How to Achieve the Most Important Requirement

Niels Malotaux
N R Malotaux - Consultancy
the Netherlands
niels@malotaux.nl

Abstract. The most important requirement for most projects is time - *time for completion*. Still, most projects are late. Isn't it weird that projects apparently judge all other requirements more important than the requirement of time while time is one of the most important requirements? Both Project Management (responsible for the project) and Systems Engineering (responsible for the product) are responsible for the consequences of ignoring this important requirement. This paper describes why it is important to be on time, what measures we can take to make sure we are on time (and which often applied intuitive measures don't work!), and how we can use Evolutionary Planning techniques to make sure that we will be on time, or, if that is simply impossible, to take the consequence. These techniques allow us in the early stages of our project to predict and to optimize what will be ready at a certain time.

Note - This is not a scientific paper but rather based on empirical evidence collected by coaching over 100 projects in the past 8 years.

The Most Important Requirement

The most important requirement for most projects is time - *time for completion*. Projects are supposed to generate a considerable Return on Investment. Therefore the cost of one day delay is not only the cost of running the project one day longer, but also the cost of not being ready one more day (cost of people or equipment waiting, missed revenue, etc), which is usually a lot more than the cost of the project itself. Project delay is costly.

Still, most projects are late. Isn't it weird that projects apparently judge all other requirements more important than the requirement of time while time is one of the most important requirements? Both Project Management (responsible for the project) and Systems Engineering (responsible for the product) are responsible for the consequences of ignoring this important requirement.

Are Systems Engineers Interested in Time?

Many people in projects, including Systems Engineers, think that the delivery time of the project result is not their responsibility, but rather the responsibility of project management. They also seem to think that the system is ready only if "all" requirements have been met. The existence of this thinking is the very reason of producing this paper.

All people working in a project spend time and should spend their time wisely, taking into account the impact of their decisions on the success of the project. Where "other" engineers still may be accused of silo-thinking, the very reason of Systems Engineering is to avoid silo-thinking, taking responsibility for a multi-dimensional variety of issues: whole lifetime (*cradle to cradle*), over *all* disciplines (including e.g. human behavior, see Malotaux 2008), balancing all systems requirements, including performances, and optimizing the design decisions over *all* requirements, *including* delivery time.

Example: the engineers who designed and built the baggage handling system of London Heathrow Airport Terminal 5 claimed that their system was a huge technical success and that the failure to get tens of thousands of bags on board of the proper aircraft was caused by "human error". After all, the terminal was delivered on time and on budget, which admittedly was quite an achievement. However, a passenger is not interested in the technical detail of baggage handling at an airport. The passenger checking in his baggage expects to receive it back in correct condition as quickly as possible after arriving at his destination. That's what performance is about. How this is achieved is irrelevant to the passenger. The "system", as seen by an important group of users (the passengers), was not delivered properly on time and the delays caused a lot of inconvenience and extra costs.

Why are Projects Late?

If we ask people of a project why they are late, they have a lot of excuses, usually external factors being the cause of delays. If we ask them what we could have done about it, they easily have suggestions. We usually know why we are late and we know ways to do something about it. The problem is that we don't do something about it. One of the problems is that customers fatalistically think that this is the way it is and keep paying. If the customers would insist on the delivery date or else wouldn't pay, the problem would have been solved a long time ago.

Some typical causes of delay are:

- waiting (before and during the project)
- indecisiveness
- doing unnecessary things
- doing things less cleverly than we could
- doing things over again
- changing requirements (but sometimes they *do* change)
- unclear requirements
- misunderstandings
- no feedback from stakeholders
- hobbying
- political ploys

The only justifiable cost is the cost of developing the right things at the right time. This looks like perfection and we know that people are not perfect. That is, however, not a license to fatally accept all these delays. A lot of delay is avoidable and therefore unjustifiable.

Why is Time so Important?

We run a project to design and realize a new system, because the new system improves upon previous performance. If it doesn't, there is no reason for the system to be realized. The improvement (e.g. less loss, more profit, faster achieving the same, doing more in shorter time, being happier than before) should have a value way more that the cost of the project.

Initially, every day of the project adds value, but towards the end of the project we are in the area of diminishing returns and every extra day may add less than the return we would gain by the use of the system. This calls for a constant attention to the business case, and a requirements, architecture and design process that optimizes the opportunities and challenges of the business case. This puts the attention to *delivery time* right in the centre of the Systems Engineering activities. In some cases some extra time can significantly increase the performance of the system, however, in other cases spending less time can also increase the revenues from the system: the longer the development takes, the longer the users have to wait for the enhanced performance that the project will provide. Time is money and we don't have the right to waste it, unless it's our own.

If the system we are realizing is a part for a larger system, the system integrator (our customer if we are a sub-contractor) prepares other systems, people and equipment to do the integration into his system at a certain time. He also alerts the potential users of the system that they can start reaping the benefits of the new system at a certain time. If he doesn't get our sub-system on time, he's loosing money, the other systems, people and equipment staying idle, while the potential users of the system have to change their plans. The cost of one day of our customer and deprived benefit to the users is a lot more than we realize.

