Controlling Project Risk by Design

Niels Malotaux
N R Malotaux - Consultancy
http://www.malotaux.nl/nrm/English - niels@malotaux.nl
Copyright © 2007 by Niels Malotaux. Published and used with permission.

Abstract. A lot of risks that plague projects have a high probability to occur: e.g. the risk that we don't deliver the right things or deliver late, and the underlying causes of these problems. This calls for proactively anticipating potential problems and organizing our projects in such a way that the probability of the impact is minimized. The Evolutionary Project Management (Evo) approach is just doing that, constantly being aware of what could go wrong and preventing it from going wrong, *by design*. In stead of assuming a theoretical model of how humans "should" behave, Evo studies actual human behavior and strives to make optimum use of how humans actually behave. After all, systems are conceptualized, designed, realized, maintained, used, and to be tolerated by people, so if Systems Engineering and Design ignores, forgets to include, or incorrectly assumes how people behave in all of these activities, the results can be devastating. Opposing what's in our genes is a lost battle.

In this paper we will first investigate prevailing risk management. Then we show how Evo practices are designed to successfully mitigate typical risks in projects. The techniques discussed are not merely theoretical ideas, but have been tested, honed and proved by the author in the practice of more than 60 projects in the past 6 years, in various environments and cultures.

Introduction

If we do nothing, the risk that we won't accomplish a certain thing is 100%. In order to accomplish what we want to accomplish, we organize a project, and at the end of the project the risks are to be reduced to an acceptable level. The level will never be zero, as, for example, a meteorite could strike our result just before delivery of the project result.

A lot of risks that plague projects have a high probability to occur: e.g. the risk that we don't deliver the right things or deliver late, and the underlying causes of these problems. This calls for proactively anticipating any potential problems and organizing our project in such a way that the probability of the impact is minimized. The Evolutionary Project Management (Evo) approach is just doing that, constantly being aware of what could go wrong and preventing it from going wrong, *by design*. In stead of assuming a theoretical model of how humans "should" behave, Evo studies actual human behavior and strives to make optimum use of how humans actually behave. After all, systems are conceptualized, designed, realized, maintained, used, and to be tolerated by people, so if Systems Engineering and Design ignores, forgets to include,

or incorrectly assumes how people behave in all of these activities, the results can be devastating. Opposing what's in our genes is a lost battle.

In this paper we will first investigate prevailing risk management. Then we show how Evo practices are designed to successfully mitigate typical risks in projects. The techniques discussed are not merely theoretical ideas, but have been tested, honed and proved by the author in the practice of more than 60 projects in various environments and cultures. If we have mitigated most of the risks that usually plague projects *by design*, then we have much more time left to handle the unexpected risks we still have to deal with. Evo provides the technique and the attitude to deal with those risks very effectively as well.

The goal of a project

In order to know whether we succeed in projects, we'll first have to define the main Goal of our efforts in projects:

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

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

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

Furthermore, let's define a Defect as:

The cause of a problem experienced by stakeholders of the system, ultimately by the customer.

If there are no defects, we'll have achieved our goal. If there are defects, we failed. Example: the Public Address system used for making announcements to the public in airports or train stations is in many cases hardly audible. Possible defects: bad equipment, bad acoustics, bad speaker. In airplanes the announcements are regularly unclear because the speaker speaks too quickly, not well articulated or in a nice but unintelligible dialect. Defect: insufficient education. Root cause: insufficient management attention. Ultimate cause: insufficient management education.

Being late or over-budget is a defect, as long as it is experienced as a problem. If there is a potential defect, but none of the stakeholders ever experiences a problem because they never use a certain part of the system, then we don't call it a defect. We may ask ourselves why we built that part of the system anyway and may call building that part a defect, because we spent time (= money) on building parts of a system that are not providing return on investment.

Within this definition-framework, Risk is:

An event that may cause a defect.

Prevailing Risk Management

Conventional Risk Management principles are quite straightforward. A common definition of Risk is:

An uncertain event that, if it occurs, has a negative effect on project success.

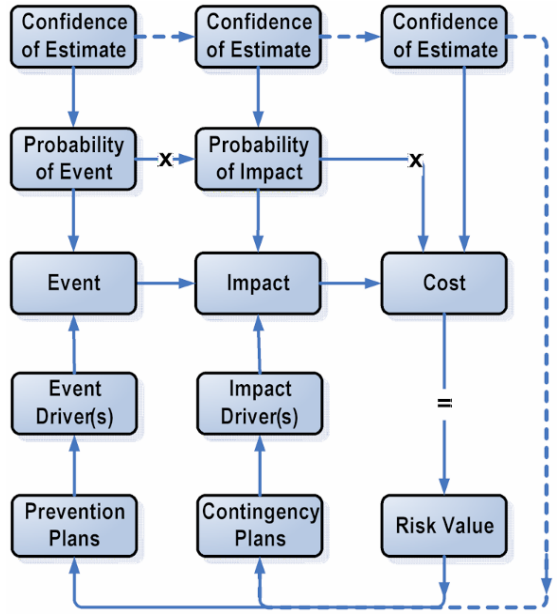


Figure 1: Risk Model

The measure of uncertainty is the probability that the event may occur. If the probability is 0%, it's not a risk. If the probability is 100%, it isn't a risk, but an issue or a problem. The aim is to proactively deal with risks before they become problems. If we deal with 80% of the risks before they become a problem, we have a lot more time to deal with the remaining 20%.

Some people include positive risks in the definition. In practice we see that this causes a lot of confusion, so we prefer to use *opportunity* instead of *positive risk*, reserving *risk* for negative effects.

If a risk event occurs, there is a probability that it impacts our success: if there is an earthquake in Japan, will it impact our project in Paris? If it does impact our success, there is Cost involved (Figure 1).

The product of the Probability that the risk event occurs (P_e), the Probability of the Impact hitting us (P_i) and the Cost if we are impacted by the event (C), is called the Risk Value (V_R): $V_R = P_e * P_i * C$

Because the Probabilities and the Cost are estimates, and often even rough guesstimates, it is better to include some awareness of the (in)confidence of the estimates, to allow for worst-case judgments.

Based on the Risk Value, we may decide to plan for Risk Prevention, preventing the risk event to occur, and/or for Contingency, to minimize the impact if the event occurs. A risk of using this Risk Value product (Figure 2a, ref (Incose 2006), and 2b), is that a very harsh consequence, with a very small likelihood to occur, may be ignored as it may be perceived as a low risk value, while if it occurs, it had better been treated as important. Example: if software for an emergency procedure in an airplane or space shuttle is not tested because other risks seemed more important, it may fail in the (*perceived* unlikely) case that the emergency does occur.

An example of pushing the mathematical treatment of risk too(?) far is the table shown in Figure 3 (Rafele 2005). The text is not well readable, but that's not relevant. Work packages are on the left and risk sources are shown at the top. In the matrix, the Risk Values are computed

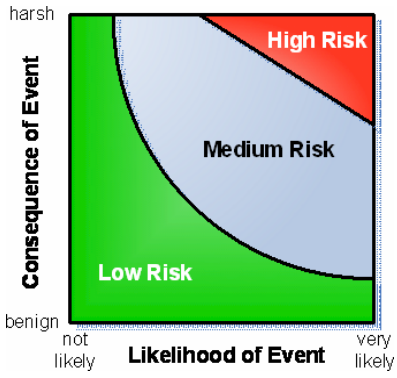


Figure 2a: Problem: very harsh consequence is treated as low risk

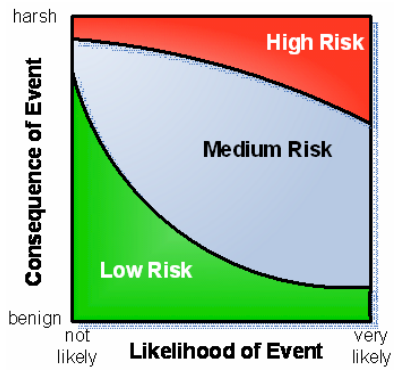


Figure 2b: Is this better?

per work package and risk source. At the right and bottom, the Risk Values in rows and columns are summed, leading to an order of priority of the risk of a certain work package and a certain risk source. It occurs to me that this may be nice theory, but that translating the risks in just numbers, where apples and oranges have been added, obscures what the risks really are all about. We may only use this technique to feed the decision makers with additional insight presenting them the full table rather than just the computed Risk Values. Then they can base their risk mitigation strategy on more than just bare numbers.

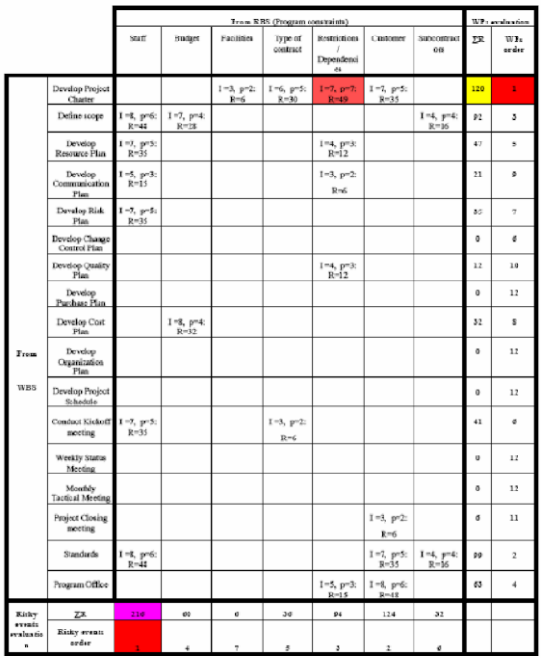


Figure 3:
Mathematical risk management can be risky

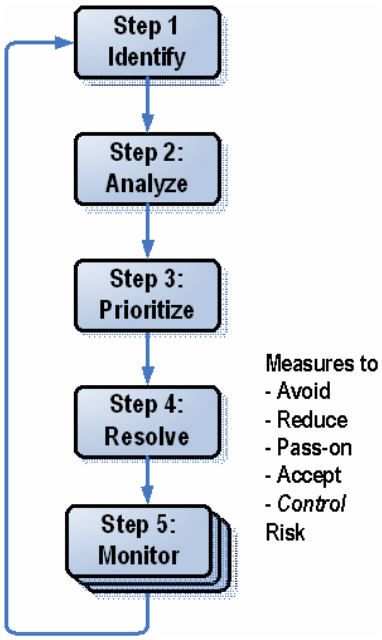


Figure 4:
Iterative Risk Management Process

Risk Management (Figure 4) is done in cycles, Identifying, Analyzing, Prioritizing, Resolving and Monitoring risks. Risks are either avoided, reduced, passed on to others, or accepted. I would like to add here: or *controlled by design*. Passing on to others may be passing the risk to sub-contractors, who may, however, not be capable to run the risk. If they fail, we still fail, so that's a good solution only if the sub-contractor is better equipped to handle the risk than us.

