible. If we extrapolate this, it means that we think it is quite normal that our car may stop after a few kilometres drive. Or that the steering wheel in some cases works just the other way: the car turns to the left when we steered to the right… Is that normal?

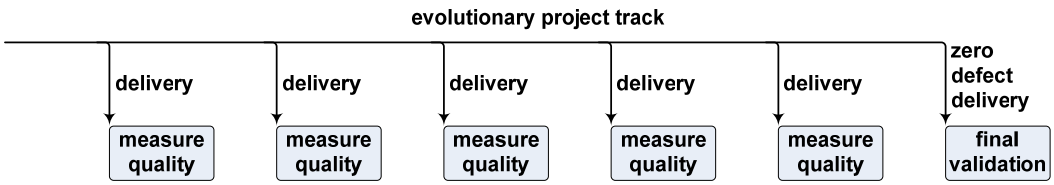
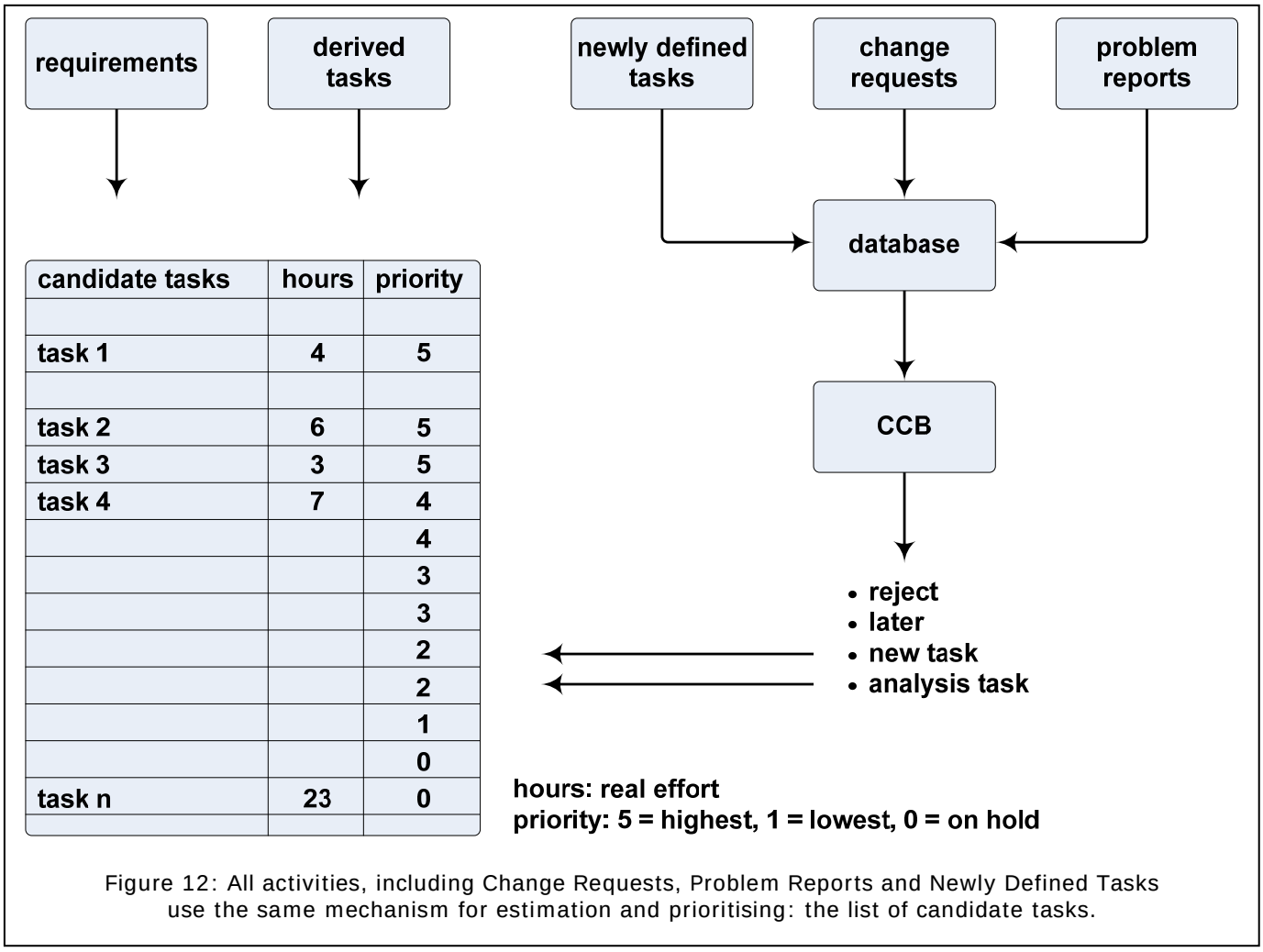


