The Dark Side of Agile Software Development

Andrea Janes
Free University of Bolzano/Bozen
Piazza Domenicani 3
39100 Bolzano, Italy
andrea.janes@unibz.it

Giancarlo Succi
Free University of Bolzano/Bozen
Piazza Domenicani 3
39100 Bolzano, Italy
giancarlo.succi@unibz.it

Abstract

We believe, that like most innovations, Agile has followed the Gartner Hype Cycle and it has now reached the Trough of Disillusionment, where it is currently stuck. Moreover, we believe this is due to a "guru phenomenon."

However, we think that Agile can make a step forward. Our experience lead us to the conviction that through the application of a suitable quality improvement paradigm Agile will be able to reach what Gartner's experts call the Slope of Enlightenment and eventually the Plateau of Productivity.

Categories and Subject Descriptors D.2.9 [**Management**]: Software process models

General Terms Management

Keywords Agile software development; Hype Cycle; Quality Improvement Paradigm

Introduction

We have seen many innovations going through a phase of over-enthusiasm or "hype" and – when the expectations turn out to be unrealistic – to result in a generalized disappointment.

New technologies offer new opportunities so the hope to overcome unsettled problems with the existing solutions is high and generates excessively optimistic expectations.

We think that Gartner's Hype Cycle [1] describes well this phenomenon. It encompasses five stages that many innovations go through before becoming productive. Not all innovations manage to reach the fifth stage [2]; some become obsolete before the end.

The five phases are shown in figure 1.

Onward! 2012, October 19–26, 2012, Tucson, Arizona, USA.

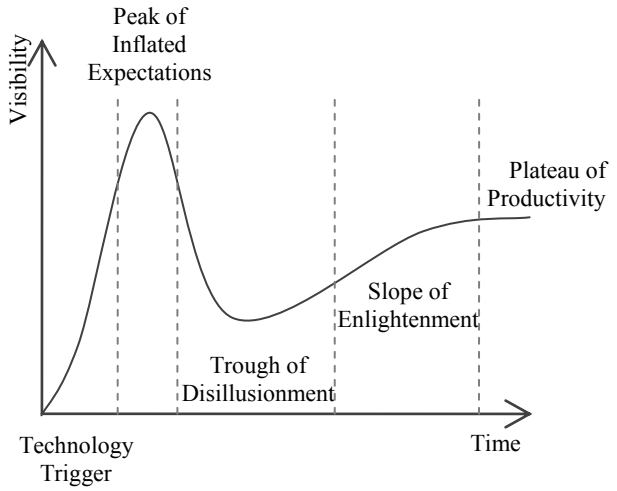

Figure 1. Gartner's Hype Cycle

The Hype Cycle begins with the technology trigger – an innovation that generates significant attention in field. As it happened for example with the e-Business hype of 1999, resulting in the burst of the dot-com bubble of 2000, it can happen that the innovation is evaluated too optimistically, and this leads to unrealistic expectations. In Gartner's Hype Cycle this phase culminates in the Peak of Inflated Expectations.

In the Peak of Inflated Expectations, the actual hype, there may be some successful applications of a technology, but there are typically more failures. According to Gartner, this is where it is clear that the excessive expectations cannot be met and so the innovation gets less and less attention and enters the Trough of Disillusionment – it becomes unfashionable. Still some companies continue to use it to further explore its real benefits and in doing this they pass through the Slope of Enlightenment. This learning phase identifies where the technology is useful and where not, how it should be used, and which disadvantages exist.

The final phase – for those technologies that do not become obsolete before reaching it – is the Plateau of Productivity – a phase in which the benefits of the innovation are widely demonstrated and accepted. The final height of the plateau (the visibility of the technology) depends on how applicable the innovation eventually is.

We remember when Java was at the peak of the hype cycle some years ago. The new language was traded as the cure for many illnesses. It was a new kind of Aspirin, developed to eradicate portability and maintainability issues, yes, to eliminate bugs in general. Java is not a hype anymore, it got better: it reached the Plateau of Productivity [37]. We do not use it blindly, whatever problem we face. We know now when and why we should use it and when we should better let our fingers from it.

This is what the Plateau of Productivity represents: a stage in which potential users are able to estimate the outcome of using a given technology.

There are technologies that become obsolete before reaching the Plateau of Productivity. This occurs to technologies that are hard to apply, where the benefit is not clear, that are complicated to use, are replaced by other technologies, and so on.

The experience of this last decade evidences that Agile Methods are often praised for virtues they do not have and are often blamed for deficiencies that they also do not have. Using a well-known expression, Agile Methods are often criticized for their virtues and not for their vices. Understanding the real key limitations of agility and the alleged limitations that are just wrong reading of the works of the "founding fathers" is an essential step to take full advantage of the excellent ideas present there to build solid software engineering processes and products.

We believe we have seen Agile Methods follow the Hype Cycle only to become mired in the Trough of Disillusionment. Let us show you why.

Agile: stuck in the trough

We think that Agile Methods brought a new perspective to software development: the remarkable focus on agility, the ability to adapt easily to a variation of the requirement or of the environment, to provide value to the customer.

In our view, the waterfall process was based on the assumption that the requirements are there (in the minds of the customers), we just have to work hard enough to get them. If modifications were needed during development, we blamed the analysts for not doing their job good enough. Indeed, this did not address the real problem: requirements emerge during the production process and that we must be flexible to cope with them when they arise. On the contrary, the waterfall advocated as a solution more planning.

Several alternatives to the waterfall were proposed through the years to address the problem of capturing properly requirements. Only the spiral model [45] acknowledged the intrinsic need of a better management of always changing requirements. Still the spiral model was more an academic endeavour and never really became an industrial standard or better, Agile was the natural industrial translation of the concepts of the spiral model.