Summarizing, prevailing Risk Management is quite straightforward and important to assist people taking the right preventive measures to proactively deal with risks in order of importance. However, calling every event that can jeopardize our project result a risk, obscures opportunities to control the mitigation of the effects more effectively. It is more useful to call most of the risks by their proper name. That's what we will do in the following chapters.

Risks in Projects

Basic risks in projects are:

- The result of the project is not right (according to the goal)
- It is too late
- It costs more than necessary

Although every project is unique, a lot of what we do in projects is always the same:

- Every project is done by people. People react in certain ways, which, once recognized, are quite predictable, although this "real" human behavior is ignored in many project management approaches, where only some theoretical human behavior is assumed. Often, theory doesn't really work as expected in practice.
- Many activities are the same in every project and can be organized by repeatable processes.
- Only a very small part of the project is really unique, causing specific risks.

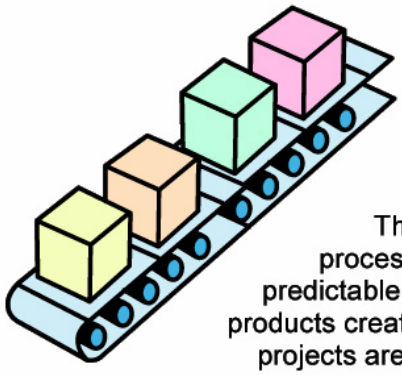


Figure 5: The project process is quite predictable, only the products created in the projects are different

This means that we'd better call the predictable risks (known risks) by their proper name and control them by repeatable processes and only treat the few new-product related risks by a special risk management process (Figure 5). In the following part we will show how most project risks can be controlled *by design*, by proper process rather than calling for separate risk management.

Evolutionary Project Management Methods (Evo)

Evolutionary Project Management is a set of methods and processes, including a certain *attitude*, that allow people to routinely complete projects successfully on time, or earlier. Researching the root-causes of project problems, the author is constantly designing and optimizing methods to overcome these problems, as well as optimizing the process of introducing these methods into projects. Because Evolutionary is a long word, we use the abbreviation Evo, as a label for the current set of methods. Being routinely successful implies that we succeed in systematically controlling the risks threatening our projects.

Elements of these methods are solving the discipline problem, exploiting our intuition mechanism, continuously balancing priorities, keeping focus, coping with differences in disciplines and cultures, adopting a Zero-Defect attitude and preventing any stakeholder's complaints. It integrates Planning, Requirements Management and Risk Management into Result Management. The basic secret is the time-honored Plan-Do-Check-Act- or Deming-cycle.

Plan-Do-Check-Act

The magic ingredient for risk mitigation and successfully running any project, or for that matter, any activity, is repeatedly and frequently going through the Plan-Do-Check-Act- or Deming-cycle (Figure 6, Deming 1986):

- We **Plan** *what* we want to accomplish *and* how we think to accomplish it best.
- We **Do** according to the *Plan*.

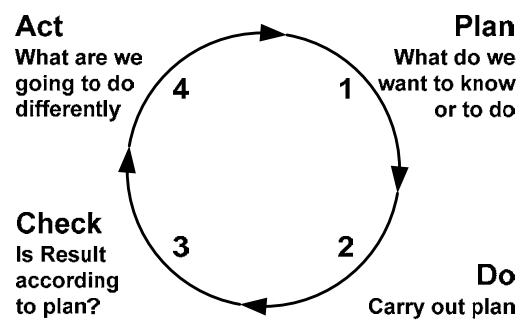


Figure 6: PDCA or Deming cycle

- We **Check** to observe whether the result from the *Do* is according to the *Plan*. If the result is ok: what can we do better. If the result is not ok: how can we make it better.
- We **Act** on our findings. *Act* produces a renewed strategy *and* the decision to follow that strategy.

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

Do is never a problem: we *Do* all the time. We *Plan* more or less, usually less. For *Check* and *Act* we have no time because we think we want to go to the next *Do*. Taking a closer look at what really is happening we can see that *Check* is often done tacitly: we seem to be quite aware what is going wrong. The real problem is that we don't *Act*: taking what we know and *actively doing something about* it. Anytime people complain about something or somebody, when they keep saying "Yes, but... ", they are stuck in the *Check*-phase. If we say: "That's a *Check*. What would be an *Act*: What could we do about it?" the same people prove to be well capable of proposing ways to solve the problem. The problem is that we never ask that question and subsequently *Act*.

Once people learn to actively *Act* on *Checks*, most problems will be solved almost effortlessly, because most problems are not really difficult to solve. The real problem is not *how* to solve the problem, but rather *deciding* to do something about it and then *actually doing it*. Many people heard about PDCA, but fail to imagine the power of PDCA until they actually learn how to use it properly.

It may be clear that we use PDCA to control risks: as soon as we see a risk (*Check*), we devise a strategy to mitigate the risk (*Act*). Then we *Plan* and conduct actions (*Do*) and *Check* whether the actions effectively and efficiently improved the situation. If the situation improved, we try to improve even more. If not, we change the strategy again. We always *Act* as necessary.

By introducing *mutations* in the Act-phase of PDCA rapidly and frequently, keeping what works better and shelving what works less, we force rapid evolution. Hence we call our approach Evolutionary.

If we appropriately organize projects in very short Plan-Do-Check-Act (PDCA) cycles, constantly selecting only the most important things to work on, we will most quickly learn what the real requirements are and how we can most effectively and efficiently realize these requirements. We spot problems quicker, allowing us more time to do something about them.

Is it then only positive? No negative effects to consider? That's actually the power of Evo: Evo itself provides the very mechanism to cope with any negative issues or risks: using PDCA, we recognize negative things and *do* something about them. The only remaining negative things are those things we don't consider important enough to do something about for the moment.

Some people fear that all this tuning will take a lot of extra time. Evo projects, however, prove to be significantly faster than other projects. In only a few weeks time we see a 30% productivity improvement when projects start working the Evo way (based on experience of the author after coaching more than 60 projects in the past 6 years). In Evo, we never do things if they take more time than necessary.

Evo is not only both iterative (using multiple cycles) and incremental (we break the work into small parts), but above all Evo is about quickly *learning* how to do things

better, using PDCA. We systematically and proactively anticipate risks before they occur and work to prevent them. We may not be able to prevent all the problems, but if we prevent most of them, we have a lot more time to cope with the few problems that slip through, before they materialize as a Stakeholder problem.

Using PDCA we are constantly optimizing:

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

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

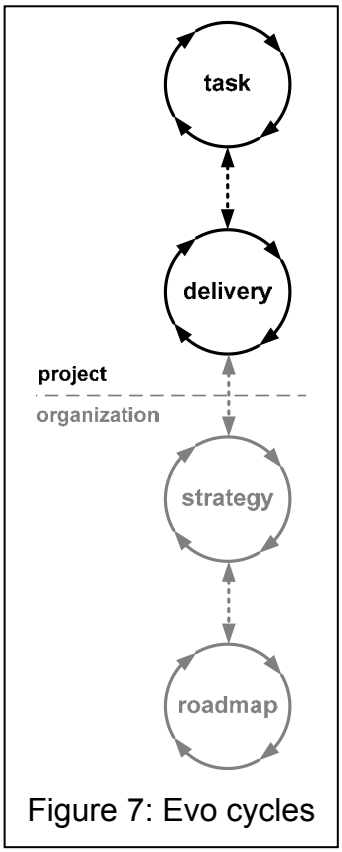


Figure 7: Evo cycles

Evo elements

All elements of Evo address typical project risks, such as:

- Schedule risk: delivering at the wrong time
- Not living up to our promises because of unrealistic optimism
- Promising more than we can do
- Making more mistakes than necessary because of fatigue
- Delivering the wrong things and delivering at the wrong time, not making the customer happy and more successful than before
- Others causing us to fail
- Gold Plating (doing more than necessary)
- Handling interrupts: Losing time on *seemingly* important things
- Coping with suppliers beyond our control
- Doing the wrong things for too long
- Trying to do more than we (as humans) can handle in a certain amount of time

In Evo, we use several learning cycles (

Figure 7):

- The weekly **TaskCycle** is used for organizing the work, optimizing estimation, planning and tracking. Every week we check what the most important tasks are, estimate the effort needed to complete these tasks completely and commit to the most important tasks we can complete during the week. We plan 2/3 of the available time as *plannable* time, leaving 1/3 as *unplannable* time for all the small interrupts that will occur anyway (email, phone calls, helping each other, …), allowing people to succeed finishing what they committed to. We use TimeBoxing helping people to focus on what is really necessary: doing less without doing too little. We constantly check whether we are doing the right things in the right order to the right level of detail. We optimize the work effectiveness and efficiency. We found that people in projects can very quickly learn to change from optimistic estimators into *realistic estimators*, *if* we are serious about time. Once we master realistic estimation, we can better predict the future and we can deliver on time as

agreed. We learn to promise what we *can* do, and then to *live up to our promises*. Estimating in TimeBoxes also relieves people from the need for tracking. All these details of the TaskCycle are designed to control *schedule risk*.

- The **DeliveryCycle**. Every two weeks or less, we deliver useful value to stakeholders: juicy bits to enthuse them to provide feedback, intermediate results that make them already successful now, or at least we deliver something that will provide the most important feedback momentarily possible. The DeliveryCycle is used for optimizing the requirements and checking the assumptions. We constantly check whether we are moving to the right product results. DeliveryCycles focus the work organized in TaskCycles. This way we control the *risk of not delivering the right results*. An important question we ask ourselves when defining a Delivery: "*What* do we have to deliver in two weeks, to *Whom* and *Why*?"
- In larger projects we see many more cycles developing, for example: parallel DeliveryCycles, shorter DeliveryCycles, and additional longer DeliveryCycles, like Releases or Intermediate Deliveries.

At the organizational or enterprise level, we use:

- The **Strategic Objectives Cycle**. In this cycle we review the strategic objectives at the organizational/enterprise level and check whether what we do still complies with the objectives. This cycle may take 1 to 3 months.
- The **Enterprise 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.

Many people think that the cycle periods we use in Evo are too short. Some people even think that such short cycles are impossible. This is one of the more difficult things to learn when changing to the Evo way: It always can be done. It requires a paradigm shift that is not difficult, but doesn't come naturally. So it has to be taught.

One reason for applying such "short" cycles is combating the risk of *doing the wrong things for too long*. The later we find out that we are doing the wrong things, the longer the wrong things have endured and the more we have to repair or redo. This way, short cycles do not cause more work (doing the Check/Act more often), but rather cause less work (having to repair less). Another reason for applying short cycles is that people simply cannot really oversee and organize longer periods of time in detail. For controlling longer periods of time, we have a separate technique, called *TimeLine*.

