Chapter 5

Managing Risks

> "If you do not actively attack the risks, they will attack you." Tom Gilb (1988)

As a project manager, you want to succeed, right? And by "succeed" you really mean: "not fail". In this chapter we will examine the concepts of *success* and *failure*, and the nasty "things" that lies between you, now, and success or failure, in a short while. These are called *risks* and *uncertainties*. Some simple practices can be used to keep a good handle over the risks that threaten a project. We will then introduce one fundamental software development practice which will help us to avoid failure, and curb down our exposure to risks: *iterating*.

Outline

- Success & failure
- Uncertainty and risk
- Risk exposure
- Steps and tactics for managing risks
- Sources of risks in software projects
- Iteration as a risk mitigation strategy
- Standards

Success

While it may be easy to understand what failure looks like, it may not always very easy to define what success looks like. Success could be defined simply as the opposite of failure, something like "success is meeting the entire set of all requirements and constraints held as project expectations by those in power" (RUP 2003) .

Note however that there maybe more stakeholders defining success than just the customer. And there is often more to success than just achieving one delivery: development needs to be sustainable, from various viewpoints: financially sustainable, but also from a team perspective, and a design and code quality perspective.

In terms of our conceptual model (see chapter 2), success is achieved when *Product* matches *Intent*, at a point in time close to that intended, and with a cost (mostly derived from the *Work* performed by the *People*) close to what was expected. Suzanne and

James Robertson (2004) identified a range of success indicators. We can organize them along the lines of the conceptual model. I marked with a star (*) the ones that may not be seen as important by your stakeholders at a first delivery, but which are important for sustainability, that is, getting your project or software product beyond the most immediate milestone:

Time:
•On time delivery.
•No excessive schedule pressure*

Product
• *Completion* (rather than cancellation)
• *User and/or customer satisfaction.*
• No nasty surprises
• No litigation

Work
• *On budget delivery*
since as we will see later one, most of the cost are associated with work
• Adequate productivity*
• Adequate process used*
• Good quality code*
• Control of configuration*
• Adequate documentation*

Intent
• Lack of creeping requirements
Less than 5% changes after they were deemed “complete”
• Correctly sized product
• No misunderstood requirements

People
• Good team morale*
If you reached your first important milestone, but nobody want to do this ever again, and they are rushing for the exits, maybe in the long range this will not be seen as a great success.

So from a project manager perspective, we can say that yes, success is meeting most requirements, demands and constraints from the important stakeholders, and then on top of this, having a team who would be happy to do another project like this one with you at the helm, and having set the conditions for being offered to do another one.

Quite often, especially in large projects, the various team members do not have a good, common picture of what success is. Often they are blinded by their own, more limited horizon, or the end date is way too far remote, or (much worse), they have their own private agenda.

A first important question for a project manager is:

> How do we define success? Do we have a shared, common understanding of what success is?

Failure is nonsuccess. We could therefore define it as "*not* meeting the expectations of those in power," your key stakeholders. But here also there are degrees of failures; it is not black and white. You can be a bit late, and slightly over budget, or have an overrun of 200%, and a project lasting 3 times longer than originally thought. Or you could deliver on time, but find yourself with no one to continue with you; or a project that is so badly built that a straw would break it.

Uncertainty and Risk

If you had a crystal ball, you would know everything that is going to happen to your project, between now and the end point. Software projects however are never like this. There are many, many things we do not know in advance. There is much *uncertainty*, and the most frequent answer to all your questions will be a frustrating "I don't know."

Uncertainty is the lack of certainty (duh!). It means that it is a state that may have many different outcomes, many different possible futures, many different possible responses, answers, or values. We can measure uncertainty by assigning a numerical value to each possible outcome: its probability of occurrence, the likelihood that it will happen. "There is a 60% chance that the number of users will be more than 1,000, and a 10% chance that we'll have more than 10,000, and about 0% chance that we'll have more than 1 billion."

Risks are a subset of these uncertain outcomes: they are the uncertain outcomes that will hurt us in some way. "There is a risk that we will have less than 100 customers." So, a risk is a state of *uncertainty* where some of the possibilities involve an undesirable outcome.

More pragmatically: A risk is whatever may stand in our way to *success*, and is currently unknown or uncertain.

RUP defines a risk as an ongoing or impending concern that has a significant probability of adversely affecting the success of major *milestones* (RUP 2003).