The Agile Guru understood that the management of requirements was a major plus for their approach. So they went ahead and they overdid. In his book Kent Beck reached the point of clarity that Agile manages to keep the cost of requirements flat [24] (figure 2) in contrast to traditional methods where the cost of change increases exponentially. The statement of Beck is not false: giving up upfront planning and defining requirements only when there is a strong need for such definition lowers the cost of their modification.

It is beyond the scope of this essay to discuss whether the cost of change can be really flattened. The point is that such claim was really appealing and lead consultants not always of the calibre of Kent Beck to claim that it is always possible to keep the cost of changing requirements constant. We had the privilege of watching Kent Beck coding, and, part of the ability of keeping the cost of change constant also related to his ability of producing beautifully crafted Smalltalk code.

To keep a long story short, not all the consultants promoting Agile had the skills of Kent Beck in organizing the process and the code and not all the projects managed by such consultants were amenable to the application of Agile. But still companies, CEOs, managers were convinced by them. Agile became the philosopher's stone.

Right after the publication of the Agile Manifesto [35], Agile Methods have been proposed as the universal solu-

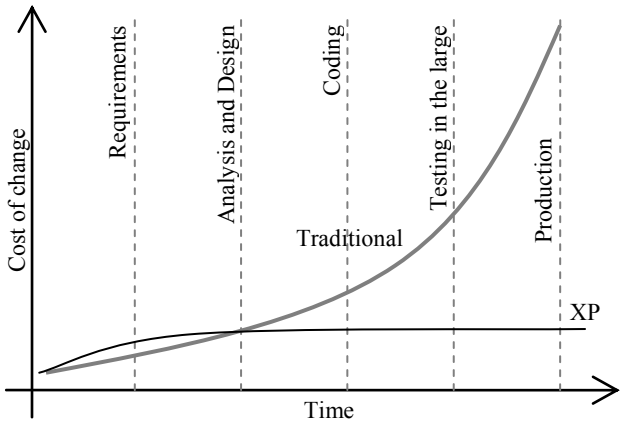

Figure 2. Traditional and agile cost of change curve (adapted from [24])

tion to a common problem: the strong demand to higher flexibility during the development of software to increase productivity and customer satisfaction through the entire development of a software system.

The problem to flatten the cost of change curve is indeed hard but advocates of Agile Methods have been very effective in their propositions, so Agile Method have gained an enormous popularity and still they are popular.

Altogether, Agility became a hype. The term agile was not very popular in computer science before the advent of agile software development. Other words were used to describe agility, e.g., flexible or rapid, while today (almost) every computer science term can be found on Google combined with the word Agile (see table 1).

If we look more closely at the evolution of the most popular Agile Method, Extreme Programming, we notice that its popularity has followed quite closely the Gartner Hype Cycle: after the first hype, there has been a period of disillusion. Google Trends [3] shows a diminishing number of people searching for websites containing the term "extreme programming" (figure 4). Similarly, discussion forums such as the Yahoo! group extremeprogramming [4] show a diminishing participation in the discussions (figure 3).

These indicators also confirm Garner's report Hype Cycle for Application Development, 2010, in which Agile Development Methods are considered as "sliding into the trough [5]".

Agile Methods passed the peak of inflated expectations, the hype is over. The attention to the topic Agile has reduced. Agile is not anymore considered the solution for everything, the computer science departments of book stores are not anymore paved with books about agile. The attention went to other buzzwords such as Lean and Kanban.

Table 1. Hits on Google searching for programming terms combined with "Agile" as of April 13, 2012.

Search term	Hits on Google
"Agile testing"	716000
"Agile web development"	364000
"Agile product management"	323000
"Agile management"	279000
"Agile database development"	186000
"Agile modeling"	117000
"Agile ruby"	103000
"Agile AJAX"	100000
"Agile SOA"	79600
"Agile architecture"	66100
"Agile documentation"	35100
"Agile game development"	32100
"Agile offshoring"	28900
"Agile embedded software development"	7520
"Agile requirement engineering"	2260

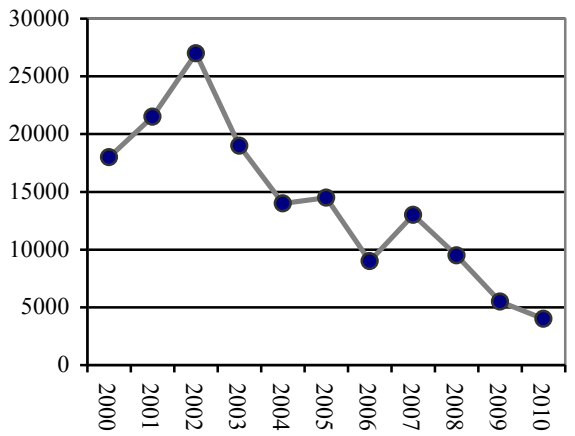

Figure 3. Number of messages posted on the "extremeprogramming" Yahoo! discussion group as of January 6, 2011

It is clear that the inflated expectations could not be fulfilled. Too often we should have told to the many consultants proposing to achieve enormous results through the blind application of part of the practices described by Kent Beck in his book what Lloyd Bentsen told to Dan Quayle on the 5th October 1988 when the future Vice President compared himself to Jack Kennedy: "Consultant, we know Kent Beck, we worked with him, and you are not Kent Beck." But still many, too many companies believed in them and their failure was reflected on a fall of the trust in Agility.

In trying to understand why agile is stuck, we first looked at the specific practices and concepts that made agile interesting, but now seem to be holding it back. Therefore, we think that it is now time to understand the underlying principles of Agile Methods, to determine the time, the scope, the means, and the extent of their applicability. We think that this is what stops software engineers and software companies to take full advantage of agility: to clearly understand when they happen to be useful and when not.