Even doing nothing is a cost factor. Managers think that there is no cost involved when people are not (yet) working on a project. This is a misconception. Once the idea of the project is born, the timer starts ticking. Every day we start a project later, it will be finished a day later, depriving us from the revenues which by definition are higher that the cost of the project, otherwise we shouldn't even start the project. The only good reason why we delay this project is that we are spending our limited resources on *more profitable* projects.

The Fallacy of "All Requirements"

In many projects people say: "*All* requirements have to be done, and it simply takes as much time as it takes; we cannot stop before *all* is done". What *all* is, is usually not really clear and should be defined by the requirements, which have to be in tune with the business case, which in most projects isn't clear to the project either. These people for some strange reason forget that delivery time is as much a requirement as "all" other requirements.

Systems Engineers are supposed to know how to define real requirements, but they know that defining the right requirements is not easy. For most customers, defining requirements is not a part of their normal work, so for customers this is even more difficult. How can we expect that customers can properly provide us with the right requirements?

Customers specify things they do not really need and forget to specify things they do need. It's the challenge for the Systems (or Requirements) Engineer to find the real relevant requirements, together with the stakeholders. Furthermore, the requirements are what the stakeholders require, however, for a project, the requirements are what the project is *planning to satisfy*. After all, we can make great systems, but if the customer cannot afford the cost, or has to wait a long time, we both loose.

Because there are always conflicting requirements (e.g. more performance can be at odds with acceptable time or cost), the design process is there to balance and come to an optimum compromise between the conflicting requirements. The notion of "all" requirements pretends that "all" requirements can be met concurrently. If this were the case, projects would on average be on time. We know better.

How to Meet the Most Important Requirement

There are many things we can do to save time in order to get the result of our project on time. As soon as we see that it's impossible to be on time, we can tell our customer and discuss what we do with this knowledge. If we tell at the end of the project, our customer really has a problem. If we tell it as soon as we could have known, which is much, much earlier in the project, the customer may not like it, but he has more time to cope with the consequences.

In the remainder of this paper we will first discuss the options we (seem to) have to get our project result earlier. Then we will discuss the techniques that are available to really actively make sure that we always will be on time.

Which Options Do We (Seem to) Have to Save Time?

What can we do if what we think[1] we have to do doesn't fit the available time, or if we want to do things faster? There are several ways we see people use to try to finish a project earlier, most of which are intuitively right, but don't work. This contradiction causes people to think that we have to accept late projects as a fact of life. After all, they did their best, even took measures (correct measures according to their intuition), and it didn't work out. There are, of course, also measures that do work.

Deceptive measures. Let's first do away with the deceptive measures. Deceptive measures are measures we often see applied, but which don't work. It's surprising that people don't learn and keep using them.

Hoping for the best (fatalistic type)
Most projects take more time than expected. Your past project took longer than expected. What makes you think that this time it will be different? If you don't change something in the way you run the project, the outcome won't be different, let alone better. Just hoping that your project will be on time this time won't help. We call this ostriching: putting your head into the sand waiting until Murphy[2] strikes again.

Going for it (macho type)
We know that the available time is insufficient, but it *has* to be done: "Let's go for it!" If nothing goes wrong (as if that ever is the case) and if we work a bit harder (as if we don't already work hard) … Well, forget it.

Working Overtime (fooling yourself)
Working overtime is fooling yourself: 40 hours of work per week is already quite hard. If you put in more hours, you'll get more tired, make more mistakes, having to spend extra time to find and "fix" the mistakes, half of which you won't. You *think* you are working hard, but you aren't working *smart*. It won't work. This is also ostriching. As a rule, never work overtime, so that you have the energy to do it once or twice a year, when it's really necessary.

Adding time: moving the deadline
Moving the deadline further away is also not a good idea: the further the deadline, the more danger of relaxing the pace of the project. We call this Parkinson's Law[3] or the Student Syndrome[4]. At the new deadline we probably hardly have done more, getting the project result even later. Not a good idea, unless we really are in the nine mother's area (see next), where nobody, even with all the optimization techniques available, could do it. Even then, just because of the Student Syndrome, it's better to optimize what we can do in the available time before the deadline. The earlier the deadline, the longer our future afterwards, in which we can decide what the next best thing there is to do. So the only way a deadline may move is *towards* us. We better optimize the time spent right from the beginning, because we'll probably need that time anyway at the end. Optimizing only at the end won't bring back the time we lost at the beginning.

---

1 We keep saying "what we *think* we have to do", because however good the requirements are, they will change, because *we* learn, *they* learn and the circumstances change. The longer the project, the more the requirements have a chance to change. And they *will* change! But what we do not yet know, we cannot yet plan for.

2 *Whatever can go wrong, will go wrong* is the popular version of Murphy's Law. The real version is: *What can go wrong, will go wrong, so we have to predict all possible ways it can go wrong, and make sure that these cannot happen*. Spark (2006).

3 Parkinson's Law: "*Work expands so as to fill the time available for its completion*" (People use the time given). Parkinson (1955) observed: "Granted that work (and especially paperwork) is elastic in its demands on time, it is manifest that there need be little or no relationship between the work to be done and the size of the staff to which it may be assigned."