TimeLine (Figure 8) is used to keep control over the total project, making sure that at the FatalDate or at the FatalBudget, we will have delivered the best possible value. We optimize the order of Deliveries in such a way that we approach the product result along the shortest path, with as little rework as possible. TimeLine is also used to

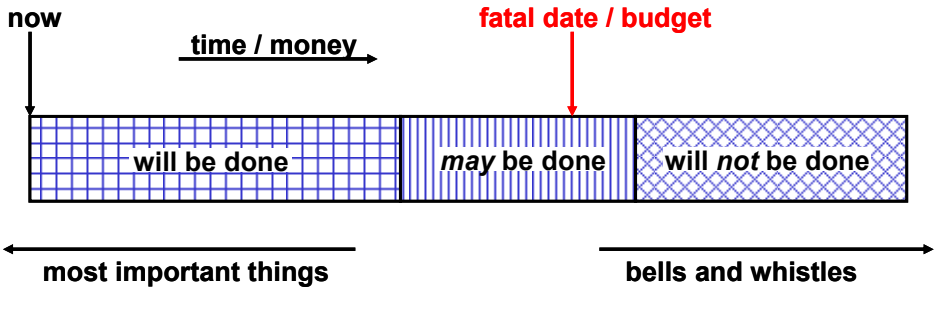


Figure 8: TimeLine

dynamically adjust the order of deliveries to the available resources. We treat FatalDates seriously and count back when we should have started to achieve what is required. Because the customer usually wants more than he can afford, it is important to know what we, at the FatalDate, surely will have done, surely not have done and what we may have done (after all, estimation is not an exact science). Better than to tell the customer at the FatalDate that we didn't succeed in the impossible, we rather tell him as soon as we possibly know. Then we can together decide what to do with this knowledge. Every day we know a potential problem earlier, we have a day more to do something about it. With TimeLine, we control the *risk of being late*, delivering optimum value in the limited available time. During the project, we continuously keep using TimeLine to predict the outcome of the project based on the ongoing development of knowledge. In Evo we don't use the concept of *Earned Value*. After all, how can we talk about Earned Value if the Requirements are evolving…? In many conventional projects we even see that Earned Value degrades to *Money Spent*. In Evo we rather use *Value Still to Earn*.

Requirements in Evo

Although the Requirements should be stable for best results, they aren't. During a project, the developers learn, the customer and the other stakeholders learn, and the market changes. So, because requirements change is a *known risk*, we *provoke* requirements change as quickly as possible, preferably before they are implemented. Because the customer and other stakeholders usually cannot very well express what they really need, we use several techniques to find out what they do need, first developing the problem, before we start developing a solution.

Every project has many Stakeholders. A Stakeholder is anyone having a stake in the requirements: customer, users, and many others, including the developers. If at the end of the project we realize that we forgot an important stakeholder, we are in trouble. If we find a requirement without a stakeholder, either it isn't a requirement, or we haven't identified the stakeholder yet. If we don't know the stakeholder, who will pay for our work, how would we know we are implementing the right things, and how would we know when we are ready?

Requirements are divided in:

- **Functional Requirements**, scoping the project. The Functional Requirements describe what we will improve in this project. We choose this particular set of functions to improve, because a different set will yield less benefit.
- **Performance Requirements** defining how much the functionality will be improved. Note that all the functions are already there. With our new product, people should, for example, be able to do what they did before more quickly, making them more productive. The Performance Requirements are the most important requirements and have the most impact on project time and cost. Therefore it is imperative to pay adequate attention to these requirements.
- **Constraints** defining what we are *not* allowed to do, e.g. for legal, environmental and moral reasons, or what we are simply not supposed to do.

Performance Requirements shall always be numerically defined, otherwise we will not be able to determine whether we have achieved the required performance. Performance Requirements are an important driver for choosing the appropriate architecture and if they are not stated early, the chosen architecture will usually prohibit achieving them later. If the airport is opened and the PA-System.Audibility turns out to be bad, it may cost a lot of acoustic redesign once we define the

out to be bad, it may cost a lot of acoustic redesign once we define the Audibility-requirement e.g. as “96 of 100 people waiting for departure can reproduce the message”. If a Performance Requirement cannot be defined numerically (is Maintainability 3 or 7?) we call it a complex concept, which has to be decomposed into components which can be numerically defined, like e.g. “Maintainability.MTTR < 15 minutes” and “Maintainability. DownTimeDuringUpdate < 1 sec”.

RQ27:	Maximum Response Time
Scale:	Seconds between <asking> for information and <appearance> of it.
Meter:	Add a function to the software to measure the maximum response time value and the range of values per <working day>.
Benchmarks:	
Past:	3 sec (our previous product)
Current:	0.6 sec [competitor y, product x, 2006] ← Marketing Survey, Jan 2006
Record:	0.2 sec [competitor x, product y]
Wish:	0.2 sec [2008] ← customer's head of R&D, 19 Feb 2005, <document ...>
Note:	Less than 0.2 sec is not noticed by the user, so there is no use in trying to be better than 0.2 sec
Requirements:	
Fail:	1 sec [99%] ← project-contract
Fail:	1.5 sec [100%] ← project-contract
Goal:	0.5 sec ← project-contract

Figure 9: Using Planguage to specify Performance Requirements

Specifying requirements

For specifying performance requirements, we use Planguage, as proposed by (Gilb 2005). In the example in Figure 9, we show some basic elements of this method. To be able to specify numerical values of the performance requirement, we need a defined *Scale*. The *Meter* defines how we measure the values on the scale.

Benchmarks define the playing field. Some examples are:

- **Past**: The value in our previous product. We won’t have improved if our new product has the same specification.
- **Current**: The current state of the art. If we don’t achieve this level, we won’t beat the competition, which would constitute a risk.
- **Record**: This is a level that will be hard to beat (like an Olympic record). It probably will cost a lot to achieve this. Could very well be much more than our customer can or is willing to afford.
- **Wish**: This is a possible future requirement. We are not planning to achieve this level now. It may be too costly to achieve with the current state of the art, but it is what we actually would like, once feasible. The customer is not prepared to pay for this in the current project.

Then we describe the Requirements, with at least a Fail and a Goal value:

- **Fail**: If we do not achieve this level, the project fails.
- **Goal**: This is the level we expect to achieve in this project. When we have achieved it, we are done.

The power of generating requirements in this fashion is that it stimulates greatly the communication and understanding of the requirements and we often see that perceived initial requirements quickly change into other, more appropriate requirements, reducing the risk that we start working on implementing the wrong requirements. During development the architecture and design tries to cover the set of conflicting (for example: performance and budget are always in conflict) requirements as best as possible. Having a range between Fail and Goal, leaves room for intelligent compromises.

Engineers are trained to achieve planned results *by design* (Figure 10). Sometimes,

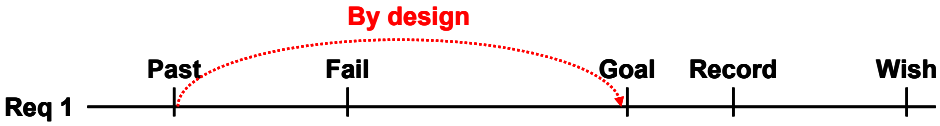


Figure 10: Engineering is achieving a goal *by design*

however, we reach a goal by improving different parts of the system, one step at the time. Many developers are used to trying to accomplish as much as possible in one step. In Evo, we always select the smallest step possible. If this step later turns out not to be the right step, we have to redo as little as possible. And the step that takes the

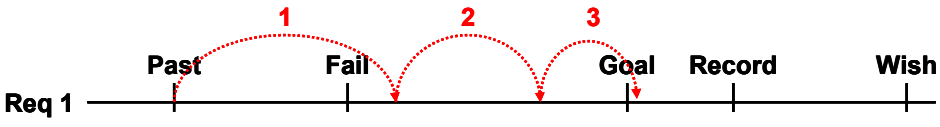


Figure 11: Sometimes we reach the goal by improving different parts of the system, *one step at the time*

least time leaves us the most time for whatever we still have to do.

Figure 11 shows how we are safe after one delivery step (better than Fail: at least we don't fail). After two more deliveries we reach the Goal value, indicating that we have achieved our goal and that we don't gain anything if we continue. Hence we stop. This way, we mitigate the *risk of Gold Plating*, which is doing more than necessary.

In real projects, we have to cope with many conflicting requirements at the same time. In one Evolutionary delivery step, we work on Requirement 1 (Figure 12: step 1),

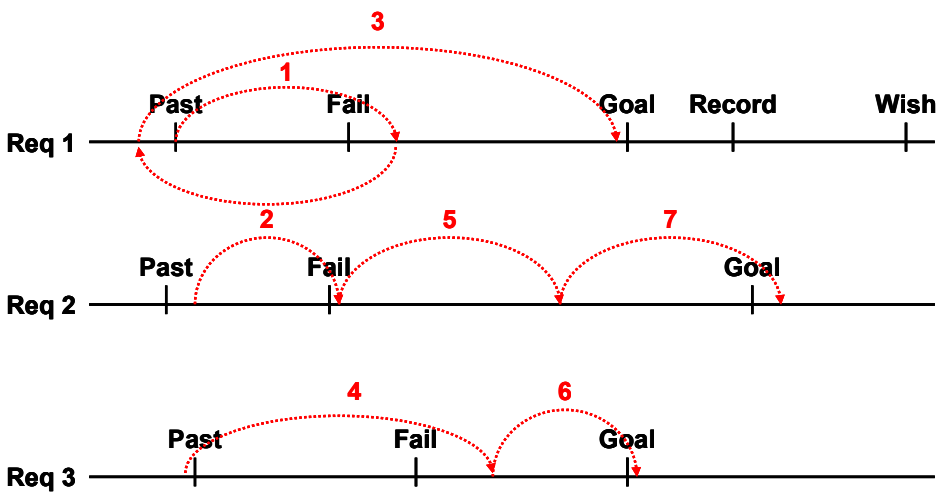


Figure 12: In practice, we have to satisfy many conflicting requirements simultaneously

getting past the Fail level. Therefore, in the next DeliveryCycle, we better first work to get another requirement beyond the Fail level (step 2). In some cases, an improvement of one performance may adversely affect another performance, which we may improve in the next step (step 3). In similar fashion we deliver step by step until all requirements are at the Goal level, or until the budget in time or money is depleted.

In Evo we never overrun the budgets. As soon as we are beyond the "safe" Fail levels for all requirements, we can basically stop at any time, for instance if the customer

decides that time-to-market is more important than further improvements. The Evo Requirements Engineering Approach addresses the *risk of delivering the wrong things and delivering at the wrong time*.

Active Synchronization

If we are working the Evo way, somewhere around us is the bad world, where people are not yet accustomed to living up to their promises. Software people may need hardware to test their software. Hardware people may need test software to test their hardware. Other disciplines may have to deliver to us, or need our results. You may be collaborating with people at other places of the world.

If you are waiting for a result outside your span of control, there are three possible cases:

1. You are sure they'll deliver Quality On Time (the right results at the time agreed)
2. You are not sure
3. You are sure they'll not deliver Quality On Time (because they never did)

Note:

- Your Evo project behaves like case 1
- From other Evo projects you can expect case 1
- If you are not sure (case 2), better assume case 3

In cases 2 and 3: Don't wait until you get stuck not receiving the agreed result on time. You know you won't if you don't Actively Synchronize, so: Do something! Go there! This has three advantages:

1. Showing up increases your priority
2. You can resolve issues which otherwise would delay delivery
3. If they are really late, you'll know much earlier

With Active Synchronization, we control the *risk of others causing us to fail*.