The sceptics rise

The adaptive nature of Agile Methods was conceived with a premise in mind: it is not possible to plan every detail in advance. This goes against conventional wisdom, which teaches us to "first think, then do."

Software development is often compared with an idealized view of construction works: since it is possible to plan

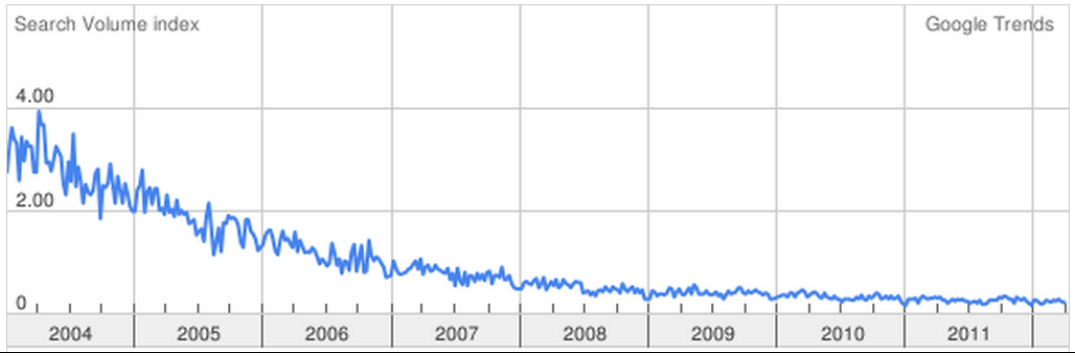

Figure 4. Google Trends search volume index for the search term "extreme programming" as of April 13, 2012

a house completely and then to construct it according to the plan, the same must be possible for software.

We heard this comparison with construction works so many times, that we need to make clear that such a view on construction works is simply wrong. Also when building a house it is not feasible to plan everything in advance, e.g., what kind of view one will have on the 3rd floor from the bedroom.

Companies operating in the construction business provide the most disastrous examples of cost and time overruns whenever something new – with only few opportunities to reuse past experiences – had to be built. There are endless examples like the following:

- The construction of the stadiums of the 1990 FIFA World Cup in Italy cost about 3 billion euros. On average, the costs were 84% higher than the planned costs [38].
- The estimated costs for the construction of a the new terminal Skylink of the airport Vienna were 400 million euros in 2006 with an estimated duration of 2 years. In 2012, when the works finished four years later than planned the costs were 800 billion euros because of "Bad planning, mismanagement, lack of a general contractor, high consultant fees, and non-inclusion of interface projects in the calculation [39]." Altogether, the construction of the new terminal required three times the planned time and twice the planned expenditures.
- The renovation of the Elbe Philharmonic Hall in Hamburg, Germany, in 2007 was scheduled to be finished in 2010 with an estimated cost of 114 million euros. Currently, the costs are estimated to be 476 million euros and the date of completion is estimated to be 2014 [47].

Our experience leads us to think that something similar happens also to software: if we had to rebuild something that had already been built, we would have only a limited time and cost overrun. Look at the software houses doing web pages: typically they are pretty much able of predicting time and cost of development.

In software there is also an extra complexity that makes it more unlikely that a new project is similar to a previous one: the technology evolves very rapidly and so the desires and expectations of the customers.

The bias and incomplete comparisons of software development with constructing a house has the purpose of (erroneously) proving that agile software development does not follow a rational sequence of activities and neither takes into account that there are bazillions example of failing projects in the construction industry nor that there is an intrinsic additional complexity in developing software linked to the high speed of change of the underlying software infrastructure.

This "conventional wisdom" was already criticized in the '80s by Peter Naur, in his seminal work "Programming as theory building," where he remarked that "much current discussion of programming seems to assume that programming is similar to industrial production, the programmer being regarded as a component of that production, a component that has to be controlled by rules of procedure and which can be replaced easily. […] At the level of industrial management these views support treating programmers as workers of fairly low responsibility, and only brief education [42]."

This reminds us the concept of "Scientific Management" of Frederick W. Taylor that proposes that the "management take over all work for which they are better fitted than the workmen [43]." This statement builds on the assumption that workers deliberately work slowly to keep the productivity low so that all workers are needed.

We notice remarkable analogies between industrial and software production. In industrial production, Scientific Management became popular to improve productivity and it is based on rigid and detailed upfront planning, division of labor and specialization of the workforce and a clear formalization of the problem and division of a large problem in small sub-problems. Later, in the '60s pioneers like Frederick Herzberg began to investigate how to improve productivity by increasing the motivation of workers and by involving them more in the decision making processes through what was called Job Enrichment [44]. We can draw a parallel: in software production the waterfall process resembles scientific management and the propositions of Agile are analogous to those of Herzberg.

We believe that – as in industrial production – there are repetitive tasks that are better managed in one way and there are creative tasks that are better managed in another way. All four methods, Scientific Management, Job Enrichment, Waterfall, and Agile Methods have their right to exist, if used within the right context.

The problem arises when one method is used in a context where it does not belong to.

Altogether, Agile went down in the popularity, and the question then is why it has not yet gone up again.

A major impediment we see is that agile software development goes against the current so-called best practices of project management. Software development projects are seen as any other project and therefore project managers assume they should be managed as any other project. Moreover, the failure that we discussed before, the one coming from consultant thinking that a blind application of Agile was the panacea, triggered the "good school" to revert back to the most traditional and standard way of managing projects.

Let us see, for example, the Guide to the Project Management Body of Knowledge (PMBOK, American National Institute standard ANSI/PMI 99/001/2008) is a recognized standard for the project management profession [10]; this is the bible of project management, what every manager knows and think he or she can apply to any project, including software project. The PMBOK has a clear reference project model, the waterfall model: the generic project life cycle is shown in figure 5 – and indeed they claim it holds for any project; well it is like taking the time machine and going back 40 years.