4 Starting as late as possible, only when the pressure of the FatalDate is really felt. Term attributed to E. Goldratt (1997).

A riskful option: adding people …
A typical move is to add people to a project, in order to get things done in less time. Intuitively, we feel that we can trade time with people and finish a 12 person-month project in 6 months with 2 people or in 3 months with 4 people, as shown in Figure 1. In his essay The Mythical Man-Month, Brooks (Brooks 1975) shows that this is a fallacy, defining Brooks' Law: *Adding people to a late project makes it later*.

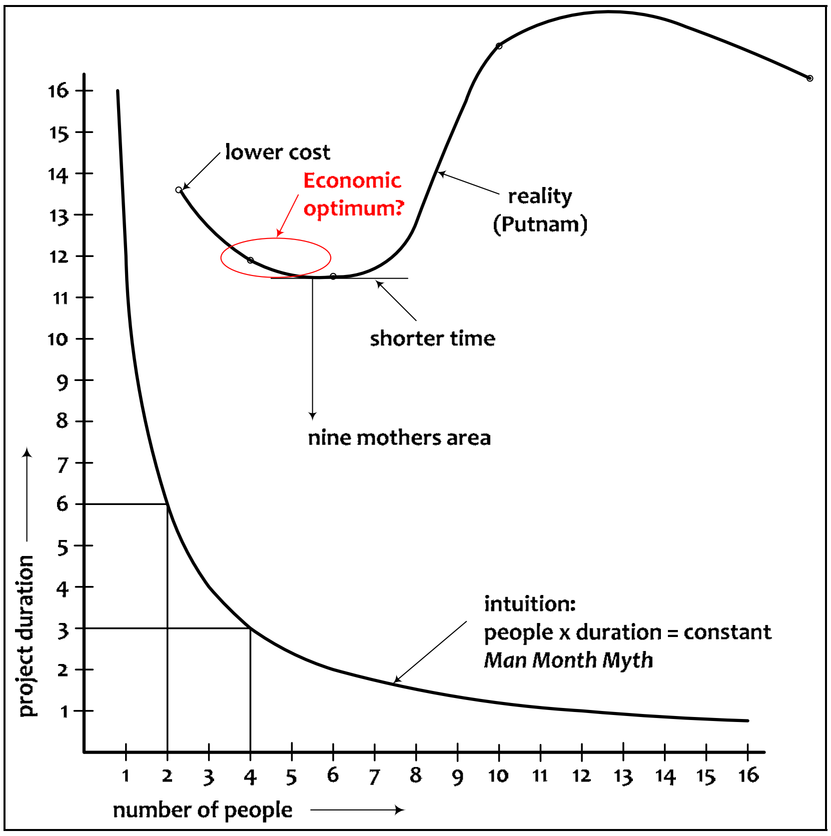


Figure 1: The Myth of the Man-Month: reality is completely different

Putnam (Putnam 1996) confirms this with measurements on some 500 (software) projects. He found that if the project is done by 2 or 3 people, the project-cost is minimized, while 5 to 7 people achieve the shortest project duration at premium cost, because the project is only 20% shorter, with double the amount of people. Adding even more people makes the project take *longer* at *excessive* cost. Apparently, the project duration cannot arbitrarily be shortened, because there is a critical path of things that cannot be parallelized. We call the time in which nobody can finish the project the *nine mothers area*, which is the area where nine mothers produce a baby in one month. When I first heard about Brooks' Law, I assumed that he meant that we shouldn't add people at the end of a project, when time is running out. After all, many projects seem to find out that they are late only by the end of the project. The effect is, however, much trickier: if in the *first several weeks* of a project we find that the speed is slower than expected, and thus have to assume that the project will be late, even then adding people can make the project later. The reason is a combination of effects. More people means more lines of communication and more people to manage, while the project manager and the architect or the Systems Engineer can oversee only a limited number of people before becoming a bottleneck themselves. So, adding people is not automatically a solution that works. It can be very risky.

How can those mega-projects, where 100's of people work together, be successful? Well, in many cases they are not. They deliver less and later than the customer expects and many projects simply fail. The only way to try to circumvent Brooks' Law is to work with many small teams, who can work in parallel, and who synchronize their results only from time to time. If you think Brook's Law won't bite you, you better beware: it will!

In a recent project that went too slow, the number of people was increased from 5 people to 20 people. The measured productivity increased by 50%. It took project management quite some time to decide to decrease (against their intuition!) the number of people from 20 to 10. Once they did, the net productivity of the 10 people was the same as with 20 people.

The Measures That Always Work - Saving Time

Fortunately, there are ways to save time, *without negatively affecting the Result of the project (on the contrary!)*. These techniques are collected and routinely used in the Evolutionary Project Management (Evo) approach to achieve the best solution in the shortest possible time.

The Evo approach uses and constantly evolutionarily optimizes the elements of saving time: Plan-Do-Check-Act cycles (or Deming cycles - Deming 1986), Zero-Defects attitude (Crosby 1996), Business Case techniques, specific Requirements Management techniques, Design techniques, Early Reviews, and Evolutionary Planning techniques like TaskCycles, DeliveryCycles and TimeLine. Background of the Evo approach can be found in Gilb (1988, 2005) and Malotaux (2004, 2006, 2007). Projects starting to use the Evo approach start saving 30% time within a few weeks, while delivering better results.