Figure 11: Testing of early deliveries helps the developers to get ready for zero-defect final delivery.

In Evo, we expect the developers to deliver zero-defect results for the final validation, so that the testers just have to check that everything works OK, as required. Although software developers usually start laughing by this very idea, we are very serious about this. The aim of testing earlier deliveries of Evo cycles is not just testing whether it "works". Also, testing is not to make life difficult for the developers. In Evo, the software developers ask the testers to help them to find out how far the developers are from the capability of delivering a defect free product at, or before, final validation (Figure 11).

8. Change Requests and Problem Reports

Change Requests (CR) are requested changes in the requirements. Problems Reports (PR) report things found wrong (defects), which we should have done right in the first place. Newly Defined Tasks (NT) are tasks we forgot to define. If any of these is encountered, we never start just changing, repairing, or doing the new task. We work only on defined tasks, of which the effort has been estimated and the priority defined. All tasks are listed on the list of candidate tasks in order of priority. Any CR, PR or NT is first collected in a database. This could be anything between a real database application and a notebook. Regularly, the database is analysed by a Change Control Board (CCB). This could be anything between a very formal group of selected people, who can and must analyse the issues (CRs, PRs and NTs), and an informal group of e.g. the project manager and a team member, who check the database and decide what to do. The CCB can decide to ignore or postpone some issues, to define a new task immediately or to define an analysis task first (Figure 12). In an analysis task, the consequences of the issue are first analysed and an advice is documented about what to do and what the implications are. Any task generated in this process is put on the list of candidate tasks, estimated and prioritised. And only when an existing or new task appears at the top of the candidate list, it will be worked on.



candidate tasks	hours	priority
task 1	4	5
task 2	6	5
task 3	3	5
task 4	7	4
		4
		3
		3
		2
		2
		1
		0
task n	23	0

Figure 12: All activities, including Change Requests, Problem Reports and Newly Defined Tasks use the same mechanism for estimation and prioritising: the list of candidate tasks.

9. TOOLS

Special tools may only be used when we know and understand the right methods. In actual projects, we have used MS Excel as an easy notepad during interactive sessions with a LCD projector showing what happens on this notepad in real time. When tasks have been defined, MS Project can be used as a spreadsheet to keep track of the tasks per person, while automatically generating a time-line in the Gantt-chart view (Figure 13, top left). This time-line tells people, including management, more than textual planning. It proved possible to let MS Project use weeks of 26 hours and days of 5.2 hours, so that durations could be entered in real effort while the time-line shows correct leadtime-days.

There is a relation between requirements, stakeholder values, deliveries and tasks (Figure 13). We even want to have different views on the list of tasks, like a list of prioritised candidate tasks of the whole project and lists of prioritised tasks per developer. This calls for the use of a relational database, to organise the relations between requirements, values, deliveries and tasks and the different views. Currently, such a database has not been made and the project manager has to keep the consistency of the relations manually. This is some extra work. However, in the beginning it helps the project manager knowing what he is doing.

Recently we did introduce an Evo Task Administration or ETA tool [18] in which the Tasks can be administered and related to deliveries and projects. This tool provides a much easier way for administering Tasks than the Excel "notepad". It is being used and optimized in many projects since beginning 2003. The tool is using a MSAccess database. A conversion of the tool to Internet browser technology is in the works. This will make the tool independent of the availability of MSAccess.

Still, we are still somewhat reluctant to introducing the ETA tool. In some projects, where people are not yet aware of the Evo details, people may start working for the tool in stead of letting the tool work for them. This may distract them from learning the benefits of Evo, to make them more productive, in stead of giving them more work to do.