Project managers trained using the PMBOK (the vast majority) expect a waterfall-like progression of the different activities of the project. The impact of stakeholders, risk, and uncertainty at the beginning of the project are considered high but then they diminish over time. The cost of changes are low at the beginning but increase with time – well, we know this story fairly well.

The PMBOK just briefly (a paragraph of 8 lines of text) mentions the possibility to divide the project in project phases (consisting each of an initiating, planning, executing, and closing step), and to perform these phases iteratively, but reserves this solution only for "largely undefined, uncertain, or rapidly changing environments such as research [10]."

Altogether, we assume that managers without IT background consider optimal what they learned in school. Agile is a strange beast for them. It is likely that they think that Agile, among many other negative habits:

- promotes inadequate preparation: it avoids big up front designs as it is expected that requirements should be analysed and studied in depth only a short time before their implementation;
- accepts exploding costs of changes: the project manager assumes a traditional costs of change and so late changes trigger such inflated costs;
- ignores the consequences of its high risk approach: while in a traditional project the stakeholder influence, risk, and uncertainty are expected to reduce over time (figure 6), and the stakeholder is made well aware of this, Agile let the stakeholder change the priorities of the project throughout the project itself, with possible catastrophic consequences.

To put it in other words: if one looks at Agile Methods from the perspective of the Guide to the Project Management Body of Knowledge, the conclusions are that Agile is not suitable way to manage software projects.

The mismatch in the expectations originates from the

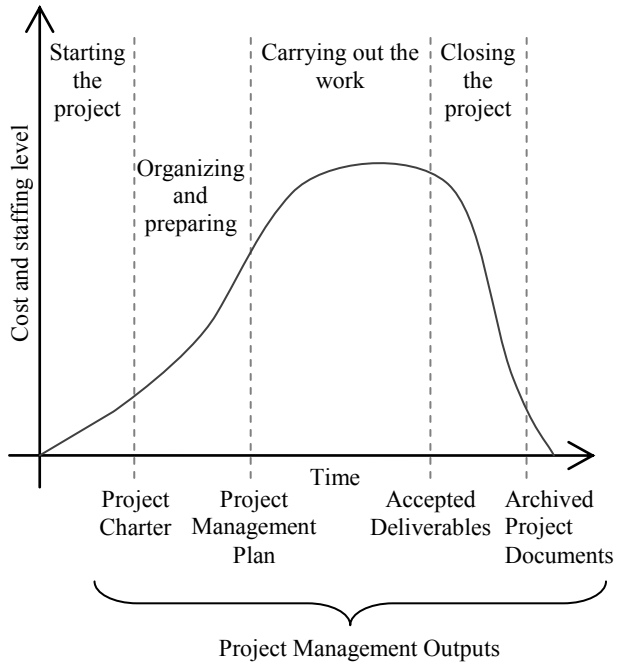

Figure 5. Typical cost and staffing levels across the project life cycle [10]

approach to use the same project management approach independent from the context.

This mismatch can be seen also with other stakeholders, not only managers: according to our experience, when customers are not familiar with software engineering at all, they imagine it as something they know and behave accordingly. Some imagine it as building a house, assume that first you have to plan and then you construct according to the plan. Such customers react very annoyed when constantly asked for feedback or requirements throughout the project since they planned to invest time in the project only at the beginning. Once one of our project partners said: "Why do you ask me? You are the expert!"

Another thinking model is to see software development as to cook – you just need to assemble different components in the right combination. When developers are confronted with problems they have never seen before and need time to develop a solution this is seen as the fault of the developer that does not know all the available components. A similar way is to see software developers like configuring a video recorder: there are a predefined number of switches and buttons and the skills of the developer decide whether he or she is able to setup the system in the right way or not.

In addition, only few customers are aware of the difficulty of developers to guarantee a correct functioning of the software since they are not domain experts.

The described difficulties do not only apply to non-technical people. It is also hard for professional developers who have been trained and whose experience is mostly in formal, waterfall-like methods and approaches (e.g., structured programming, structured analysis, UML, CMM ISO 9000, etc.) to appreciate that projects could be done differently. They have the Miss Andrew syndrome: Miss Andrew is the old nanny of Mr Bank in Mary Poppins, and she think that the only way to educate children is through harsh control. Thinking of the Mary Poppins sugar-based approach sounds like something really odd.

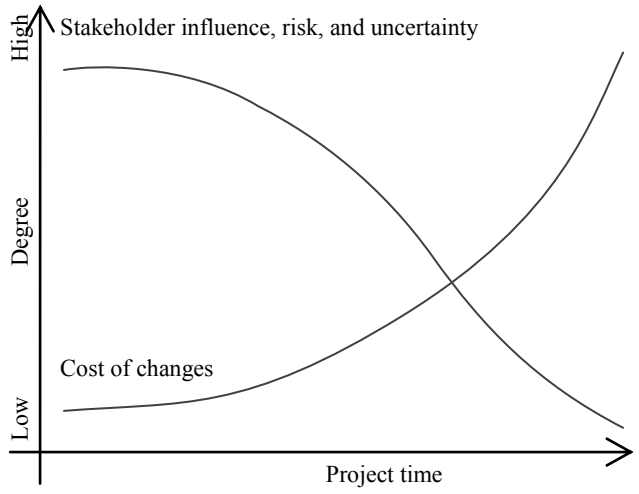

Figure 6. Change of stakeholder influence, risk, uncertainty and the cost of changes based on the project time [10].

In summary, different types of stakeholders have expectations that for them are obvious and therefore are not communicated. Nevertheless it is necessary to deal with them: managers might come to the conclusion that what Agile Methods are proposing does not lead to the aimed objectives, i.e., that Agile Methods are not aligned to the business goals. Customers might come to the conclusion that such a company is incompetent because its developers constantly ask for feedback.