The elements of saving time are:

Improving the efficiency in *what* (*why*, for *whom*) we do: doing only what is needed, not doing things that later prove to be not needed, preventing mistakes and preventing working on superfluous things. Because people tend to do more than necessary, especially if the goals aren't clear, there is ample opportunity for *not* doing what is *not* needed. We use the *Business Case* and *continuous Requirements Management* to control this process. We use the *TaskCycle*, to weekly decide what we are going to do and what we are *not going to do, before* we do it. This saves time. Afterwards we only can identify what we unnecessarily did, but the time is already spent and cannot be regained.

Improving the efficiency in *how* we do it: doing things differently.

This works in several dimensions:

The product
Choosing the proper and most efficient solution. The solution chosen determines both the performance and cost of the product, as well as the time and cost of the project. Because performance and project time are usually in competition, the solution should be an optimum compromise and not just the first solution that comes to mind. We use *Architecture* and *Design* processes to optimize the result. We use *DeliveryCycles* to check the requirements and assumptions.

The project
We can probably do the same in less time if we don't immediately do it the way we always did, but first think of an alternative and more efficient way. We do not only design the product, we also continuously *redesign the project*. We use *Evolutionary Planning* to control this process.

Continuous improvement and prevention processes
Actively and constantly learning how to do things better and how to overcome bad tendencies. We use rapid and frequent Plan-Do-Check-Act (PDCA- or Deming) cycles to actively improve the product, the project *and* the processes. We use *Early Reviews* to recognize and tackle tendencies before they pollute our work products any further and we use a *Zero-Defect attitude* because that is the only way ever to approach Zero Defects.

Improving the efficiency of *when* we do it: doing things at the right time, in the right order**.** A lot of time is wasted by synchronization problems like waiting for each other, or redoing things that were done in the wrong order. *Actively Synchronizing* and *designing* the order of what we do saves time. We use *Evolutionary Planning* with constant, active prioritization to control this process, with *TaskCycles* and *DeliveryCycles* to make sure we do the right things in the right order, and *TimeLine* to get and keep the whole project under control. Elements of these are *Just Enough Estimation*, *Dynamic Prioritizing* and *Calibration* techniques.

All of these elements are huge time savers. Of course we don't have to wait for a project getting into trouble. We can also apply these time savers if what we think we have to do easily fits in the available time, to produce results even faster. We may even need the time saved to cope with an unexpected drawback, in order still to be on time and not needing any excuse.

TimeBoxing provides the incentive to constantly apply these ways to save time, in order to stay within the TimeBox. For TimeBoxing to work properly, it is important to change from optimistic or pessimistic, to realistic estimation. If the TimeBox is too short, we cause stress with adverse effects. If the TimeBox is too long, we're wasting time. In the experience of the author, people in projects can easily change into realistic estimators in a few weeks time, *if and only if* we are *serious about time*. TimeBoxing is much more efficient than FeatureBoxing (= waiting until we're ready), because with FeatureBoxing we lack a deadline, causing Parkinson's Law and the Student Syndrome to kick in badly.

Note that this concept of saving time is similar to "eliminating waste" in Lean thinking, and already indicated by Henry Ford in his book "My Life and Work", back in 1922 (Ford 1922): "*We eliminated a great number of wastes*".

Because the time saving actions don't come easy (otherwise this would be practiced already everywhere), it's advisable for Systems Engineering to work together and synchronize with Project Management, to constantly seek for ways to improve on this. We suggest studying the Evolutionary approach and using it to the advantage of the project success.

Evolutionary Planning

The Evolutionary Planning process uses three main elements:

- **The weekly TaskCycle** to organize the work, to make sure we are at any time working only on the most important things and don't work on less important things. We quickly learn to promise what we can do and then to live up to our promises. This removes a lot of quick-sand from under the project.
- **The bi-weekly DeliveryCycle**, to check the requirements and challenge our assumptions.
- **TimeLine** to get and keep control over longer periods of time and to provide reliable information to Portfolio/Program/Resource management.

We'll show now how these three elements fit together to get and keep the project under control.

TimeLine. In many projects all the work we think we have to do is cut into pieces, the pieces are estimated, and the estimations are added up to arrive at an estimate of the effort to do the work. Then this is divided over the available people (remember Brook's Law!), to arrive at an estimate of the duration of the work, which, after adding some contingency, is presented as the duration of the project (Figure 2). A problem is that in many cases the required delivery date is earlier. Discussions about this tension between estimated and expected delivery consume time, while the required delivery date doesn't change, leaving even less time for the project.

Because the delivery date is a requirement just as all the other requirements, it has to be treated as seriously as all the other requirements. With TimeLine, we treat delivery dates seriously and we meet these dates, or we very quickly explain why the delivery date cannot be met.