Important before selecting any tool is to know what we want to accomplish and why and how. Only then we can check whether the tool could save time and bureaucracy rather that costing time and bureaucracy.

10. CONCLUSION

We described issues that are addressed by the Evo methods and the way we organise Evo projects. By using these methods in actual projects we find:

- ***Faster results***
  Evo projects deliver better results in 30% shorter time than otherwise. Note: 30% shorter than what by conventional methods would have been achieved. This may be longer than initially hoped for.
  Although this 30% is not scientifically proven, it is rather plausible by considering that we constantly check whether we are doing the *right things in the right order to the right level of detail for that moment*.

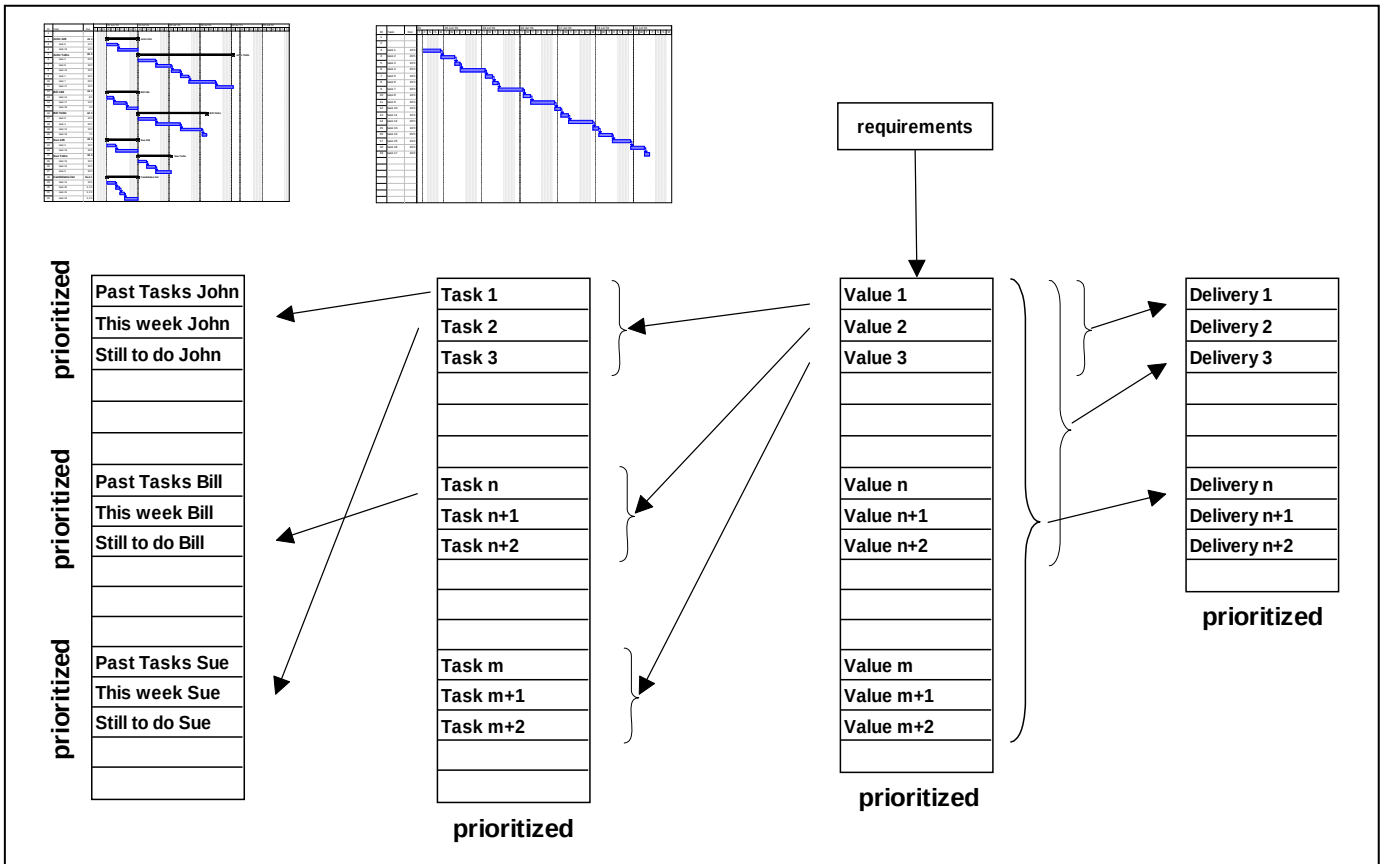


Figure 13: Relations between requirements, stakeholder values, deliveries, and different views on tasks.