To respond to the scepticism and to be able to productively work together, all stakeholders need to agree – as recommended by meeting facilitators – on:

1. Where we are: an assessment of the current situation
2. Where we want to go: a description of the goals
3. How to get there: a plan to achieve the goal

The mismatch we described above originates from a different understanding of the three points above. Particularly, the point "how to get there" requires that all stakeholders can explain which goals they pursue and which means they intend to use.

To claim "to follow an Agile Software Development Method" is not enough. As we have explained that the different stakeholders have an entirely different understanding of how development is carried out, it is necessary to explain what this means, which practices will be used, why these will be used, and what kind of outcome is expected.

Unfortunately, this is not always possible, and defending agile approaches is sometimes difficult since they acquired a reputation in the management community as a sloppy, undisciplined method to hack, not develop software. The next section gives some reasons why this happened.

The dark side of Agile software development

Agile methods address critical shortcomings of traditional software development approaches; therefore, they have created a notable interest in the software engineering community. Today, it is almost impossible to find a practitioner in software production that has not heard about Agile Methods.

However, with all the popularity Agile Methods have gained, it is surprising that they have not yet reached the Slope of Enlightenment, meaning the precise boundary of their applicability are not yet fully understood.

This all boils down to the fact that Agile Methods are still not a recognized standard to develop software. Whenever we have a new project with somebody we have to

begin with something like: "Well, we have a different way to do things, we will not collect all requirements upfront now, then ..." We always have to introduce Agile Software Development as something alternative, and always the person on the other side of the desk seems to have a mixed feeling about our approach.

Unfortunately, often the proponent of Agile Methods instead of providing a rational explanation of their application cites the gurus, the holy texts, and, worse, does not really understand whether the Agile Method first should have been applied in such context, and just feels like being a prophet.

In any case, if anything goes wrong, Agile Methods are blamed: we just followed the fad!

The mission of agility is to increase the ability to change things also late in the development process to be able to produce value throughout the process.

The need to be more flexible is beyond any doubt overwhelming. This explains why agility became immediately so popular: it was seen as the answer to such need. The metaphor used was luring: agility is the opposite of large and heavy – so if software systems and software processes become too large and heavy, let's be agile!

It was so fashionable to be agile that always more people claimed to be so, even if they were not. In particular, a number of consultants and managers adopted quite liberally the term agility to promote their services [6]. Agility was then interpreted just as the cut in complexity, a rather simplistic and unjustified cut in complexity whatever the term complexity was – documentation, good development practices as design, etc. And this was against the ideas of the proposers – we recommend reading the seminal paper of Martin Fowler on agility in design [7].

A *dark* side of Agility emerged.

Most likely, also against the wills of the proposers, enthusiastic early subscribers to the Agile Manifesto became zealot. They wanted to do more, to do better. So they read the agile manifesto, but then, again the Miss Andrew syndrome emerged. They wanted to be better, to support Agile Methods with all their strength, and the means became an end. They interpreted the Agile Manifesto as in figure 7.

The dark side of agile caused a set of misconceptions about agile software development – still common today – such as that it is forbidden to document or to plan within Agile Methods. The two phenomena also evidence what is still today missing in the application of the Agile Manifesto and it is highlighted by the last prescription of the Dark Agile Manifesto: teams need to understand how much to value the items on the left – how much working software, how much customer collaboration, how much responding to change – or the items on the right – how much planning, how much contract negotiation, etc.

We are uncovering ~~better~~ **the only** ways~~s~~ of developing software by doing it and ~~helping~~ **teaching** others do it.
Through this work we have come to value:

Individuals and interactions ~~over~~ **and not** processes and tools
Working software ~~over~~ **and not** comprehensive documentation
Customer collaboration ~~over~~ **and not** contract negotiation
Responding to change ~~over~~ **and not** following a plan

That is, ~~while~~ **since** there is **no** value in the items on the right, we value **only** the items on the left ~~more~~.

Figure 7. The dark agile manifesto.

Some the misconceptions about agile can be directly linked to the four statements of the dark agile manifesto, a cynical interpretation of it is [8]:

- Individuals and interactions over processes and tools: "Talking to people instead of using a process gives us the freedom to do whatever we want."
- Working software over comprehensive documentation: "We want to spend all our time coding. Remember, real programmers don't write documentation."
- Customer collaboration over contract negotiation: "Haggling over the details is merely a distraction from the real work of coding. We'll work out the details once we deliver something."
- Responding to change over following a plan. "Following a plan implies we have to think about the problem and how we might actually solve it. Why would we want to do that when we could be coding?"

Such translations show that to some, Agile Methods appear as approach that advocates coding without discipline, planning, and documenting: "this is nothing more than an attempt to legitimize hacker behaviour [8]."

In programming, "a hack is something we do to an existing tool that gives it some new aptitude that was not part of its original feature set [9]." So by hacker behaviour the critics claim that the adopters of Agile Methods do not develop software following standards, processes, policies, i.e., do not follow state-of-the-art practices that have been proven to be necessary or to contribute positively to the successful outcome of the project.

The agile community developed another term to describe such behaviour and to dissociate itself from it: the cowboy coder. The cowboy method ignores defined processes to be faster.

Agile Methods, e.g., Extreme Programming actually are based on Best Practices and follow state-of-the-art methods. Agile Methods acknowledge their importance and combine them in a way to maximize agility, i.e., the ability of the development team to respond to changes of the requirements.

We partly attribute to agile extremists that Agile Methods acquired a reputation to be for hackers, not serious developers.

One reason that agile extremism could develop is because Agile Methods are not defined precisely. The Agile Manifesto leaves a certain degree of freedom that enabled agile extremists to prosper; another reason is described in the next section.

The question that remains is how we can avoid it, how to understand how much e.g., planning is needed to maximize value, when planning begins to destroy value, and so on.