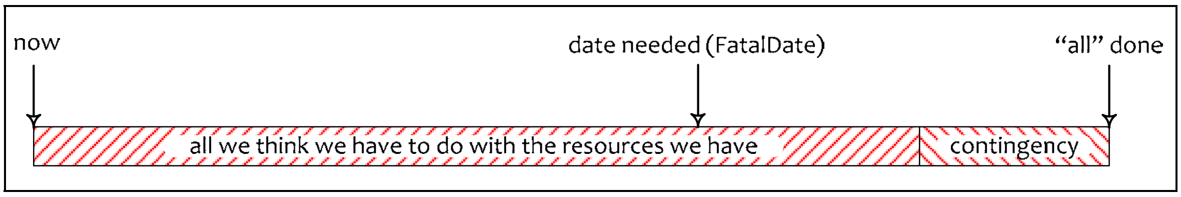


Figure 2: Standard approach: it takes what it takes, but that may be too late

We don't wait till the FatalDate to tell that we didn't make it, because if it's really impossible, we know it much earlier. If it *is* possible, we deliver, using all the time-saving techniques to optimize what we can deliver when.

TimeLine can be used on any scale: on a program, a project, a sub-project, on deliveries, and even on tasks. The technique is always the same. We still estimate what we think we have to do. Then we discuss the TimeLine with our customer and explain (Figure 3):

- What, at the FatalDate, surely will be done
- What surely will not be done
- What may be done (after all, estimation is not an exact science)

If what surely will be done is not sufficient for success, we better stop now to avoid wasting time and money. Note that we put what we plan in strict order of priority, so that at the FatalDate we'll only have done the most important things. Customers don't mind about the bells and whistles if Time to Market is important. Because priorities may change very dynamically, we have to constantly reconsider the order of what we do when.

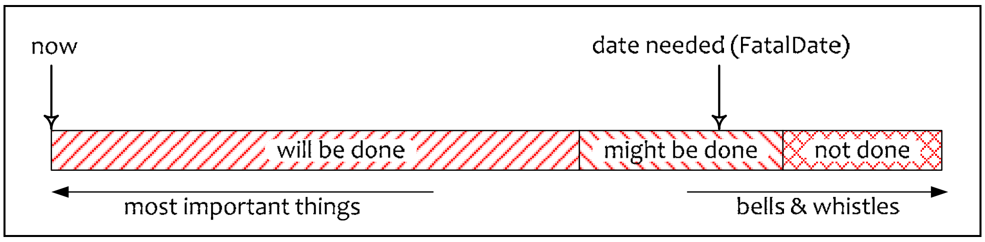


Figure 3: Basic TimeLine: what will surely be done, what will not, and what may be done

Setting a Horizon. If the total project takes more than 10 weeks, we define a Horizon at about 10 weeks on the TimeLine, because we cannot really oversee longer periods of time. A period of 10 weeks proves to be a good compromise between what we can oversee, while still being long enough to allow for optimizing the *order* in which we deliver results. We don't *forget* what's beyond the horizon, but for now, we concentrate on the coming 10 weeks.

DeliveryCycles. Within these 10 weeks, we plan DeliveryCycles (Figure 4) of maximum 2 weeks, asking: "*What* are we going to deliver to *whom* and *why*?" Deliveries are for getting feedback from Stakeholders. We are humble enough to admit that our (and their) perception of the requirements is not perfect and that many of our assumptions may be incorrect. Therefore we need communication and feedback. We deliver to *eagerly waiting* Stakeholders, otherwise we don't get feedback. If the appropriate Stakeholders aren't eagerly waiting, either they're not interested and we'd better work for other Stakeholders, or they *have to* be made eagerly waiting by delivering *Juicy Bits*. How can juicy bits have a high priority? If we don't get appropriate feedback, we will probably be working with incorrect assumptions, doing things wrong, which will cause delay later. So it may be high priority to deliver juicy bits to Stakeholders to make them eagerly waiting in order to get the feedback that we awfully need.

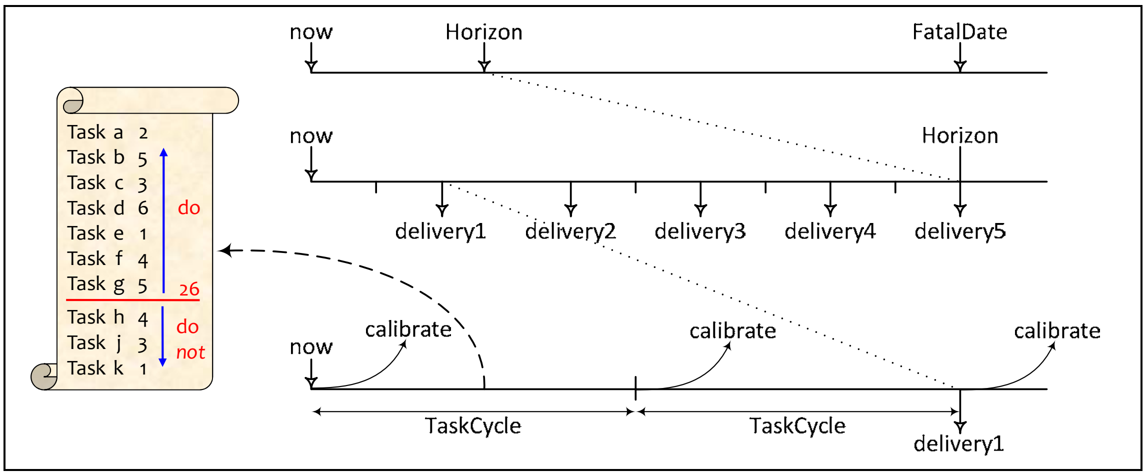


Figure 4: TimeLine summary: setting a FatalDate, a Horizon, Deliveries, TaskCycles, and then calibrating back