And the IEEE definition is not too far either: A risk is the likelihood of an event, hazard, threat, or situation occurring and its undesirable consequences; a potential problem (IEEE 1540).

Examples of such "unsatisfactory outcomes":

- Budget and cost overruns
- Product with the wrong functionality or incomplete functionality
- User-interface shortfall

- Performance shortfalls
- lack of reliability
- Software impossible to evolve or maintain

Looks bad. These seems to be mostly technical, but actually many shortcomings can be traced back to bad project management.

Like for an uncertainty, we can attribute some *probability* or likelihood to that adverse, negative outcome. “There is 5% chance that we will have less than 100 customers.” But this is not enough: we also need to distinguish among all the potential risks, the ones that can really affect us big time. Hence we introduce another concept: exposure.

Risk Exposure

For a risk, not only would we ask ourselves: “what’s the likelihood?”, but also “How much damage would this do to the project if it were to become true, become a problem?”

Impact, or damage

Traditionally in project management, people have estimated the damage done using financial measure: “how much money do we lose if this happens?”

For example, if we are late by a week, the cost is $57,000.00 in salaries, plus some lost sales, say $130,000, for a total or $187,000.00. That is the damage to the project; or the loss, or the impact.

If being late by a week has a very small chance of happening, we would say: well, this is not a big risk, despite the big loss amount.

That’s why we introduce the concept of *exposure*, as measure of how bad a risk really is. Exposure combine the probability with impact, by a simple multiplication.

for a risk R: Risk exposure (R) = Probability (R) x Impact (R)

So if the likelihood of being a week late is only 5%, the exposure is only $9,350. But if there is a fifty-fifty chance of being a week late, the exposure is: $93,500. Big difference.

The exposure is here also in monetary terms. We can add up the exposure for the various risks to determine the complete project exposure.

Risk	Probability	Impact	Exposure
Late by a week	10%	$187,000	$18,700
Show-stopper bug in User-Interface	30%	$200,000	$60,000
John leaves	2%	$500,000	$10,000
... etc.	...	...	....
		Total exposure:	$88,700

And the traditional line of reasoning for project management would be roughly to make sure that your likelihood of profit exceeds your total exposure, otherwise, it is not a project you should embark on. This is the “risk neutral” approach.

I am sure you can smell some nice maths and statistics lurking beneath this. Yes, we can for example refine this technique to giving a probability *distribution* for your risk exposure, instead of a single number:

Fig here.

And these things for =m the basis of quantitative risk analysis and the statistical decision theory. And we will *not* go there. Not now.

Setting up and computing a table like the one above looks pretty easy to do at first sight, but there are 3 fundamental problems to make this really practical for a software project.

1) To compute the *complete* project exposure, we would need to be able to identify *all* potential risks, and for each one of them estimate a probability and damage. We’ll so many different things can go wrong, how would we be able to enumerate them all?

2) How can we allocate a priority to each risks. There are things we know they are unknown and uncertain, but can we put a number on their probability? We can try but it may lead to hours of rather useless arguing.

3) Finally, trying to define damage in monetary terms is also very difficult in practice. While the cost of a week delay seems straightforward, how to estimate the cost of a missed requirements, or a team member leaving, in dollars or euros?

Solving these issues would distract us from our main objective: succeed with the project. So we need to devise some simpler solution to help us handle risks, especially allowing us to focus on the most daring ones.

Simpler risk exposure
A much simpler and expeditious method is to just work with some simple 'buckets' along our 2 risk attributes: probability and impact.

For impact, let us define a scale of:

1 - Minor annoyance
2 - Small delay or cost overrun
3 - Minor quality issue
4 - major delay, cost overrun, quality issue
5 - Catastrophic, project sudden and certain death

and similarly for probability:

1 - Once in a lifetime (<0.1% chance)
2 - Very unlikely (<1% chance)
3 - It may happen (< 10% chance)
4 - It happens quite often (20% to 50% chance)
5 - It happens very often (> 50% chance)

So, the risk "we are hit by an earthquake", has an impact of 5, and a probability 1. Risk of "john leaving the project" has an impact 4, and a probability 3, right now.