The guru problem

Agile Methods have been conceived and refined by gurus, e.g., Extreme Programming by Kent Beck, Crystal Clear by Alistair Cockburn, and Scrum by Ken Schwaber. This might be one of the intrinsic reasons for the problems we mentioned before: the language of the gurus must be persuasive and often elusive.

The guru is the person with wisdom – he or she knows what, when, and how things should be done to achieve the desired goal. It is the interest of the guru to hide the assumptions on which the rules are based, on which previous works he or she based his or her findings, how he or she verified that what is claimed really works. For example, it is not clear – reading the Agile Manifesto – why it is good to focus on individuals and interactions over processes and tools: it is not stated why and how software development should benefit from it. The guru speaks using luring metaphors, fascinating analogies, taking words from poems. The disciples are fascinated, appreciate that he or she earns a lot of money, and simply tries to replicate the teachings without going in deep.

The guru has no advantage in making his or her followers independent adults, otherwise his or her role as a guru would vanish. In this way, the adopters remain dependent on the guru: the adopters need the guru to continue to use the method in cases not described by the guru up front, e.g., how it can be extended, in which order the different practices should be adopted, etc.

Let us look more in detail at, for example, the concept of emergent design. In Agile the architecture of a program is not defined up front but "emerges" as the functionality is developed. This is indeed a very tempting idea. It gives us the hope to push the cost of change curve down even for architectural decisions. However, this approach does not work in all possible situations, but it requires an object design philosophy that is based on the structure of the problem domain [46]. This is not always the case. The guru does know when it is the case and has several consulting opportunities. So he or she picks smartly only those where such approach is suitable and winning. Unfortunately, the poor consultant does not fully appreciate the scope of applicability of emerging design and does not have so many job opportunities, so he or she simply applies this concept throughout all her or his possible endeavours.

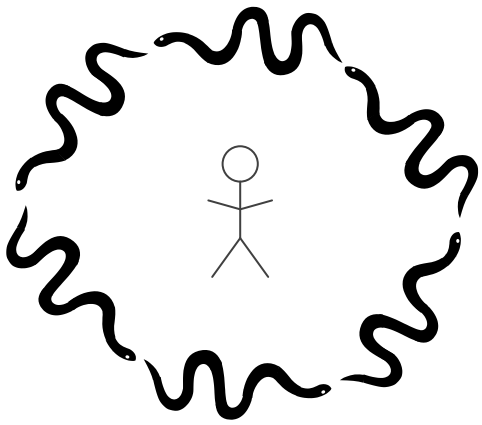

Figure 8. Extreme Programming depicted as a ring of poisonous snakes [14].

Altogether, the result is that the market is dominated by a few "enlightened" gurus that keep the knowledge secret among themselves, followed by many quacksalvers.

The described strategy of the gurus works because a precondition is met: the followers of the guru – programmers, managers, requirement engineers, project managers – are looking for a silver bullet: they want a simple, safe method that solves their problems. And gurus are willing to give it to them.

This situation leads to a number of issues because we have to treat the method as a black box: it is not possible to use its nuts and bolts for an evaluation; they have to be reengineered (or ignored) if anything else than the entire package has to be evaluated.

Having to treat an Agile Method as a black box, we don't know the rationale behind the different constituent parts. Whenever we have to assess something complex, conventional wisdom tells us to verify its constituent parts. Let's say we want to evaluate the quality of a car. One step would be to check the quality of the tires, the engine, etc. If we want to do that for an Agile Method we have to conduct a study on our own (as for example in [11]), since "agile method practices sometimes lack clarity and the underpinning rationale is often unclear [12]".

Some authors define Agile Methods, Extreme Programming in particular, as irrational and vague [13, 14].

Other authors claim that the agile method practices are on a too high level of abstraction which causes an inconsistent interpretation and implementation [15, 16, 17].

Not having a clearly stated rationale behind method and practices makes it also difficult to tailor a method to specific needs [18, 19, 20, 21, 22, 23]. Without this knowledge we don't know what we are risking if we omit one method or if we extend another.

Stephens and Rosenberg compare Extreme Programming with a ring of poisonous snakes (sometimes depicted as in figure 8), daisy-chained together. Each snake represents a practice that has issues that are compensated by the use of another practice. „All it takes is for one of the snakes to wriggle loose, and you've got a very angry, poisonous serpent heading your way. What happens next depends on the snake and to an extent on how good you are at damage control [14]."

Stephens and Rosenberg show that the different practices depend on each other such as:

1. Requirements are elicited gradually, which – since developers do not have a complete picture – can result in wasted effort. This problem is alleviated by incremental design, i.e. to be ready to change the design incrementally.
2. Incremental design avoids designing everything up-front to minimize the risk of being wrong. This practice is considered safe because the code is constantly refactored.
3. Constant refactoring involves rewriting existing code that was thought to be finished: it went through discussions and thinking to get the design right. It potentially introduces new bugs, but it's considered safe because of the presence of unit tests.
4. Unit tests act as a safety net for incremental design and constant refactoring. With unit tests it is difficult to verify the quality of the design of an architecture. Pair programming alleviates this problem.
5. And so on.

Their sarcastic description implicitly shows that the authors perceive a problem within Extreme Programming: it is unknown how to use the practices individually, to which outcome they contribute how much, etc.

If we want to reach the Plateau of Productivity , i.e., if we want to be able to use an Agile Method efficiently and effectively, we must understand when it pays off, why it works, how it works, how it can be adapted, what the value of single activities is, etc.

Taking the statements of the agile manifesto, we should be interested on how to manage knowledge on how to value individuals and interactions over processes and tools, working software over comprehensive documentation, customer collaboration over contract negotiation, and responding to change over following a plan.