TaskCycles. Once we have divided the work over Deliveries, which are Horizons as well, we now concentrate on the first few Deliveries and define the actual work that has to be done to produce these Deliveries. We organize this work in TaskCycles of one week. In a TaskCycle we define Tasks, estimated in net effort-hours (see Malotaux 2004, section 6.1, for a more detailed explanation). We plan the work in plannable effort time, which defaults to 2/3 of the available time (26 hrs in case of a 40 hr week), *confining* all unplannable project activities like email, phone-calls, planning, small interrupts, etc, to the remainder of the time. We put this work in optimum order, divide it over the people in the project, have these people estimate the time they would need to do the work, see that they don't get overloaded and that they synchronize their work to optimize the duration.

Calibration

Having estimated the work that has to be done in the first week, we have captured the first metrics for *calibrating* our estimations on the TimeLine. If the Tasks for the first week would deliver about half of what we need to do in that week, we now can extrapolate that our project is going to take twice as long, *if* we don't do something about it. Initially this seems weak evidence, but it's already an indication that our estimations may be too optimistic. Putting our head in the sand for this evidence is dangerous: I've heard all the excuses about "one-time causes". Later there were always other "one-time causes".

One week later, when we have the *actual* results of the first week, we have slightly better numbers to extrapolate and scale how long our project really may take. Week after week we will gather more information with which we calibrate and adjust our notion of what will be done at the FatalDate or what will be done at any earlier date. This way, the TimeLine process provides us with very early warnings about the risks of being late. The earlier we get these warnings, the more time we have to do something about it.

Let's take an example of taking the consequence of the TimeLine (Figure 5): At the start, we estimate that the work we think we have to do in the coming 10 weeks is about 50 person Weeks of Work ('WoW', line a). We start with 5 people. After 4 weeks, we find that 10 WoW have been completed (line b), instead of the expected 20 WoW. If we don't change our ways of working, the project will take twice as long (line d) or produce only half (line c). If the deadline is really hard, a typical reaction of management is to throw more people at the project (line e). However, based on our progressing understanding of the work, we found that we forgot to plan some work that also "has" to be done: we now think we still have to do 50 WoW in the remaining 6 weeks (line f).

Management decides to add even more people to the project, because they don't want the project to take longer. This solution probably won't produce the desired outcome, and even may work out counterproductive, because of Brook's Law.

We can counter this dilemma by *actively saving time*, doing only what is really necessary (line g), or a combination of *not* doing what is *not* necessary (line g2) and doing things more productively (line h), as explained on page 6 and 7. Actively *designing* what exactly to do and in which order, saves a lot of time.