From here you will define an exposure using a table like this one:

5	Minor	Serious	Major	Major	Extreme
4	Minor	Minor	Serious	Major	Major
3	Minor	Minor	Serious	Serious	Major
2	Minor	Minor	Minor	Serious	Serious
1	Ignore	Minor	Minor	Minor	Minor
	1	**2**	**3**	**4**	**5**
Probability	**Impact**				

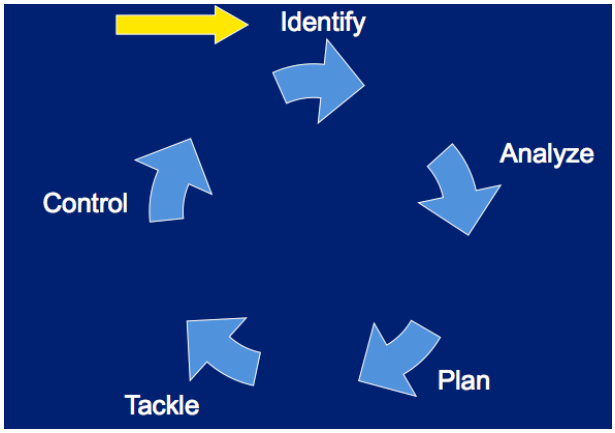
Or, again, simply multiplying probability and impact, ending up in a scale of 1 to 25.

You could have a more complex range of 1 to 10, both either or impact and probability, but in practice it brings some difficulty to define the various levels, and does not bring much benefit. Or you can simplify and have a 3 x 3 grid with impact: low, medium, high, and probability: low, medium, high.

Basic risk management

Now that we have introduced the key concepts, what should the project manager do? Basic risk management is a repetitive, cyclical process with a few simple steps:

- Risk Identification: what are the risks?
- Risk Analysis: how serious or threatening are they?
- Risk Planning: what are we going to do about them?
- Risk Tackling: do something about the risk
- Risk Controlling: Monitoring: are they turning into real problems? Resolution: did we get rid of them, or reduced them to small “nits”?



Risk Identification

The first step is to enumerate the risks: what can go wrong in the project, what can threaten it, have a negative outcome.

To do this we introduce our first concrete artifact: the risk list. The risk lsit is best handled as a spreadsheet, looking like this:

Who identifies the risks? Where do they come from? As the project manager, you start. If you have some experience of previous projects, think about what went wrong in your past experience, and decide wether your new project is likely to run into the same issues or problems. You can also looks at known, published lsits of software risks, and see if some of them apply to you. See section 5-? for examples. Such risks are usually expressed in rather high level, abstract terms; make them more concrete:

Staff turnover => We lose of product manager
User interface quality => Dialog for bank teller is not intuitive
=> Lack of acceptance by bank staff
[more?]

Then, ask your immediate environment. Have a little brainstorming session with your team to enumerate what can go wrong. Speak to your outside stakeholders: product managers, support people, etc.

Express all the risk uniformly as conditions that endanger the project, with a phrase that could be prefixed by “The project is in danger if or when” <risk short description>.

When eliciting risks from your colleagues, do not try to resolve the issue, just state the potential issue. Do not assign now responsibility for any mitigation action. Just identify the risks.

Risk Analysis
The second step is to determine which are the important risks. If you were facing with only a handful, there would not be a problem, but if you are doing a good job, you will rapidly realize that you list of risks has dozens, or even hundreds of risks.

So you need now to determine exposure. Assign a probability (category) and an impact (category) to each risk, and compute (by a simple multiplication) the exposure. Then rank them in decreasing order of exposure. Your most daring risks, the killer ones are now floating on the top of your risk list.

You might want to do some pair comparison going down, and apply a bit of simple common sense to decide if they are not inverted, and go back and revisit the impact or the probability.

“Knowing the enemy,” wrote Karl Wiegers (1998); it is a good starting point, and make it known to the rest of your team. Barry Boehm suggested to publish regularly the top ten risks, not only to your own team, but also to some of the immediate environment: management, customer representative.], etc. The risk you are facing are not a dirty little secret that should,prevent the project manager to sleep at night. They are everybody’s business.

Risk Planning
Awareness is only the beginning. What are you doing about risk? There are several tactics. Ideally you would like to eliminate them completely, but when this is not possible, you would like at least to reduce your exposure. This can be done by either reducing the likelihood, or reducing the impact.