Towards the plateau of productivity

To summarize, different underlying assumptions of computer scientists, economists, project managers, engineers, etc. can lead to different interpretations of what is happening during the project. Agile zealots create excessive expectations that move Agile Methods up the Hype Cycle but at the end of the day cannot be met. In part this is possible because Agile Methods are promoted by gurus, not interested in explaining what is behind their practices.

To overcome these problems we propose to collect and disseminate know-how about Agile Methods in the form of: "To achieve objective o we use tactics y that has the cost c, its performance can be understood observing m and we use this tactic until m reaches the target value v."

We kindly ask the reader to excuse our formalism, but the point we want to make is that the agile community has – and deserves – to understand better the underlying assumptions behind every practice, and if the underlying assumption is just wishful thinking, it has to be stated unambiguously.

The same idea can be described using the terms tactic and strategy. In software architectures, the term tactic is sometimes used to describe a design decision that influences how good a software architecture responds to a quality requirement and the term strategy is used to describe a collection of tactics that help to achieve a common goal [28]. For example, an possible architectural tactic to achieve a performance requirement is to introduce concurrency. A tactic is a decision that helps to achieve a goal.

These are military terms as tactics is the branch of military science dealing with detailed manoeuvres to achieve objectives set by strategy and a strategy is the branch of military science dealing with military command and the planning and conduct of a war.

In the same way we recommend to collect software process tactics, i.e., decisions whether to shape the process in a certain way or not to achieve a given quality (e.g., code quality, time-to-market, low turnover, etc.) about Agile Methods.

Practitioners who want to productively use Agile Methods need to collect such set of tactics to adopt in different circumstances [25, 32, 33, 34]. It is also necessary to know the advantages (the achieved objective o) and the drawbacks (the costs c) of using a specific tactic y. Since software is invisible, and projects can progress towards the wrong direction for a long time until this is noticeable, it is necessary to define a way to understand whether the tactic y is profitable or not (the measure m).

To make an example, the practice of test-first programming, is defined as to "write a failing automated test before

changing any code [24]." The gurus claim that test-first programming helps to avoid scope creep, obtain loosely coupled, highly cohesive code, establish trust among developers, and save time [24]. The missing knowledge to be able to decide rationally about the use of the tactic of test-first programming are the costs c, e.g., how much time it will cost to write tests for every changed code and the achieved objective o, e.g., how much time will be saved because of fewer bugs. In the case of test-first programming we obtain a measurement m that tells us how much of test-first programming we did: we can measure the testing coverage. Still it is unclear how much coverage is good enough, what is v.

Tactics that help to achieve a common goal could be then grouped into strategies. Agile software development strategies could describe a set of tactics that help to achieve a common goal (e.g., fast time-to-market).

The here described knowledge is still missing and practitioners do not seem to collect such knowledge. We want to promote such knowledge collection proposing a scientific method, based on gathering empirical and measurable evidence.

Once such evidence is collected, the reality can speak for itself, and contradict the theories when those theories are incorrect [36], i.e., claims have to become falsifiable.

To be clear, we do not claim that Agile is too informal and needs to be more rigorous and scientific; those that speak about Agile have to become more rigorous and scientific.

Scientific means falsifiable, viz., can be proven wrong. The claim "if you use pair programming, you will double your profits" is indeed falsifiable. I use pair programming in one of my projects and compare the profit with a similar project done using solo programming and may find out that the claim does not always hold.

On the other hand, the claim "if you use pair programming, your code will be more maintainable," is not falsifiable, since the term maintainable is not defined with precision, unless a suitable metrics or a set of metrics have been agreed upon and every reader of the claim is aware of them.

Science developed many different ways to collect and refine evidence to gain the necessary knowledge to formulate the above described rules. We present one that can be used to collect such knowledge: the Quality Improvement Paradigm [27].

A step forward

The Quality Improvement Paradigm is an approach within software engineering to collect and reuse past experiences with the goal to constantly improve. This experience can be used in two ways:

- as a controlling instrument: to compare expected outcomes (those coming from the accumulated experience) with realized outcomes; this will allow to understand the status and the progress of development, to detect variations in the process that are not caused by common reasons, and to understand which counter measures will bring the process back into a controlled state [26];
- as a process improvement instrument; by reusing accumulated experience to improve (e.g., avoid repeating the same errors, standardize work, distribute knowledge, etc.) the software development process.

The second point addresses the problem mentioned above: it helps to collect data about the costs and advantages of using certain tactics. Moreover it might help to identify compatible tactics to define strategies.

The Quality Improvement Paradigm consists of two cycles (see figure 9): the first cycle occurs once during one project, the second takes place continuously during the execution phase. The second cycle represents the feedback cycle to provide early feedback to the process execution to control the process.

The main cycle consists of the following six steps [27]:

1. Characterize and understand: the current project is categorized with respect to different characteristics to find a similar set of projects. This will help to identify a context in which it will be possible to reuse experiences, evaluate and compare the
2. new project in respect to the similar past ones, and use past projects for prediction. Different aspects can be used to categorize a project:
   - people factors (e.g., expertise, organization, problem experience, etc.),
   - process factors (e.g., life cycle model, methods, techniques, tools, etc.),
   - product factors (e.g., deliverables, required quality, etc.), and
   - resource factors (e.g., target and development machines, time, budget, legacy software, etc.)
3. Set goals: at this point the goals for the process execution have to be determined. This step formalizes all the aspects that are important (and that should be observed) during the project development. The goals reflect the goals of the organization and have to be measurable, i.e., they have to be specific enough that later we can compare the results to the goals. It would be pointless to have a goal such as "to have more beautiful user interfaces" without specifying what more beautiful means. After the development no one could objectively tell if and how much more beautiful the user interface now really is. This means that we cannot assess how much the means we used to obtain a more beautiful user interface contributed to the outcome: if they are helpful or not, how much they help, etc.