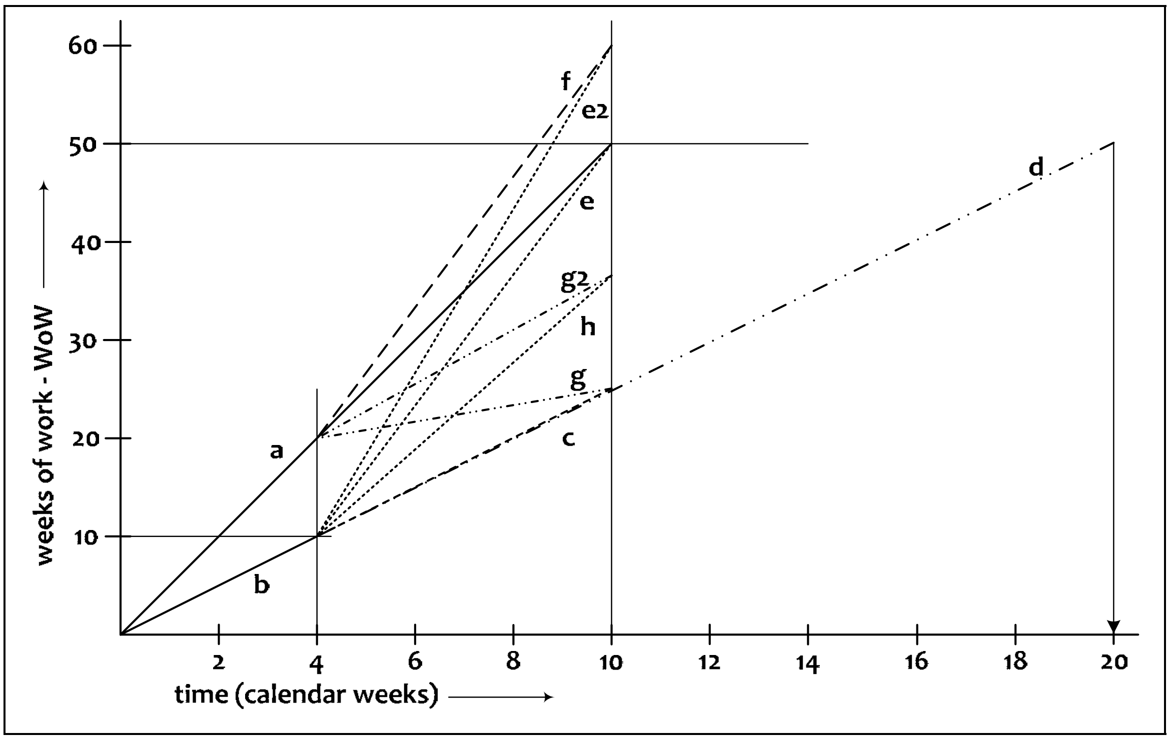


Figure 5: Earned Value (up to week 4) and Value Still to Earn (from week 5)

Predicting the future. Using calibration, we can quite well predict what will be done when (figure 6). The estimates don't have to be exact, as long as the relative values are consistent: if Activity1 is estimated to take 2 (units of estimation) and Activity2 to take 1, then we assume that Activity1 will take twice as long as Activity2. We see that people are reluctant to accept that rather imprecise estimates yield rather good overall predictions. In practice, the positive and negative inaccuracies average out, providing better credibility of the total accuracy.

Once we have done several activities, we know how long these activities took and now we can calibrate the remainder of the estimations to reality. We average the calibration factor over a several recent activities until now:

$$\text{Calibration Factor} = \frac{\sum_{now-1}^{now-n} Ar}{\sum_{now-1}^{now-n} Ae} \quad (Ar \text{ is real time, } Ae \text{ is estimated time of an Activity})$$

Now we can use this calibration factor to predict how much time we need for future activities:

$$\text{Value Still to Earn}_{(\text{by } then)} = \text{Calibration Factor} * \sum_{now}^{then} Ae$$

This way we can predict when we will have done what, or when "all" is done.

This list of activities still to do (*Value Still to Earn*) will constantly be updated:

- Activities will be added when we recognize that we forgot some things we have to do
- Activities will be updated when we realize that we have to better define them
- Activities will be deleted once we see that they don't add value

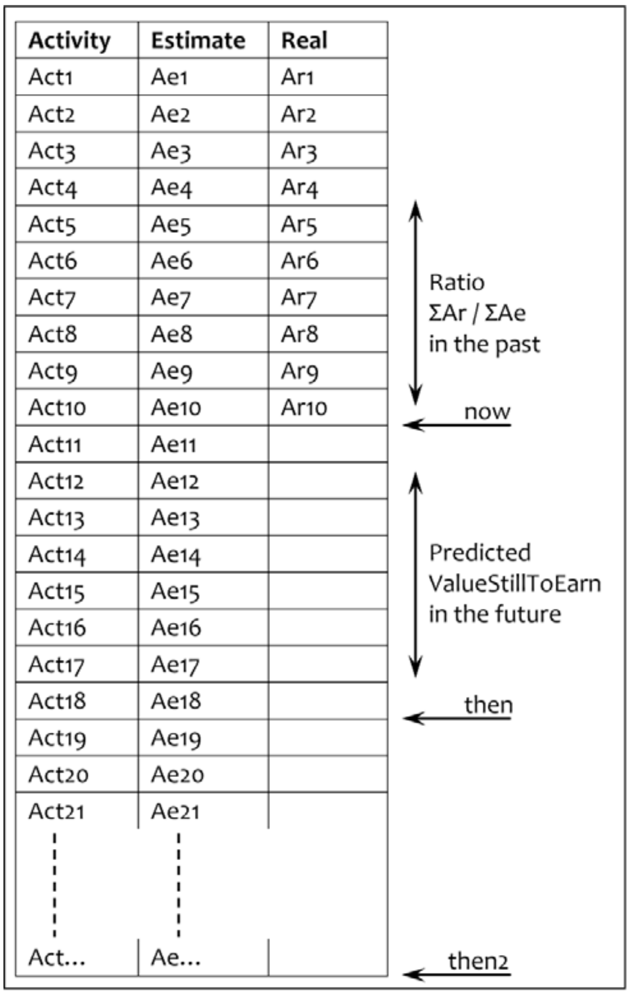
The order of activities will be changed, once we find out that the priorities have changed

Estimates will be updated to reflect better insight, although the estimates should be made based on the same assumptions as the original estimates, to keep the calibration working

Note that we shouldn't use these numbers mechanistically. We still have to judge the credibility of what the mathematics tell us and adjust our understanding accordingly.

In conventional projects this manual interpretation may still lead to over-optimistic predictions, especially if what the numbers tell us is "undesirable". In Evo projects, however, we want to succeed in the available time or earlier, so we are realistic and rather see any warning we can use to constantly improve, or to discuss the consequences with the customer as soon as possible.