Risk avoidance:
Can we change our strategy, design, requirements, etc., in order to not have to face that risk? If for instance there is a risk associated with the migration to a new database, sticking with the old database would avoid the risk (at the expense of something else probably, but there is no free lunch). The question becomes often “can we not do this <risky thing>?”

Risk mitigation:
If we cannot eliminate the risk, can we at least reduce the impact or the probability of occurrence. Very often this involves doing some more investigation, some experimentation, some prototyping to try to discover more, to reduce the uncertainty. In some cases you will discover what the impact really is, or change your mind about the probability. In some cases, the risk will turn to become a problem.

Risk Contingency:

Colloquially, we often call this “having a plan B”. If the risk were to become a problem, do we have another solution, another approach.

Risk transfer:
Finally, could we push the risk to another party, so that it is not our risk anymore but someone else’s risk? This game is played a lot, but it is in general not a very useful strategy, unless you transfer the risk to another party, such as a subcontractor or consultant, who has much more expertise than your own team, and therefore for whom it would not be a risk. If you transfer the risk to some external party (like an outsourcer), with equal or less expertise, you may not have a complete win in the end. If the risk becomes a problem, you just have found someone to blame, but the problem will remain and be yours anyhow. So risk transfer is a possibility, but only if it actually reduces exposure.

Practically, go down your list of risk, in decreasing order of exposure, and assign on strategy, possibly with someone responsible for carrying it out, by a certain date. You can capture this in the same spreadsheet.

Direct and indirect risks:
While you are doing this, you will discover some risks for which you are a bit stumped: there seem to be nothing your team can do to mitigate them in anyway.
“New competitor on the market.” In most cases it is not possible to avoid, transfer.
We call them indirect risks. Direct risks are the ones you can actually do something about, such as bad “workload estimates” or choices of new technologies; you can experiment, learn, improve, use new techniques.
So there is another non-tactic: Risk Acceptance: just live with it. For indirect risks, we can only have mitigation strategies: reduce the the impact. For direct risks we can have more deliberate strategies, make then non risk, that is, completely resolve the uncertainty or reduce the probability of occurrence to nothing.

Risk Monitoring
Now you have a sorted list of risks, and for most of them some resolution tactic. Some have a contingency plan. Do not sleep on this. List also some indicators, some signs that would be a hint that the risk is turning into a problem. And revisit the list of risks regularly, once a week at least. The top ten direct risks, at minimum. As the development progresses, things that were unknown start to be known, problems get resolved, and in general we know more.

Risk Resolution
That’s the happy part of the cycle, were we can cross out risks from our list.

And do it again, and again
This is not a process you through once at the beginning of a project, and than tick if off: “risk management: done”. In reality at the beginning of the project you will not be able to identify many risks, besides the very generic ones, that all software projects have. It is only when you start doing the work, concretely as a team that new ideas of risks will

emerge, difficulty will point on the horizon. The daily stand-up meeting (see chapter ?) is a good source of risk identification. The end-of-iteration retrospective is also a source of risk identification (see chapter ?). In the first 1/3rd of the project it is even a bit depressing: your list of risk seem to grow forever, none are being resolved.

Sources of software project risks

How do we get started with risks? What are the main classes of risk and more generally uncertainty in software projects? Our conceptual model has risks and uncertainty associated with each of the 4 key concepts.

Intent

- Lack of clear vision, no definition of success
- Lack of agreement on requirements
- Unrealistic customer expectations
- Vague requirements
- Rapidly changing requirements, requirements creep (see chap. 9)
- Quality attributes ill-defined, not quantified (see chap. 8)

Work

- No clear process, too much improvisation
- No good support for automating tedious, repetitive tasks
- Inadequate planning
- No visibility on the project status
- Excessive pressure schedule

People

- Not enough people
- Not the right people in terms of skills, competence
- Dependencies on the work of other groups
- Communication, especially if team not collocated
- Interpersonal conflicts
- Turnover
- Cultural misunderstandings

Product

- Dependencies on other technologies
- Low quality
- Ineffective testing

As we progress in this course, we will dig further into all these sources of risks. Other sources of “risk ideas” can be found in the SEI’s *Taxonomy of risk* (Carr, Kondra, Monarch, Ulrich, & Walker, 1993), Steve McConnell’s *Rapid Development* (1996), and the work of Caper Jones: *Assessment and control of software risks* (1994)

Iterating, as a risk mitigation strategy

Tom Gilb (1988) wrote “If you do not actively attack the risks, they will attack you.” Iterating will give you an opportunity to actively attach risks.