Interrupts

One of the potential risks of loosing time is interrupts. In Evo, we only work on planned tasks, never on undefined tasks. In case a new task (or a new requirement) suddenly appears in the middle of a Task Cycle, we call this an Interrupt.

Assume the boss comes in and asks us to paint the fence. We don't say Yes, but we also don't say No. After all, painting the fence may be more important than anything we have currently planned, if an important customer would turn his heels when he sees the shabby fence. Instead, we follow the Interrupt Procedure:

- Define the expected Results of the new Task properly. What is the actual risk that the shabby fence causes trouble? Should we thoroughly grind, ground and paint the fence, or buy a new fence that doesn't need paint, or just put a quick layer on the old fence, just covering the dirt?
- Estimate the time needed to perform the new Task, to the level of detail really needed.
- Go to the task planning tool (many Evo projects use the Evo Task Administrator tool (ETA 2004)).

- Decide which of the planned Tasks have to be sacrificed (up to the number of hours needed for the new Task).
- Weigh the priorities of the new Task against the Task(s) to be sacrificed.
- Decide which is more important.
- If the new Task is more important: replan accordingly.
- If the new Task is not more important, then do not replan and do not work on the new Task. Of course the new Task may be added to the Candidate Task List, to be considered later.
- Now we are still working on planned Tasks.

But couldn't this be delaying the work we originally planned? Yes, of course it could. But we deal with the consequences of the change in the plan. We *Act*. We don't *let* things happen randomly, we rather *control* how they happen. If some requirements become more important than others, the order of what we do *should* change. Priorities do change all the time, so the thing is to dynamically reprioritize as needed. Revisiting TimeLine will tell us what the consequences will be for what will be done, what will not be done, and what may be done at the FatalDate.

We simply cannot do more than we can do. We don't try to do the impossible. All we can do is making sure that looking back we always can say we did the best possible job. With the Interrupt procedure, we control the risk of *losing time on **seemingly** important things*.

Boehm's 10 top software risk items

Barry Boehm described 10 software risk items (Boehm 1991), which probably still are considered risks today, also for non-software projects. Let's check whether and how we address these risk items the Evo way:

- **Personnel Shortfalls**. We have a certain number of people available in our organization. At the organizational level (Figure 7, in gray), we compare the priorities of all the work we could do with the available resources. If a certain project does not get the appropriate number of people needed for a certain development load, this will be only because other projects create even more value than this project. With TimeLine the project determines what it can do with the available resources and if this is less than needed, they inform management about the consequences. This information is input to the organizational prioritization process. This isn't a risk, it's a choice.
- **Unrealistic schedules and budgets**. If the requirements aren't clear (which is usually the case; be honest!), any schedule will do. If the requirements change anyway, how can we talk about realistic or unrealistic schedules and budgets? We take time and budgets as a given and fill them delivering the best possible value. If, within the available time and cost we can't deliver sufficient value, we won't even start. We constantly update the TimeLine in order to predict what at a certain date will be done, won't be done and may be done, and we take the consequences. It is our experience that we can quickly change People in projects from optimistic estimators into realistic estimators, making them able to live up to their promises. This way, they have *facts* to explain management about the realisticness of schedules. In such an environment, managers who keep asking for unrealistic schedules shouldn't survive. Or, if managers insist in unrealistic schedules (*Check*), either they should be educated (*Act*), or they *want* the project

to fail. If they want the project to fail, they should better tell us, because then we can do that much more efficiently.

- **Developing the wrong functions and properties**. The Evo requirements process deals with this issue. Frequent stakeholder feedback is used to optimize the requirements and check the assumptions, making sure were are developing the right things.
- **Developing the wrong user interface**. Same as previous. Because Evo requirements are stated in terms of stakeholder success, which may include improvement of user productivity, the developers will make sure that the user interface supports those requirements. We check our assumptions about the user interface with early deliveries and check whether we are achieving the real requirements or not. If necessary, we *do* something about it.
- **Gold-plating** is suppressed because we deliver as per requirements, specified by Planguage. When we achieve goal values, we are done. Normally, people tend to do more than necessary, especially if it's not clear what should be done. Making it more clear is a big time-saver.
- **Continuing stream of requirements changes**. Requirements do change because *we* learn, *they* learn and the market changes. If we would deliver according to obsoleted requirements, we won't secure customer success, forsaking the Goal of the project. So, we expect requirements changes and have a process to deal with them. We even *provoke* requirements change as soon as possible, preferably before we implemented any requirements that had to be changed. If we are not sure about a requirement, we do as little as needed, just enough to find out what the real requirement is, in order to redo as little as possible in case our assumptions were wrong. TimeLine is used to guard what we will and will not deliver at the FatalDate. We *Act*, if *Check* indicates a problem.
- **Shortfalls in externally furnished components.** We use Active Synchronization to stay on top of this issue. We know that when our FatalDate has come and we don't deliver, there is no point in finger pointing: we simply failed! Well before our FatalDate we would have got deliveries from the external suppliers, knowing early about any problems. Any day we know a problem earlier, we have a day more to do something about it.
- **Shortfalls in externally performed tasks**. Same as previous. We request regular Evo deliveries from our external suppliers, so any potential problems will show up quickly. Note that the possibility of shortfalls influences the order of deliveries: highest risks first. If a risk turns out for the worst at the end of the project, we are in trouble. We don't want to get into trouble, so we *design* the order of whatever we do in our project to minimize the chance for trouble and optimize the use of our time.
- **Real-time performance shortfalls**. Come on. That's simply a Performance Requirement and then an engineering issue. Are we amateurs?
- **Straining computer-science capabilities**. In Evo, we plan to do the right things, in the right order, to the right level of detail. We don't start with the easy things, but rather with those things we are not yet sure of. And if we find out that the necessary requirements cannot be met within an acceptable budget of time and cost we report this to management and the customer and discuss what we do with this knowledge.

Concluding: all of these so called risks are not really risks. There are adequate processes to cope with these issues in a responsible way. Again: we aren't amateurs,

are we?

The biggest risk

The biggest risk is the risk that we are overlooking something:

- It's within our span of control, but we don't recognize it.
- It's not within our span of control, but we don't anticipate, or we don't do enough to avoid the problem to occur (we should have Actively Synchronized to avoid this, but missed it).

The trick is to be ahead of problems, before they occur. We don't ostrich; we actively take our head out of the sand. If somebody complains, we're too late. If the FatalDay is there, excuses and finger pointing are irrelevant. If we don't deliver, we fail. We don't want to fail, so we do whatever (ethical) we can to avoid failure. Why? Because we want to achieve the best possible results in the shortest possible time. Why? Simply because that's what we like.

Conclusion

A lot of so-called risks in projects aren't really risks: we may not know when exactly, but they almost always do happen. If we devise good, practical techniques and processes to cope with these "risks", we can minimize the impact, so that they don't jeopardize the successful result of our project. Evo provides these techniques and processes, continuously acting proactively and making sure that we are delivering the right things at the right time. If we have mitigated *by design* most of the risks that usually plague projects, then we have much more time left to handle the unexpected risks we still have to deal with. Evo provides the technique and the attitude to deal with the unexpected risks as well, applying PDCA rigorously and frequently.

We design not only what we agree to deliver (the product), we also *design* the way we work to succeed in our goal (the project). Evo seeks to optimize this process, by constant learning to doing things better. In this process, risks are not handled separately, but as an integral part of running a project. Evo is full of small details designed to ensure success, some examples of which are described in this paper. The process itself being evolutionary as well, Evo constantly optimizes its own ways. Some people fear that all this analyzing, learning and optimizing may take a lot of extra time. In practice we see that it saves time: A project adopting these methods for the first time, generally completes successfully in 30% shorter time. Nothing of the above is merely theoretical. It has been tested, honed and proved by the author in the practice of more than 60 projects in various environments and cultures.

References and further reading

Boehm, Barry W., “Software Risk Management: Principles and Practices”, IEEE Software, vol. 08, no. 1, pp. 32-41, Jan/Feb, 1991.

Deming, W.E.: *Out of the Crisis*. MIT, 1986.
also: Walton, M: *Deming Management At Work*. The Berkley Publishing Group, 1990.

ETA: *Evo Task Administrator*, 2004. Tool for administering Tasks in Evo projects. http://www.malotaux.nl/nrm/Evo/ETAF.htm

Gilb, Tom: *Competitive Engineering* Elsevier, 2005.

Incose, *Systems Engineering Handbook*, Version 3, June 2006, Figure 7-6.

Larman, Craig, Vic Basili: “Iterative and Incremental Development: A Brief History”, *IEEE Computer*, June, 2003.

Malotaux, Niels: *Evolutionary Project Management Methods*, 2001. Issues and first experience, introducing Evo at a large company. http://www.malotaux.nl/nrm/pdf/MxEvo.pdf

Malotaux, Niels: *How Quality is Assured by Evolutionary Methods*, 2004. Practical details how to implement Evo, based on experience in some 25 projects in 9 organizations. http://www.malotaux.nl/nrm/pdf/Booklet2.pdf

Malotaux, Niels: *Optimizing the Contribution of Testing to Project Success*, 2005. How to apply the Evo ideas on the testing process to iterate more quickly towards Zero Defects. http://www.malotaux.nl/nrm/pdf/TestingEvo.pdf

Malotaux, Niels: *Optimizing Quality Assurance for Better Results*, 2005. Similar to the previous, but targeted at non-software QA. http://www.malotaux.nl/nrm/pdf/Booklet3QAV01.pdf

Rafele, Carlo, David Hillson, Sabrina Grimaldi: “Understanding Project Risk Exposure Using the Two-Dimensional Risk Breakdown Matrix”. *PMI Global Congress Proceedings* - Edinburgh, Scotland, 2005.

Biography

Niels Malotaux is, since 1998, an independent consultant and project coach, teaching immediately applicable methods for delivering Quality On Time to projects and organizations. He has over 30 years experience in designing electronic and software systems, at Delft University of Technology, in the Dutch Army, at Philips Electronics, and 20 years leading his own systems design company. Niels researches how human behavior affects the results of what people do. He developed both techniques to optimize the results of what people do in projects and how to teach people this successful approach very effectively and efficiently. Since 2001 he taught and coached over 60 projects in some 18 different organizations in the Netherlands, Belgium, Ireland, India, Japan and the USA, creating a wealth of experience about what really works in practice. He is a frequent speaker at conferences and published four booklets on the subject. Niels can be contacted at niels@malotaux.nl. More can be found at www.malotaux.nl/nrm/English.

Controlling Project Risk by Design

Niels Malotaux
N R Malotaux - Consultancy
niels@malotaux.nl - http://www.malotaux.nl/nrm/English
Copyright © 2007 by Niels Malotaux. Published and used with permission.