In practice I've seen calibration factors of 2 at the start of the project and then growing and stabilizing at 4 when the project is running at full strength. In other projects we see other factors. Note that the calibration factors of different projects are not good or bad and cannot be compared: they are simply the ratio of how much time this project needs to accomplish its activities, and the estimations as produced by the project's estimation standard. Both factors are different for different projects. They merely calibrate the assumptions used at the original estimation, where we may not have taken into account V&V, SE, project management, education, and many other things that have to be done in the project as well. Once the calibration factor has stabilized, we can use the slope of the factor to warn for deterioration and to see the effect of process improvements.



Activity	Estimate	Real
Act1	Ae1	Ar1
Act2	Ae2	Ar2
Act3	Ae3	Ar3
Act4	Ae4	Ar4
Act5	Ae5	Ar5
Act6	Ae6	Ar6
Act7	Ae7	Ar7
Act8	Ae8	Ar8
Act9	Ae9	Ar9
Act10	Ae10	Ar10
Act11	Ae11	
Act12	Ae12	
Act13	Ae13	
Act14	Ae14	
Act15	Ae15	
Act16	Ae16	
Act17	Ae17	
Act18	Ae18	
Act19	Ae19	
Act20	Ae20	
Act21	Ae21	
Act...	Ae...	

Figure 6: Using the list of activities to predict what will be done when

Time as a Requirement for the System After the Project

There are much more time requirements than only the duration of the project. *During* the project, we can still influence and optimize the time spent on what we think we have to do to realize the system. *After* the project, however, the system is left to its own devices and must perform autonomously. We have no influence on the timings of the system any more and any performance improvement delivered by the new system has to be there *by design*. This is typically a responsibility and required skill of Systems Engineering. Very important for the success of the project, however, this is not further elaborated in this paper.

Conclusion

Evolutionary Planning doesn't *solve* our problems. It rather is a set of techniques to plan and early expose the real status of our project. Instead of *accepting* an undesired outcome, we have ample opportunity of *doing* something about it. People do a lot of unnecessary things in projects, so it's important to identify those things before spending time on them. If we later find out that we did unnecessary things, the time is already spent, and never can be regained. By revisiting the TimeLine every week, we stay on top of how the project is developing and we can easily report to management the real status of the project and also show the consequences of management decisions affecting the project.

Doesn't all this planning take a lot of time? The first few times it does, because we have to learn how to use the techniques. After a few times, however, we dash it off and we can start optimizing the results of the project, producing more than ever before. Evolutionary Planning allows us to take our head out of the sand, stay in control of the project and deliver Results successfully, on time. Still, many Project Managers hesitate to start using these techniques. However, after having done it once, the usual reaction is: "Wow! I got much better oversight over the project than I ever expected", and the hesitation is over. Another reaction: "We never did this before. Now we're finally in control!"

These techniques are not mere theory. They're highly pragmatic, and successfully used in many projects coached by the author. The most commonly encountered bottleneck is that no one in the project has an oversight of what exactly the project is supposed to accomplish. May be this is why Project Managers hesitate to start using these techniques. If you don't know well what to do, planning isn't easy. Redefining what the project is to accomplish and henceforth focusing on this goal is the first immediate timesaver, with many more savings to follow.

I hear many Systems Engineers say that they know all these things and that they are doing these things already. Be honest. We do know most of the techniques mentioned in this paper, but do we really use and continuously improve on them? If we really would, we wouldn't need excuses for late deliveries any more.

References

Crosby 1996. *Quality is still free*. McGraw-Hill, ISBN 0070145326

Deming 1986, *Out of the Crisis*. MIT, ISBN 0911379010

Ford 1922, *My Life and Work*, www.gutenberg.org/dirs/etext05/hnfrd10.txt

Gilb 1988. Principles of Software Engineering Management. Addison Wesley, ISBN 0201192462

Gilb 2005. *Competitive Engineering*. Elsevier, ISBN 0750665076

Goldratt 1997. *Critical Chain*. Gower, ISBN 0566080389

Malotaux 2004. *How Quality is Assured by Evolutionary Methods*. www.malotaux.nl/nrm/pdf/Booklet2.pdf

Malotaux 2006. *Controlling Project Risk by Design*. www.malotaux.nl/nrm/pdf/EvoRisk.pdf

Malotaux 2007. *TimeLine: Getting and Keeping Control over your Project*. www.malotaux.nl/nrm/pdf/TimeLine.pdf

Malotaux 2008. Recognizing and Understanding Human Behaviour to Improve Systems Engineering Results. APCOSE2008. www.malotaux.nl/nrm/pdf/HumanBehavior.pdf

Parkinson 1955. *Parkinson's Law*. www.adstockweb.com/business-lore/Parkinson's_Law.htm

Putnam 1996. Team Size Can Be the Key to a Successful Project. www.qsm.com/process_01.html

Spark 2006. *A History of Murphy's Law*, Periscope Film, ISBN 0978638891.

Bio

Niels Malotaux is an independent Project Coach specializing in optimizing project performance. He has over 35 years experience in designing 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 100 projects in 25+ organizations in the Netherlands, Belgium, China, Germany, India, Ireland, Israel, Japan, Romania, South Africa and the US, which led to a wealth of experience in which approaches work better and which work less in the practice of real projects.