The waterfall lifecycle

A simplistic approach to software development can be caricatured as:

1. Let us define completely the requirements
2. Then once this is done, let us do a complete design
3. Once the design has been demonstrated to satisfy all the requirements, let us write all the code
4. Then we can test the code relative to the requirements
5. We can then integrate all the pieces of the system, and deliver.

This is known as the waterfall lifecycle model, because the outcome of each of the phases, flows into the next phase, like a cascade, and there is apparently very little need for feedback.

It would actually be very nice if we were able to do this in software. Waste is minimized, it seems rigorous and rational. And moreover, this is after all the way other engineering disciplines approach their work, civil engineering for example.

But in software development, although many organizations have tried very hard to make it work, it consistently fails for several reasons:

1. It assumes stable requirements; this is rarely the case. Requirements become gradually understood as the project is being developed. Moreover the business environment is often volatile and there are legitimate reasons to change the course of the project.
2. It relies on the analysis of intermediate artifacts, mostly review. There are few techniques and tools available to check the requirements, their completeness, consistency, accuracy. Similarly there is very little we can assert about the quality of a design. In software, we do not have like other engineering disciplines the laws of physics to validate mathematically or by computer modeling or simulation the main characteristics of our designs.
3. It leaves very little room for learning: we are supposed to be suddenly very good at software design, and we have only one chance at it.
4. The feedback loop is rather limited, defects found in a phase may affect decisions made several phases earlier, and have big rippling effects. And the more general feedback on the process is too long, since we will be able to apply what we’ve learn only on the next project.

But above all there are certain issues, problems, defects that will be discovered very late in the lifecycle, and their thorough correction almost always lead to costly overruns and sometimes project cancellations.

When analyzing such failed projects, it appears that the risks were known quite early, but only manifested themselves late in the lifecycle, when there was code to test,

components to integrate. As Barry Boehm noted, the cost of fixing a problem raise exponentially as it is discovered late in the cycle [ref?].

The main issue is that the risks are addressed, confronted too late. But unlike other discipline, software is eminently soft, malleable, like play-do, and we can take advantage of this to develop it iteratively.

Iterative Lifecycle
We can instead:
1. Develop some of the requirements, starting with the most critical ones
2. Identify some of the major technical risks
3. Design, code and test just enough software to address these requirements and mitigate or even completely resolve some of the risk.

Then we can reflect on what we have learned, at the level of the product (requirements), the technology (language, tools), the process we use, and even the people, take corrective actions where needed, and then iterate again.

1. Develop more requirements
2. Evolve the design
3. Add or modify the code
4. Run an expanded set of tests

etc.

Each such iteration:
• produces an executable “release”, running software that can be tested,
• includes integration and test, gradually building up the product,
and the earlier iterations should address, resolve or mitigate our greatest risks.
What we have done is shift risk resolution forward.

Why tackling the biggest, most challenging risks first? Doing the hard things first? should you ask. Why not doing a few easy iterations, with the easy stuff, that is risk-less, giving us time to think more about the hard, risk-prone stuff? Well, if a project must die because of some unsurmountable problem, be it technical or resources, we want it to die as early as possible, not as late as possible after we have spent all the budget and time, and some more. If we need to cancel a project, or significantly rethink it, we want to do this as early as possible. Therefore we should tackle the hard, unknown, uncertain, risky stuff first.