This means that any other process is always less efficient. Most processes (even if you don't know which process you follow, you are following an intuitive *ad hoc* process) cause much work to be done incorrectly and then repaired, as well as unnecessary work. Most developers admit that they use more than half of the total project time on debugging. That is repairing things they did wrong the first time. In Evo, most "bugs" are prevented.

- ***Better quality***
  We define quality as (Crosby [4]) "Conformance to Requirements" (how else can we design for quality and measure quality). In Evo we constantly reconsider the validity of the requirements and our assumptions and make sure that we deliver the most important requirements first. Thus the result will be at least as good as what is delivered with the less rigorous approach we encounter in other approaches.
- ***Less stressed developers***
  In conventional projects, where it is normal that tasks are not completed in time, developers constantly feel that they fail. This is very demotivating. In Evo projects, developers succeed regularly and see regularly real results of their work. People enjoy success. It motivates greatly. And because motivation is the motor of productivity, the productivity soars. This is what we see happening *within two weeks* in Evo projects: People get relaxed, happy, smiling again, while producing more.
- ***Happy customers***
  Customers enjoy getting early deliveries and producing regular feedback. They know that they have difficulty in specifying what they really need. By showing them early deliveries and being responsive to their requirements changes, they feel that we know what we are doing. In other developments, they are constantly anxious about the result, which they get only at the end, while experience tells them that the first results are usually not OK and too late. Now they get actual results even much earlier. They start trusting our predictions. And they get a choice of time to market because we deliver complete, functioning results, with growing completeness of functions and qualities, well before the deadline. This has never happened before.
- ***More profits***
  If we use less time to deliver better quality in a predictable way, we save a lot of money, while we can earn more money with the result. Combined, we make a lot more profit.

In short, although Brooks predicted a long time ago that "There is no silver bullet" [2], we found that the methods presented, which are based on ideas practiced even before the "silver bullet" article, do seem to be a "magic bullet" because of the remarkable results obtained.

ACKNOWLEDGEMENT

A lot (but not all) of the experience with the approach described in this booklet has originally been gained at Philips Remote Control Systems, Leuven, Belgium.

In a symbiotic cooperation with the group leader, Bart Vanderbeke, the approach has been introduced in all software projects of his team. Using short discuss-implement-check-act improvement cycles during a period of 8 months, the approach led to a visibly better manageability and an increased comfort-level for the team members, as well as for the product managers.

We would like to thank the team members and product managers for their contribution to the results.