Abstract. A lot of risks that plague projects have a high probability to occur: e.g. the risk that we don't deliver the right things or deliver late, and the underlying causes of these problems. This calls for proactively anticipating potential problems and organizing our projects in such a way that the probability of the impact is minimized. The Evolutionary Project Management (Evo) approach is just doing that, constantly being aware of what could go wrong and preventing it from going wrong, *by design*. In stead of assuming a theoretical model of how humans "should" behave, Evo studies actual human behavior and strives to make optimum use of how humans actually behave. After all, systems are conceptualized, designed, realized, maintained, used, and to be tolerated by people, so if Systems Engineering and Design ignores, forgets to include, or incorrectly assumes how people behave in all of these activities, the results can be devastating. Opposing what's in our genes is a lost battle.

In this paper we will first investigate prevailing risk management. Then we show how Evo practices are designed to successfully mitigate typical risks in projects. The techniques discussed are not merely theoretical ideas, but have been tested, honed and proved by the author in the practice of more than 60 projects in the past 6 years, in various environments and cultures.

Introduction

If we do nothing, the risk that we won't accomplish a certain thing is 100%. In order to accomplish what we want to accomplish, we organize a project, and at the end of the project the risks are to be reduced to an acceptable level. The level will never be zero, as, for example, a meteorite could strike our result just before delivery of the project result.

A lot of risks that plague projects have a high probability to occur: e.g. the risk that we don't deliver the right things or deliver late, and the underlying causes of these problems. This calls for proactively anticipating any potential problems and organizing our project in such a way that the probability of the impact is minimized. The Evolutionary Project Management (Evo) approach is just doing that, constantly being aware of what could go wrong and preventing it from going wrong, *by design*. In stead of assuming a theoretical model of how humans "should" behave, Evo studies actual human behavior and strives to make optimum use of how humans actually behave. After all, systems are conceptualized, designed, realized, maintained, used, and to be tolerated by people, so if Systems Engineering and Design ignores, forgets to include, or incorrectly assumes how people behave in all of these activities, the results can be devastating. Opposing what's in our genes is a lost battle.

In this paper we will first investigate prevailing risk management. Then we show how Evo practices are designed to successfully mitigate typical risks in projects. The techniques discussed are not merely theoretical ideas, but have been tested, honed and proved by the author in the practice of more than 60 projects in various environments and cultures. If we have mitigated most of the risks that usually plague projects *by design*, then we have much more time left to handle the unexpected risks we still have to deal with. Evo provides the technique and the attitude to deal with those risks very effectively as well.

The goal of a project

In order to know whether we succeed in projects, we'll first have to define the main Goal of our efforts in projects:

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

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

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

Furthermore, let's define a Defect as:

The cause of a problem experienced by stakeholders of the system, ultimately by the customer.

If there are no defects, we'll have achieved our goal. If there are defects, we failed. Example: the Public Address system used for making announcements to the public in airports or train stations is in many cases hardly audible. Possible defects: bad equipment, bad acoustics, bad speaker. In airplanes the announcements are regularly unclear because the speaker speaks too quickly, not well articulated or in a nice but unintelligible dialect. Defect: insufficient education. Root cause: insufficient management attention. Ultimate cause: insufficient management education.

Being late or over-budget is a defect, as long as it is experienced as a problem. If there is a potential defect, but none of the stakeholders ever experiences a problem because they never use a certain part of the system, then we don't call it a defect. We may ask ourselves why we built that part of the system anyway and may call building that part a defect, because we spent time (= money) on building parts of a system that are not providing return on investment.

Within this definition-framework, Risk is:

An event that may cause a defect.

Prevailing Risk Management

Conventional Risk Management principles are quite straightforward. A common definition of Risk is:

An uncertain event that, if it occurs, has a negative effect on project success.

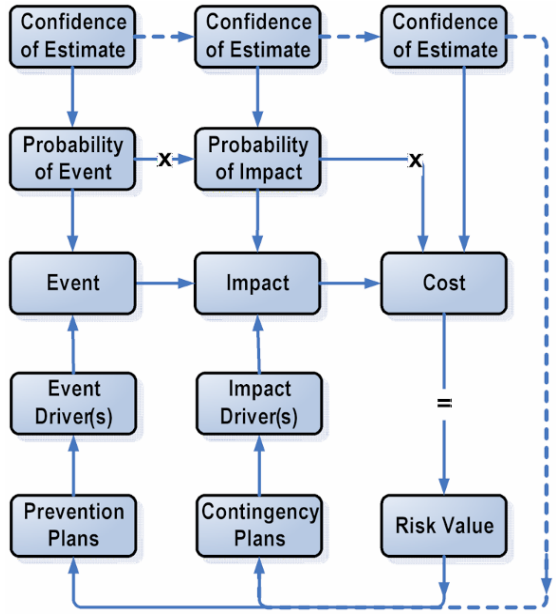


Figure 1: Risk Model

The measure of uncertainty is the probability that the event may occur. If the probability is 0%, it's not a risk. If the probability is 100%, it isn't a risk, but an issue or a problem. The aim is to proactively deal with risks before they become problems. If we deal with 80% of the risks before they become a problem, we have a lot more time to deal with the remaining 20%.

Some people include positive risks in the definition. In practice we see that this causes a lot of confusion, so we prefer to use *opportunity* instead of *positive risk*, reserving *risk* for negative effects.

If a risk event occurs, there is a probability that it impacts our success: if there is an earthquake in Japan, will it impact our project in Paris? If it does impact our success, there is Cost involved (Figure 1).

The product of the Probability that the risk event occurs (P_e), the Probability of the Impact hitting us (P_i) and the Cost if we are impacted by the event (C), is called the Risk Value (V_R): $V_R = P_e * P_i * C$

Because the Probabilities and the Cost are estimates, and often even rough guesstimates, it is better to include some awareness of the (in)confidence of the estimates, to allow for worst-case judgments.

Based on the Risk Value, we may decide to plan for Risk Prevention, preventing the risk event to occur, and/or for Contingency, to minimize the impact if the event occurs. A risk of using this Risk Value product (Figure 2a, ref (Incose 2006), and 2b), is that a very harsh consequence, with a very small likelihood to occur, may be ignored as it may be perceived as a low risk value, while if it occurs, it had better been treated as important. Example: if software for an emergency procedure in an airplane or space shuttle is not tested because other risks seemed more important, it may fail in the (*perceived* unlikely) case that the emergency does occur.

An example of pushing the mathematical treatment of risk too(?) far is the table shown in Figure 3 (Rafele 2005). The text is not well readable, but that's not relevant. Work packages are on the left and risk sources are shown at the top. In the matrix, the Risk Values are computed per work

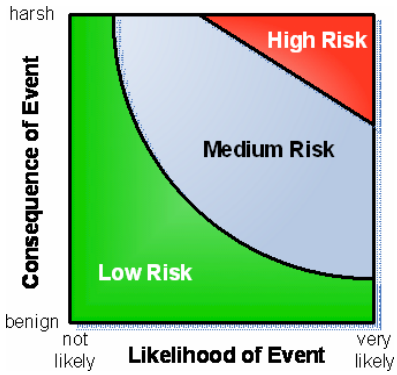


Figure 2a: Problem: very harsh consequence is treated as low risk

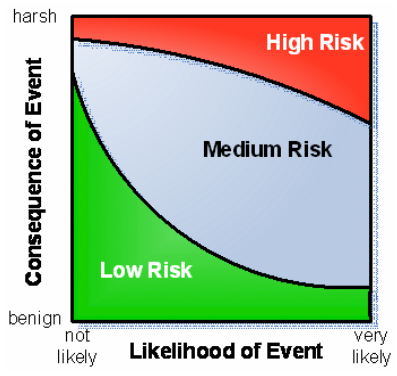


Figure 2b: Is this better?

package and risk source. At the right and bottom, the Risk Values in rows and columns are summed, leading to an order of priority of the risk of a certain work package and a certain risk source. It occurs to me that this may be nice theory, but that translating the risks in just numbers, where apples and oranges have been added, obscures what the risks really are all about. We may only use this technique to feed the decision makers with additional insight presenting them the full table rather than just the computed Risk Values. Then they can base their risk mitigation strategy on more than just bare numbers.

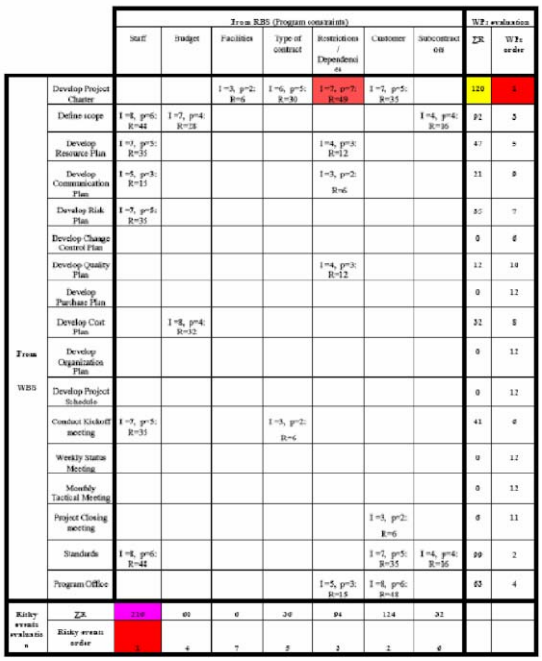


Figure 3:
Mathematical risk management can be risky

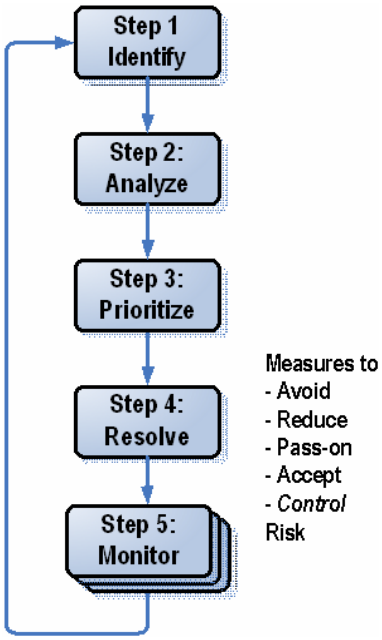


Figure 4:
Iterative Risk Management Process

Risk Management (Figure 4) is done in cycles, Identifying, Analyzing, Prioritizing, Resolving and Monitoring risks. Risks are either avoided, reduced, passed on to others, or accepted. I would like to add here: or *controlled by design*. Passing on to others may be passing the risk to sub-contractors, who may, however, not be capable to run the risk. If they fail, we still fail, so that's a good solution only if the sub-contractor is better equipped to handle the risk than us.

Summarizing, prevailing Risk Management is quite straightforward and important to assist people taking the right preventive measures to proactively deal with risks in order of importance. However, calling every event that can jeopardize our project result a risk, obscures opportunities to control the mitigation of the effects more effectively. It is more useful to call most of the risks by their proper name. That's what we will do in the following chapters.