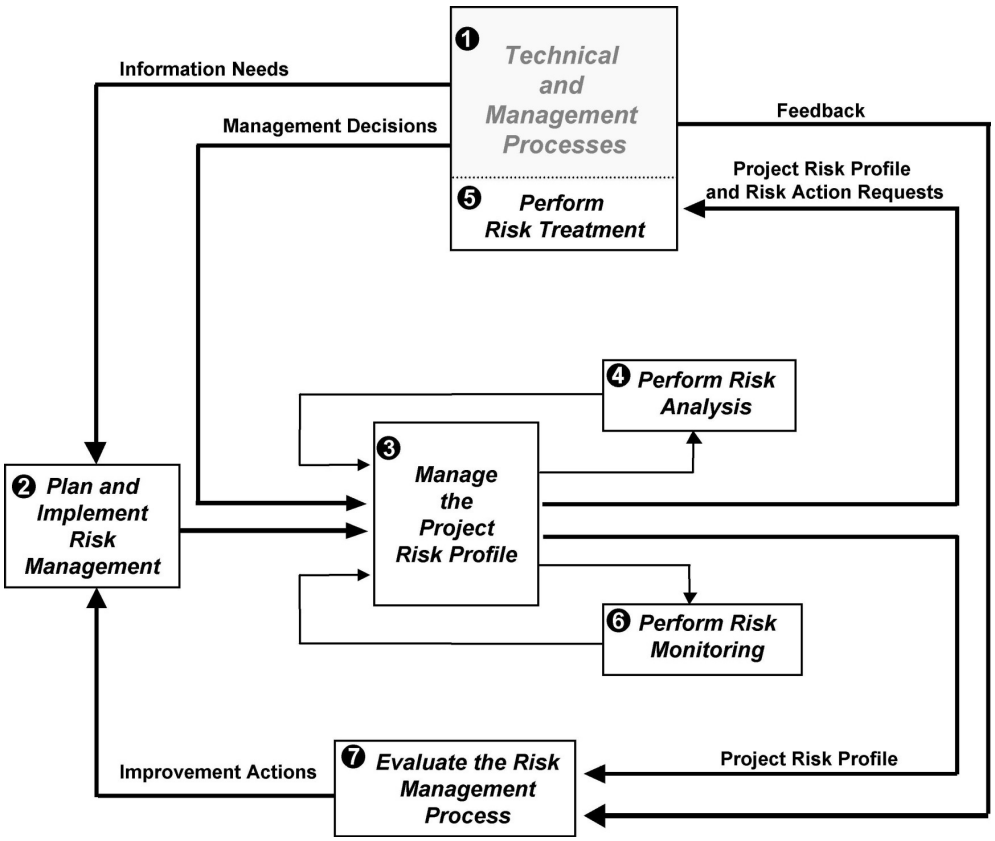
We will revisit the iterative lifecycle many times throughout this course, in particular in the 2 chapters 6 and 7 on managing time. For now, let us just see iteration as our number one practice for mitigating risks and reducing other uncertainties.

More about risk and uncertainty
[Cancellation
Opportunities

Risk and value (Lister: “Avoiding a risk usually lowers the value” and “no risk projects have no value”]

Standards

The risk management process described in the IEEE Standard on Software Risk Management: IEEE std 1540-1998, is a bit heavy weight, for most software projects; the process presented in this chapter is not too far from it, though much lighter-weight. The gist of the standard rests in this figure:



The risk profile in IEEE 1540 corresponds roughly to our *risk list* (as it evolved over the course of the project, that is, with historical information).

ISO/IEC published a revised version of the *Guide 73* (ISO/IEC, 2009b) and a *Principles and Guidelines for Risk Management* (ISO/IEC, 2009a).

Risk Management Plan

One of the recommendations of IEEE std 1540 is the creation of a document called a *risk management plan*. In many project this is just a section of a larger, more comprehensive *Software development plan* (SDP) (see chapter 4). On a large, long-lived complicated project, with multiple teams involved, or a project in a regulated environment, you may want to define explicitly how risks are being managed in the

project: what are the artifact (risk list), and the processes used to create and maintain them, the responsibilities.
In many large organization, there is a set of predefined templates for such plans, where most of your effort will be to tailor one for your specific needs.
A risk management plan can be useful to make sure everyone has the same understanding of risk, impact, probability, exposure, and how risks are being managed in the risk list (or some more elaborate artifact), and by whom.

Summary

- Build up early and maintain throughout the project a list of risk.
- At regular intervals, expose the top 10 risks, especially the direct risks, that is, those the project can do something about.
- For all direct risks, define a risk resolution strategy: mitigation, avoidance, contingency.
- As more is known about the risks, update the list, once a week at least.
- Even if it is a very modest one, define explicitly your risk management policy, for everybody to understand and participate.
- Iterate: define a little, design a little, build (code) a little, and test a little.
- Drive the planning of early iterations by risks: make sure you address your most severe risks, and resolve them in early iterations.