REFERENCES

[1] H.D. Mills: Top-Down Programming in Large Systems. In Debugging Techniques in Large Systems. Ed. R. Ruskin, Englewood Cliffs, NJ: Prentice Hall, 1971.
[2] F.P. Brooks, Jr.: No Silver Bullet: essence and Accidents of Software Engineering. In Computer vol 20, no.4 (April 1987): 10-19.
[3] T. Gilb: Principles of Software Engineering Management. Addison-Wesley Pub Co, 1988, ISBN: 0201192462.
[4] P.B. Crosby: Quality Is Still Free. McGraw-Hill, 1996. 4th edition ISBN 0070145326
[5] T. Gilb: manuscript: Evo: The Evolutionary Project Managers Handbook, 1997, www.gilb.com/Download/EvoBook.pdf
[6] S.J.Prowell, C.J.Trammell, R.C.Linger, J.H.Poore: Cleanroom Software Engineering, Technology and process. Addison-Wesley, 1999, ISBN 0201854805.
[7] T. Gilb: Impact Estimation Tables: Understanding Complex Technology Quantatively, 1998, www.stsc.hill.af.mil/crosstalk/1998/12/gilb.asp
[8] N.R. Malotaux: TaskSheet, 2000, www.malotaux.nl/nrm/English/Forms.htm
[9] F.P. Brooks, Jr.: The mythical man-month. Addison-Wesley, 1975, ISBN 0201006502. Reprint 1995, ISBN 0201835959.
[10] C. Northcote Parkinson: Parkinsons Law. Buccaneer Books, 1996, ISBN 1568490151.
[11] N.R. Malotaux: Powerpoint slides: Evolutionary Delivery. 2001, www.malotaux.nl/nrm/pdf/EvoIntro.pdf
[12] E. Dijkstra: Paper: Programming Considered as a Human Activity, 1965. Reprint in Classics in Software Engineering. Yourdon Press, 1979, ISBN 0917072146.
[13] W. A. Shewhart: Statistical Method from the Viewpoint of Quality Control. Dover Publications, 1986. ISBN 0486652327.
[14] W.E. Deming: Out of the Crisis. MIT, 1986, ISBN 0911379010.
[15] Kent Beck: Extreme Programming Explained, Addison Wesley, 1999, ISBN 0201616416.
[16] www.gilb.com
[17] www.extremeprogramming.org
[18] www.malotaux.nl/nrm/Evo/ETAF.htm

N R Malotaux - Consultancy, Bongerdlaan 53, 3723 VB Bilthoven, The Netherlands
Phone: +31-30-228 88 68, Fax: +31-30-228 88 69
E-mail: niels@malotaux.nl, Web: www.malotaux.nl/nrm/English

Niels Malotaux

Evolutionary Project Management Methods

How to deliver *Quality On Time* in Software Development and Systems Engineering Projects

In this booklet, we show methods and techniques, which enable software and systems developers and management to successfully managing projects within the constraints of cost, schedule, functionality and quality. These methods are taught and coached in actual development projects with remarkable results: typically, projects can be done in 30% shorter time than before.

While software development results were usually delivered late, the delays in other disciplines (like hardware and mechanical development) seemed to be non-existent. Where we have taught Software Development to deliver Quality On Time (the right things at the right time and within budget), the delays in the other disciplines become exposed. The methods and techniques described in this booklet are obviously not limited to just software development. For those projects where delivering Quality On Time is important, it is about time that we are going to apply the techniques at the Systems Development level. Therefore the next target for Evolutionary Development Methods will be Systems Engineering.

Niels Malotaux is an independent Project Coach specializing in optimizing project performance. He has over 30 years experience in designing 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 coached some 80 projects in 20+ organizations in the Netherlands, Belgium, Ireland, India, Japan and the US, which led to a wealth of experience in which approaches work better and which work less well in practice. He is a frequent speaker at conferences, see www.malotaux.nl/nrm/Conf.

Find more at: www.malotaux.nl	Evo pages are at: www.malotaux.nl/nrm/Evo
1 Evolutionary Project Management Methods	www.malotaux.nl/nrm/pdf/MxEvo.pdf
2 How Quality is Assured by Evolutionary Methods	www.malotaux.nl/nrm/pdf/Booklet2.pdf
3 Optimizing the Contribution of Testing to Project Success	www.malotaux.nl/nrm/pdf/EvoTesting.pdf
3a Optimizing Quality Assurance for Better Results (same as 3, but now for non-software projects)	www.malotaux.nl/nrm/pdf/EvoQA.pdf
4 Controlling Project Risk by Design	www.malotaux.nl/nrm/pdf/EvoRisk.pdf
5 TimeLine: Getting and Keeping Control over your Project	www.malotaux.nl/nrm/pdf/TimeLine.pdf
6 Recognizing and Understanding Human Behaviour	www.malotaux.nl/nrm/pdf/HumanBehavior.pdf
7 Evolutionary Planning (this booklet) (similar to TimeLine, but other order and added predictability)	www.malotaux.nl/nrm/pdf/EvoPlanning.pdf
ETA: Evo Task Administration tool	www.malotaux.nl/nrm/Evo/ETAF.htm

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 www.malotaux.nl/nrm/English
Evoweb www.malotaux.nl/nrm/Evo

First published: Dec 2001 - This print: V1.4b 23 Aug 2007