Risks in Projects

Basic risks in projects are:

- The result of the project is not right (according to the goal)
- It is too late
- It costs more than necessary

Although every project is unique, a lot of what we do in projects is always the same:

- Every project is done by people. People react in certain ways, which, once recognized, are quite predictable, although this "real" human behavior is ignored in many project management approaches, where only some theoretical human behavior is assumed. Often, theory doesn't really work as expected in practice.
- Many activities are the same in every project and can be organized by repeatable processes.
- Only a very small part of the project is really unique, causing specific risks.

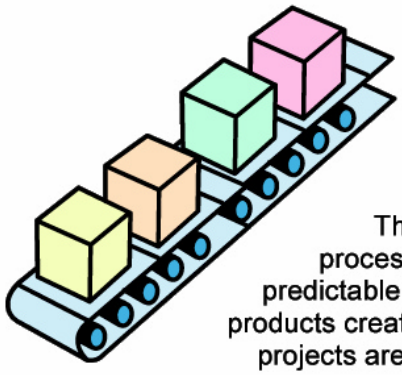


Figure 5: The project process is quite predictable, only the products created in the projects are different

This means that we'd better call the predictable risks (known risks) by their proper name and control them by repeatable processes and only treat the few new-product related risks by a special risk management process (Figure 5). In the following part we will show how most project risks can be controlled *by design*, by proper process rather than calling for separate risk management.

Evolutionary Project Management Methods (Evo)

Evolutionary Project Management is a set of methods and processes, including a certain *attitude*, that allow people to routinely complete projects successfully on time, or earlier. Researching the root-causes of project problems, the author is constantly designing and optimizing methods to overcome these problems, as well as optimizing the process of introducing these methods into projects. Because Evolutionary is a long word, we use the abbreviation Evo, as a label for the current set of methods. Being routinely successful implies that we succeed in systematically controlling the risks threatening our projects.

Elements of these methods are solving the discipline problem, exploiting our intuition mechanism, continuously balancing priorities, keeping focus, coping with differences in disciplines and cultures, adopting a Zero-Defect attitude and preventing any stakeholder's complaints. It integrates Planning, Requirements Management and Risk Management into Result Management. The basic secret is the time-honored Plan-Do-Check-Act- or Deming-cycle.

Plan-Do-Check-Act

The magic ingredient for risk mitigation and successfully running any project, or for that matter, any activity, is repeatedly and frequently going through the Plan-Do-Check-Act- or Deming-cycle (Figure 6, Deming 1986):

- We **Plan** *what* we want to accomplish *and* how we think to accomplish it best.
- We **Do** according to the *Plan*.

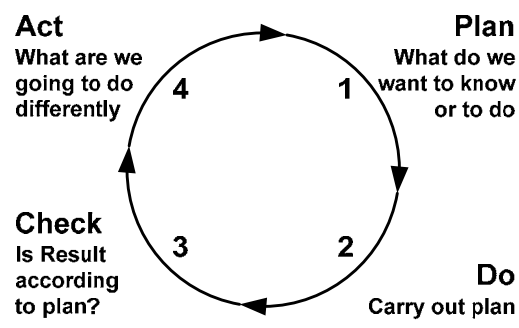


Figure 6: PDCA or Deming cycle

- We **Check** to observe whether the result from the *Do* is according to the *Plan*. If the result is ok: what can we do better. If the result is not ok: how can we make it better.
- We **Act** on our findings. *Act* produces a renewed strategy *and* the decision to follow that strategy.

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

Do is never a problem: we *Do* all the time. We *Plan* more or less, usually less. For *Check* and *Act* we have no time because we think we want to go to the next *Do*. Taking a closer look at what really is happening we can see that *Check* is often done tacitly: we seem to be quite aware what is going wrong. The real problem is that we don't *Act*: taking what we know and *actively doing something about* it. Anytime people complain about something or somebody, when they keep saying "Yes, but... ", they are stuck in the *Check*-phase. If we say: "That's a *Check*. What would be an *Act*: What could we do about it?" the same people prove to be well capable of proposing ways to solve the problem. The problem is that we never ask that question and subsequently *Act*.

Once people learn to actively *Act* on *Checks*, most problems will be solved almost effortlessly, because most problems are not really difficult to solve. The real problem is not *how* to solve the problem, but rather *deciding* to do something about it and then *actually doing it*. Many people heard about PDCA, but fail to imagine the power of PDCA until they actually learn how to use it properly.

It may be clear that we use PDCA to control risks: as soon as we see a risk (*Check*), we devise a strategy to mitigate the risk (*Act*). Then we *Plan* and conduct actions (*Do*) and *Check* whether the actions effectively and efficiently improved the situation. If the situation improved, we try to improve even more. If not, we change the strategy again. We always *Act* as necessary.

By introducing *mutations* in the Act-phase of PDCA rapidly and frequently, keeping what works better and shelving what works less, we force rapid evolution. Hence we call our approach Evolutionary.

If we appropriately organize projects in very short Plan-Do-Check-Act (PDCA) cycles, constantly selecting only the most important things to work on, we will most quickly learn what the real requirements are and how we can most effectively and efficiently realize these requirements. We spot problems quicker, allowing us more time to do something about them.

Is it then only positive? No negative effects to consider? That's actually the power of Evo: Evo itself provides the very mechanism to cope with any negative issues or risks: using PDCA, we recognize negative things and *do* something about them. The only remaining negative things are those things we don't consider important enough to do something about for the moment.

Some people fear that all this tuning will take a lot of extra time. Evo projects, however, prove to be significantly faster than other projects. In only a few weeks time we see a 30% productivity improvement when projects start working the Evo way (based on experience of the author after coaching more than 60 projects in the past 6 years). In Evo, we never do things if they take more time than necessary.

Evo is not only both iterative (using multiple cycles) and incremental (we break the work into small parts), but above all Evo is about quickly *learning* how to do things better, using PDCA. We systematically and proactively anticipate risks before they

occur and work to prevent them. We may not be able to prevent all the problems, but if we prevent most of them, we have a lot more time to cope with the few problems that slip through, before they materialize as a Stakeholder problem.

Using PDCA we are constantly optimizing:

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

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

Evo elements

All elements of Evo address typical project risks, such as:

- Schedule risk: delivering at the wrong time
- Not living up to our promises because of unrealistic optimism
- Promising more than we can do
- Making more mistakes than necessary because of fatigue
- Delivering the wrong things and delivering at the wrong time, not making the customer happy and more successful than before
- Others causing us to fail
- Gold Plating (doing more than necessary)
- Handling interrupts: Losing time on *seemingly* important things
- Coping with suppliers beyond our control
- Doing the wrong things for too long
- Trying to do more than we (as humans) can handle in a certain amount of time

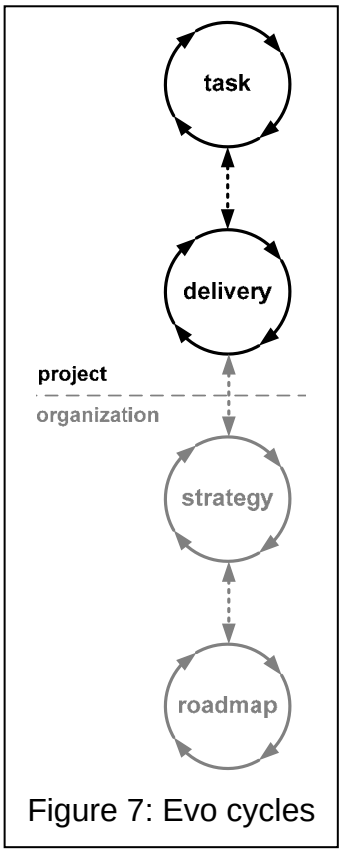


Figure 7: Evo cycles

In Evo, we use several learning cycles (

Figure 7):

- The weekly **TaskCycle** is used for organizing the work, optimizing estimation, planning and tracking. Every week we check what the most important tasks are, estimate the effort needed to complete these tasks completely and commit to the most important tasks we can complete during the week. We plan 2/3 of the available time as *plannable* time, leaving 1/3 as *unplannable* time for all the small interrupts that will occur anyway (email, phone calls, helping each other, …), allowing people to succeed finishing what they committed to. We use TimeBoxing helping people to focus on what is really necessary: doing less without doing too little. We constantly check whether we are doing the right things in the right order to the right level of detail. We optimize the work effectiveness and efficiency. We found that people in projects can very quickly learn to change from optimistic estimators into *realistic estimators*, *if* we are serious about time. Once we master realistic estimation, we can better predict the future and we can deliver on time as agreed. We learn to promise what we *can* do, and then to *live up to our promises*. Estimating in TimeBoxes also relieves people from the need for tracking. All these details of the

TaskCycle are designed to control *schedule risk*.

- The **DeliveryCycle**. Every two weeks or less, we deliver useful value to stakeholders: juicy bits to enthuse them to provide feedback, intermediate results that make them already successful now, or at least we deliver something that will provide the most important feedback momentarily possible. The DeliveryCycle is used for optimizing the requirements and checking the assumptions. We constantly check whether we are moving to the right product results. DeliveryCycles focus the work organized in TaskCycles. This way we control the *risk of not delivering the right results*. An important question we ask ourselves when defining a Delivery: "*What* do we have to deliver in two weeks, to *Whom* and *Why?*"
- In larger projects we see many more cycles developing, for example: parallel DeliveryCycles, shorter DeliveryCycles, and additional longer DeliveryCycles, like Releases or Intermediate Deliveries.

At the organizational or enterprise level, we use:

- The **Strategic Objectives Cycle**. In this cycle we review the strategic objectives at the organizational/enterprise level and check whether what we do still complies with the objectives. This cycle may take 1 to 3 months.
- The **Enterprise 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.

Many people think that the cycle periods we use in Evo are too short. Some people even think that such short cycles are impossible. This is one of the more difficult things to learn when changing to the Evo way: It always can be done. It requires a paradigm shift that is not difficult, but doesn't come naturally. So it has to be taught.

One reason for applying such "short" cycles is combating the risk of *doing the wrong things for too long*. The later we find out that we are doing the wrong things, the longer the wrong things have endured and the more we have to repair or redo. This way, short cycles do not cause more work (doing the Check/Act more often), but rather cause less work (having to repair less). Another reason for applying short cycles is that people simply cannot really oversee and organize longer periods of time in detail. For controlling longer periods of time, we have a separate technique, called *TimeLine*.