For further reading

- In his all-time (1991) classic, "Software Risk Management: Principles and Practices," Barry Boehm defines the key risk management concepts: impact, exposure, mitigation, and so on, from a software development perspective. He presents a six-step strategy and a handful of simple practices, such as monitoring the "top 10 risks."
- Dick Fairley (1994) presents a simple seven-step approach, not too different from Boehm's, which he illustrates with an extensive case study.
- Art Gemmer (1997) provides additional practical advice. In particular, he elaborates on the concepts of probability and impact, and how to elicit them from your stakeholders and team members.
- The articles above and 3 others have been packaged by the IEEE Computer Society in a small primer on software risk management (Kruchten, 2007).
- Karl Wiegers's article "Know your enemy: Software risk management" (1998) has become chapter 6 in his book "Project Initiation" but can still be found on line.
- Two important sources of risk ideas are the SEI's *Taxonomy of risk* (Carr, et al., 1993) and the work of Caper Jones: *Assessment and control of software risks* (1994).
- Finally, consult the IEEE std 1540-2001: *IEEE Standard for Software Life Cycle Processes—Risk Management.*
- For a more thorough treatment of risk, the book *Waltzing With Bears: Managing Risk on Software Projects* by Tom DeMarco and Tim Lister (2003) reads almost like a novel. From the same Tom DeMarco, *The Deadline* (1997) is actually a novel about software project management.
- The great classic on the waterfall model is the paper by Winston Royce (1970) "Managing the development of large software systems"; although many people point

at this paper as the source of all waterfall evil, Royce introduces the concept of iteration.

- Barry Boehm introduced his “Spiral model of software development and enhancement” in (1986), but as you will read in (Larman & Basili, 2003) the history of iterative development is older than 1986.

Bibliography[1]

Boehm, B. W. (1986). A Spiral Model of Software Development and Enhancement. *ACM SIGSOFT Software Engineering Notes, 11*, 22-42.

Boehm, B. W. (1991). Software Risk Management: Principles and Practices. *IEEE Software, 8*(1), 32-41.

Carr, M., Kondra, S., Monarch, I., Ulrich, F., & Walker, C. (1993). *Taxonomy Based Risk Identification* (Technical report No. SEI-93-TR-006). Pittsburgh, PA: Software Engineering Institute.

DeMarco, T. (1997). *The Deadline--A novel about project management*. New York: Dorset House.

DeMarco, T., & Lister, T. (2003). *Waltzing With Bears: Managing Risk on Software Projects*: Dorset House Pub. Co.

Fairley, R. (1994). Risk Management for Software Project. *IEEE Software, 11*(3), 57-67.

Gemmer, A. (1997). Risk management: moving beyond process. *Computer, 30*(5), 33-43.

Gilb, T. (1988). *Principles of Software Engineering Management*. Reading, MA: Addison-Wesley.

IBM. (2003). Rational Unified Process (Version 2003). Cupertino, CA: IBM Rational Software.

IEEE. (2001). IEEE std 1540-2001 -- Standard for Software Life Cycle Processes—Risk Management. New York: IEEE.

ISO/IEC. (2009a). *IOS/IEC 31000:2009 Risk management -- Principles and guidelines*. Geneva: ISO.

ISO/IEC. (2009b). *ISO/IEC Guide 73:2009, Risk Management - Vocabulary*. Geneva: ISO.

Jones, C. (1994). *Assessment and Control of Software Risks*. Englewood Cliffs NJ: Yourdon Press.

Kruchten, P. (2007). Risk Management in Software Development: A Primer Available from http://www.computer.org/portal/web/buildyourcareer/ts001

Larman, C., & Basili, V. R. (2003). Iterative and Incremental Development: A Brief History. *IEEE Computer, 36*(6), 47-56.

McConnell, S. (1996). *Rapid Development*. Redmond, WA: Microsoft Press.

Robertson, S., & Robertson, J. C. (2004). *Requirements-Led Project Management: Discovering David's Slingshot*: Addison-Wesley Professional.

Royce, W. (1970, Aug. 25-28). *Managing the Development of Large Software System.* Paper presented at the IEEE WESCON, Los Angeles.

---

[1] Bibliography will be regrouped at the end of the book. It is easier for the draft to build it chapter by chapter.

Wiegers, K. (1998). Know Your Enemy: Software Risk Management. *Software Development Magazine, 6*(10), 38-42.