TimeLine (Figure 8) is used to keep control over the total project, making sure that at the FatalDate or at the FatalBudget, we will have delivered the best possible value. We optimize the order of Deliveries in such a way that we approach the product result along the shortest path, with as little rework as possible. TimeLine is also used to dynamically adjust the order of deliveries to the available resources. We treat FatalDates seriously and count back when we should have started to achieve what is required. Because the customer usually wants more than he can afford, it is important to

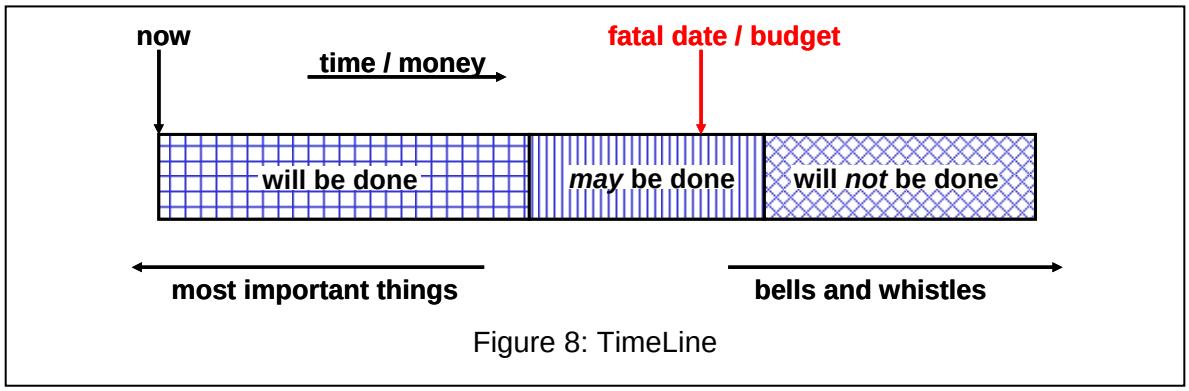


Figure 8: TimeLine

know what we, at the FatalDate, surely will have done, surely not have done and what we may have done (after all, estimation is not an exact science). Better than to tell the customer at the FatalDate that we didn't succeed in the impossible, we rather tell him as soon as we possibly know. Then we can together decide what to do with this knowledge. Every day we know a potential problem earlier, we have a day more to do something about it. With TimeLine, we control the *risk of being late*, delivering optimum value in the limited available time. During the project, we continuously keep using TimeLine to predict the outcome of the project based on the ongoing development of knowledge. In Evo we don't use the concept of *Earned Value*. After all, how can we talk about Earned Value if the Requirements are evolving…? In many conventional projects we even see that Earned Value degrades to *Money Spent*. In Evo we rather use *Value Still to Earn*.

Requirements in Evo

Although the Requirements should be stable for best results, they aren't. During a project, the developers learn, the customer and the other stakeholders learn, and the market changes. So, because requirements change is a *known risk*, we *provoke* requirements change as quickly as possible, preferably before they are implemented. Because the customer and other stakeholders usually cannot very well express what they really need, we use several techniques to find out what they do need, first developing the problem, before we start developing a solution.

Every project has many Stakeholders. A Stakeholder is anyone having a stake in the requirements: customer, users, and many others, including the developers. If at the end of the project we realize that we forgot an important stakeholder, we are in trouble. If we find a requirement without a stakeholder, either it isn't a requirement, or we haven't identified the stakeholder yet. If we don't know the stakeholder, who will pay for our work, how would we know we are implementing the right things, and how would we know when we are ready?

Requirements are divided in:

- **Functional Requirements**, scoping the project. The Functional Requirements describe what we will improve in this project. We choose this particular set of functions to improve, because a different set will yield less benefit.
- **Performance Requirements** defining how much the functionality will be improved. Note that all the functions are already there. With our new product, people should, for example, be able to do what they did before more quickly, making them more productive. The Performance Requirements are the most important requirements and have the most impact on project time and cost. Therefore it is imperative to pay adequate attention to these requirements.
- **Constraints** defining what we are *not* allowed to do, e.g. for legal, environmental and moral reasons, or what we are simply not supposed to do.

Performance Requirements shall always be numerically defined, otherwise we will not be able to determine whether we have achieved the required performance. Performance Requirements are an important driver for choosing the appropriate architecture and if they are not stated early, the chosen architecture will usually prohibit achieving them later. If the airport is opened and the PA-System.Audibility turns out to be bad, it may cost a lot of acoustic redesign once we define the Audibility-requirement e.g. as "96 of 100 people waiting for departure can reproduce the message". If a Performance Requirement cannot be defined numerically (is Maintainability 3 or 7?)

we call it a complex concept, which has to be decomposed into components which can be numerically defined, like e.g. “Maintainability.MTTR < 15 minutes” and “Maintainability. DownTimeDuringUpdate < 1 sec”.

Specifying requirements

For specifying performance requirements, we use Planguage, as proposed by (Gilb 2005). In the example in Figure 9, we show some basic elements of this method. To be able to specify numerical values of the performance requirement, we need a defined *Scale*. The *Meter* defines how we measure the values on the scale.

RQ27: Maximum Response Time
Scale: Seconds between <asking> for information and <appearance> of it.
Meter: Add a function to the software to measure the maximum response time value and the range of values per <working day>.

Benchmarks:

Past: 3 sec (our previous product)
Current: 0.6 sec [competitor y, product x, 2006] ← Marketing Survey, Jan 2006
Record: 0.2 sec [competitor x, product y]
Wish: 0.2 sec [2008] ← customer's head of R&D, 19 Feb 2005, <document ...>
Note: Less than 0.2 sec is not noticed by the user, so there is no use in trying to be better than 0.2 sec

Requirements:

Fail: 1 sec [99%] ← project-contract
Fail: 1.5 sec [100%] ← project-contract
Goal: 0.5 sec ← project-contract

Figure 9: Using Planguage to specify Performance Requirements

Benchmarks define the playing field. Some examples are:

- **Past**: The value in our previous product. We won’t have improved if our new product has the same specification.
- **Current**: The current state of the art. If we don’t achieve this level, we won’t beat the competition, which would constitute a risk.
- **Record**: This is a level that will be hard to beat (like an Olympic record). It probably will cost a lot to achieve this. Could very well be much more than our customer can or is willing to afford.
- **Wish**: This is a possible future requirement. We are not planning to achieve this level now. It may be too costly to achieve with the current state of the art, but it is what we actually would like, once feasible. The customer is not prepared to pay for this in the current project.

Then we describe the Requirements, with at least a Fail and a Goal value:

- **Fail**: If we do not achieve this level, the project fails.
- **Goal**: This is the level we expect to achieve in this project. When we have achieved it, we are done.

The power of generating requirements in this fashion is that it stimulates greatly the communication and understanding of the requirements and we often see that perceived initial requirements quickly change into other, more appropriate requirements, reducing the risk that we start working on implementing the wrong requirements. During development the architecture and design tries to cover the set of conflicting (for example: performance and budget are always in conflict) requirements as best as possible. Having a range between Fail and Goal, leaves room for intelligent compromises.

Engineers are trained to achieve planned results *by design* (Figure 10). Sometimes, however, we reach a goal by improving different parts of the system, one step at the time. Many developers are used to trying to accomplish as much as possible in one step. In Evo, we always select the smallest step possible. If this step later turns out not

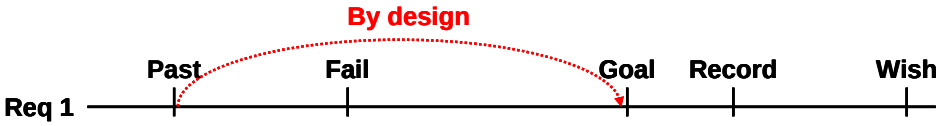


Figure 10: Engineering is achieving a goal *by design*

to be the right step, we have to redo as little as possible. And the step that takes the least time leaves us the most time for whatever we still have to do.

Figure 11 shows how we are safe after one delivery step (better than Fail: at least we don't fail). After two more deliveries we reach the Goal value, indicating that we have

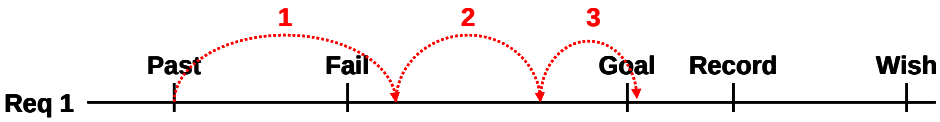


Figure 11: Sometimes we reach the goal by improving different parts of the system, *one step at the time*

achieved our goal and that we don't gain anything if we continue. Hence we stop. This way, we mitigate the *risk of Gold Plating*, which is doing more than necessary.

In real projects, we have to cope with many conflicting requirements at the same time. In one Evolutionary delivery step, we work on Requirement 1 (Figure 12: step 1), getting past the Fail level. Therefore, in the next DeliveryCycle, we better first work to get another requirement beyond the Fail level (step 2). In some cases, an improvement of one performance may adversely affect another performance, which we may

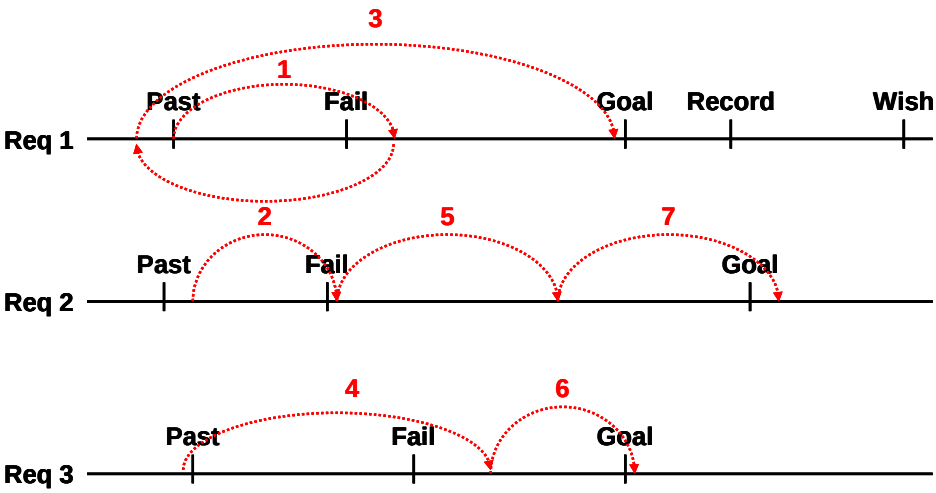


Figure 12: In practice, we have to satisfy many conflicting requirements simultaneously

improve in the next step (step 3). In similar fashion we deliver step by step until all requirements are at the Goal level, or until the budget in time or money is depleted.

In Evo we never overrun the budgets. As soon as we are beyond the "safe" Fail levels for all requirements, we can basically stop at any time, for instance if the customer decides that time-to-market is more important than further improvements. The Evo Requirements Engineering Approach addresses the *risk of delivering the wrong things and delivering at the wrong time*.

Active Synchronization

If we are working the Evo way, somewhere around us is the bad world, where people are not yet accustomed to living up to their promises. Software people may need hardware to test their software. Hardware people may need test software to test their hardware. Other disciplines may have to deliver to us, or need our results. You may be collaborating with people at other places of the world.

If you are waiting for a result outside your span of control, there are three possible cases:

1. You are sure they'll deliver Quality On Time (the right results at the time agreed)
2. You are not sure
3. You are sure they'll not deliver Quality On Time (because they never did)

Note:

- Your Evo project behaves like case 1
- From other Evo projects you can expect case 1
- If you are not sure (case 2), better assume case 3

In cases 2 and 3: Don't wait until you get stuck not receiving the agreed result on time. You know you won't if you don't Actively Synchronize, so: Do something! Go there! This has three advantages:

1. Showing up increases your priority
2. You can resolve issues which otherwise would delay delivery
3. If they are really late, you'll know much earlier

With Active Synchronization, we control the *risk of others causing us to fail*.

Interrupts

One of the potential risks of loosing time is interrupts. In Evo, we only work on planned tasks, never on undefined tasks. In case a new task (or a new requirement) suddenly appears in the middle of a Task Cycle, we call this an Interrupt.

Assume the boss comes in and asks us to paint the fence. We don't say Yes, but we also don't say No. After all, painting the fence may be more important than anything we have currently planned, if an important customer would turn his heels when he sees the shabby fence. Instead, we follow the Interrupt Procedure:

- Define the expected Results of the new Task properly. What is the actual risk that the shabby fence causes trouble? Should we thoroughly grind, ground and paint the fence, or buy a new fence that doesn't need paint, or just put a quick layer on the old fence, just covering the dirt?
- Estimate the time needed to perform the new Task, to the level of detail really needed.
- Go to the task planning tool (many Evo projects use the Evo Task Administrator tool (ETA 2004)).
- Decide which of the planned Tasks have to be sacrificed (up to the number of hours needed for the new Task).
- Weigh the priorities of the new Task against the Task(s) to be sacrificed.
- Decide which is more important.
- If the new Task is more important: replan accordingly.
- If the new Task is not more important, then do not replan and do not work on the

new Task. Of course the new Task may be added to the Candidate Task List, to be considered later.
- Now we are still working on planned Tasks.

But couldn't this be delaying the work we originally planned? Yes, of course it could. But we deal with the consequences of the change in the plan. We *Act*. We don't *let* things happen randomly, we rather *control* how they happen. If some requirements become more important than others, the order of what we do *should* change. Priorities do change all the time, so the thing is to dynamically reprioritize as needed. Revisiting TimeLine will tell us what the consequences will be for what will be done, what will not be done, and what may be done at the FatalDate.

We simply cannot do more than we can do. We don't try to do the impossible. All we can do is making sure that looking back we always can say we did the best possible job. With the Interrupt procedure, we control the risk of *losing time on **seemingly** important things*.

Boehm's 10 top software risk items

Barry Boehm described 10 software risk items (Boehm 1991), which probably still are considered risks today, also for non-software projects. Let's check whether and how we address these risk items the Evo way:

- **Personnel Shortfalls**. We have a certain number of people available in our organization. At the organizational level (Figure 7, in gray), we compare the priorities of all the work we could do with the available resources. If a certain project does not get the appropriate number of people needed for a certain development load, this will be only because other projects create even more value than this project. With TimeLine the project determines what it can do with the available resources and if this is less than needed, they inform management about the consequences. This information is input to the organizational prioritization process. This isn't a risk, it's a choice.
- **Unrealistic schedules and budgets**. If the requirements aren't clear (which is usually the case; be honest!), any schedule will do. If the requirements change anyway, how can we talk about realistic or unrealistic schedules and budgets? We take time and budgets as a given and fill them delivering the best possible value. If, within the available time and cost we can't deliver sufficient value, we won't even start. We constantly update the TimeLine in order to predict what at a certain date will be done, won't be done and may be done, and we take the consequences. It is our experience that we can quickly change People in projects from optimistic estimators into realistic estimators, making them able to live up to their promises. This way, they have *facts* to explain management about the realisticness of schedules. In such an environment, managers who keep asking for unrealistic schedules shouldn't survive. Or, if managers insist in unrealistic schedules (*Check*), either they should be educated (*Act*), or they *want* the project to fail. If they want the project to fail, they should better tell us, because then we can do that much more efficiently.
- **Developing the wrong functions and properties**. The Evo requirements process deals with this issue. Frequent stakeholder feedback is used to optimize the requirements and check the assumptions, making sure were are developing the right things.

- **Developing the wrong user interface**. Same as previous. Because Evo requirements are stated in terms of stakeholder success, which may include improvement of user productivity, the developers will make sure that the user interface supports those requirements. We check our assumptions about the user interface with early deliveries and check whether we are achieving the real requirements or not. If necessary, we *do* something about it.
- **Gold-plating** is suppressed because we deliver as per requirements, specified by Planguage. When we achieve goal values, we are done. Normally, people tend to do more than necessary, especially if it's not clear what should be done. Making it more clear is a big time-saver.
- **Continuing stream of requirements changes**. Requirements do change because *we* learn, *they* learn and the market changes. If we would deliver according to obsoleted requirements, we won't secure customer success, forsaking the Goal of the project. So, we expect requirements changes and have a process to deal with them. We even *provoke* requirements change as soon as possible, preferably before we implemented any requirements that had to be changed. If we are not sure about a requirement, we do as little as needed, just enough to find out what the real requirement is, in order to redo as little as possible in case our assumptions were wrong. TimeLine is used to guard what we will and will not deliver at the FatalDate. We *Act*, if *Check* indicates a problem.
- **Shortfalls in externally furnished components.** We use Active Synchronization to stay on top of this issue. We know that when our FatalDate has come and we don't deliver, there is no point in finger pointing: we simply failed! Well before our FatalDate we would have got deliveries from the external suppliers, knowing early about any problems. Any day we know a problem earlier, we have a day more to do something about it.
- **Shortfalls in externally performed tasks**. Same as previous. We request regular Evo deliveries from our external suppliers, so any potential problems will show up quickly. Note that the possibility of shortfalls influences the order of deliveries: highest risks first. If a risk turns out for the worst at the end of the project, we are in trouble. We don't want to get into trouble, so we *design* the order of whatever we do in our project to minimize the chance for trouble and optimize the use of our time.
- **Real-time performance shortfalls**. Come on. That's simply a Performance Requirement and then an engineering issue. Are we amateurs?
- **Straining computer-science capabilities**. In Evo, we plan to do the right things, in the right order, to the right level of detail. We don't start with the easy things, but rather with those things we are not yet sure of. And if we find out that the necessary requirements cannot be met within an acceptable budget of time and cost we report this to management and the customer and discuss what we do with this knowledge.

Concluding: all of these so called risks are not really risks. There are adequate processes to cope with these issues in a responsible way. Again: we aren't amateurs, are we?

The biggest risk

The biggest risk is the risk that we are overlooking something:

- It's within our span of control, but we don't recognize it.
- It's not within our span of control, but we don't anticipate, or we don't do enough to avoid the problem to occur (we should have Actively Synchronized to avoid this, but

missed it).

The trick is to be ahead of problems, before they occur. We don't ostrich; we actively take our head out of the sand. If somebody complains, we're too late. If the FatalDay is there, excuses and finger pointing are irrelevant. If we don't deliver, we fail. We don't want to fail, so we do whatever (ethical) we can to avoid failure. Why? Because we want to achieve the best possible results in the shortest possible time. Why? Simply because that's what we like.

Conclusion

A lot of so-called risks in projects aren't really risks: we may not know when exactly, but they almost always do happen. If we devise good, practical techniques and processes to cope with these "risks", we can minimize the impact, so that they don't jeopardize the successful result of our project. Evo provides these techniques and processes, continuously acting proactively and making sure that we are delivering the right things at the right time. If we have mitigated *by design* most of the risks that usually plague projects, then we have much more time left to handle the unexpected risks we still have to deal with. Evo provides the technique and the attitude to deal with the unexpected risks as well, applying PDCA rigorously and frequently.

We design not only what we agree to deliver (the product), we also *design* the way we work to succeed in our goal (the project). Evo seeks to optimize this process, by constant learning to doing things better. In this process, risks are not handled separately, but as an integral part of running a project. Evo is full of small details designed to ensure success, some examples of which are described in this paper. The process itself being evolutionary as well, Evo constantly optimizes its own ways. Some people fear that all this analyzing, learning and optimizing may take a lot of extra time. In practice we see that it saves time: A project adopting these methods for the first time, generally completes successfully in 30% shorter time. Nothing of the above is merely theoretical. It has been tested, honed and proved by the author in the practice of more than 60 projects in various environments and cultures.

References and further reading

Boehm, Barry W., “Software Risk Management: Principles and Practices”, IEEE Software, vol. 08, no. 1, pp. 32-41, Jan/Feb, 1991.

Deming, W.E.: *Out of the Crisis*. MIT, 1986.
also: Walton, M: *Deming Management At Work*. The Berkley Publishing Group, 1990.

ETA: *Evo Task Administrator*, 2004. Tool for administering Tasks in Evo projects. http://www.malotaux.nl/nrm/Evo/ETAF.htm

Gilb, Tom: *Competitive Engineering* Elsevier, 2005.

Incose, *Systems Engineering Handbook*, Version 3, June 2006, Figure 7-6.

Larman, Craig, Vic Basili: “Iterative and Incremental Development: A Brief History”, *IEEE Computer*, June, 2003.

Malotaux, Niels: *Evolutionary Project Management Methods*, 2001. Issues and first experience, introducing Evo at a large company. http://www.malotaux.nl/nrm/pdf/MxEvo.pdf

Malotaux, Niels: *How Quality is Assured by Evolutionary Methods*, 2004. Practical details how to implement Evo, based on experience in some 25 projects in 9 organizations. http://www.malotaux.nl/nrm/pdf/Booklet2.pdf

Malotaux, Niels: *Optimizing the Contribution of Testing to Project Success*, 2005. How to apply the Evo ideas on the testing process to iterate more quickly towards Zero Defects. http://www.malotaux.nl/nrm/pdf/TestingEvo.pdf

Malotaux, Niels: *Optimizing Quality Assurance for Better Results*, 2005. Similar to the previous, but targeted at non-software QA. http://www.malotaux.nl/nrm/pdf/Booklet3QAV01.pdf

Rafele, Carlo, David Hillson, Sabrina Grimaldi: “Understanding Project Risk Exposure Using the Two-Dimensional Risk Breakdown Matrix”. *PMI Global Congress Proceedings* - Edinburgh, Scotland, 2005.

Biography

Niels Malotaux is, since 1998, an independent consultant and project coach, teaching immediately applicable methods for delivering Quality On Time to projects and organizations. He has over 30 years experience in designing electronic and software systems, at Delft University of Technology, in the Dutch Army, at Philips Electronics, and 20 years leading his own systems design company. Niels researches how human behavior affects the results of what people do. He developed both techniques to optimize the results of what people do in projects and how to teach people this successful approach very effectively and efficiently. Since 2001 he taught and coached over 60 projects in some 18 different organizations in the Netherlands, Belgium, Ireland, India, Japan and the USA, creating a wealth of experience about what really works in practice. He is a frequent speaker at conferences and published four booklets on the subject. Niels can be contacted at niels@malotaux.nl. More can be found at www.malotaux.nl/nrm/English.