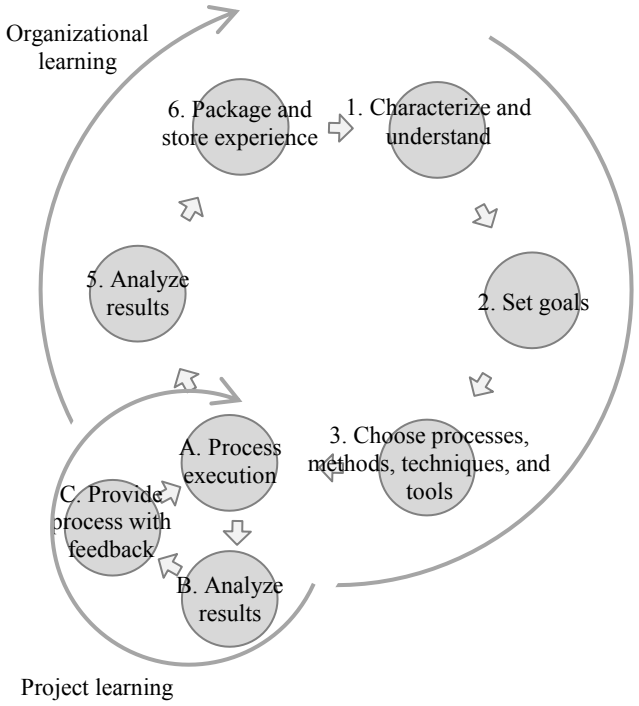

Figure 9: The Quality Improvement Paradigm (adapted from [27]).

4. Choose processes, methods, techniques, and tools: after settings the goals, the appropriate means (e.g., process models, methods, techniques, and tools) have to be selected. The word appropriate means that we aim to select and adjust means that are effective to obtain the stated goals within the identified context. Moreover, we aim to improve our ability in choosing the appropriate means to achieve our goals. For this reason it is necessary that what we choose is measurable in respect to the degree of how much it helps to achieve the stated goals.
5. Execute the process: in this step the development process prepared in the three steps before is executed and data about the execution is collected to provide the necessary input for step number 5. Software is development, not production; it is evolutionary and experimental. The essential difficulty of the project is at step 4: the process execution.
6. Analyse results: Based on the stated goals of step 2, the collected data is studied to improve our ability to perform step 3, to choose effective means to achieve our goals. This phase aims to improve our understanding of the results of our choices, to answer questions like "what is the impact of the environmental factor A on the effectiveness of technique B?" or "how does the development time change when technique X is used?"
7. Package and store experience: to profit from the gained experience of step 5, future projects have to be able to access and use it. The best way how this is done depends on the type of experience, some examples are: process models, lessons learned, check lists, source code, components, patterns, etc.

The aim of using of the Quality Improvement Paradigm is to collect knowledge about software development tactics. The idea to collect knowledge in a reusable form is not new: design patterns [29], process patterns [30], or organizational patterns [31] are some examples that show the success of such an approach.

The adoption of the Quality Improvement Paradigm or something similar requires a cultural change. Instead of climbing on the bandwagon of each hype, software developers (intended as both software engineers and software artists) need to become more scientific. Scientific means here that software developers need to collect experience in the form of tactics and strategies instead of commandments of gurus.

The Quality Improvement Paradigm is just one possibility. The message is that we have to create an nondenominational Agility, based on facts instead of myths. Unfortunately we cannot play Leonardo da Vinci and anatomize some guru and look what's inside. We have to start collecting knowledge (for example using automatic tools [40, 41]), understand what we are doing [32, 33, 34, 48] and break out from our role as Agile disciples.

Conclusions and future work

We observe a rising scepticism towards Agile Methods: practitioners are not able to use them in an efficient and effective way since the underlying assumptions are hidden by gurus. This missing knowledge is furthermore used by agile zealots to indeed legitimize hacker behaviour. The reputation of Agile is at stake.

To avoid that Agile Methods become obsolete, we think it is necessary to understand software development process tactics, i.e., knowledge about when to use which practice, knowing its advantages, its disadvantages, and how to apply it.

Therefore, we propose the use an approach to collect falsifiable knowledge about agile methods, for example using the Quality Improvement Paradigm.

To change this attitude, i.e., to switch from believing a guru to adopt a scientific approach requires educational, cultural, and managerial changes. To study how to accomplish these, will be the challenge of the future.

Acknowledgments

We thank Professor David West for his constructive comments that helped us to improve the content of this paper.

References

[1] Gartner Group: "The Gartner Hype Cycle", http://www.gartner.com/technology/research/methodologies/hype-cycle.jsp, accessed on April 13th, 2012.

[2] Jarvenpaa, H., Makinen, S.J.: "Empirically detecting the Hype Cycle with the life cycle indicators: An exploratory analysis of three technologies." In: Proceedings of the IEEE International Conference on Industrial Engineering and Engineering Management (IEEM) 2008, pp.12–16. IEEE Press, Singapore (2008).

[3] Google: "Google Trends" , http://www.google.de/trends, accessed on April 13th, 2012.

[4] Yahoo Groups: "extremeprogramming", http://tech.groups.yahoo.com/group/extremeprogramming, accessed on April 13th, 2012.

[5] Blechar, M.: Hype Cycle for Application Development. Garner Research (2010).

[6] Ceschi, M., Sillitti, A., Succi, G., De Panfilis, S.: "Project Management in Plan-Based and Agile Companies." In: IEEE Software, vol. 22, no. 3, pp. 21–27. IEEE Computer Society (2005).

[7] Fowler, M.: "Is Design Dead?" In: Succi, G. and Marchesi, M. (eds.) Extreme Programming Explained. Addison Wesley Longman (2001).

[8] Rakitin, S. R.: "Manifesto Elicits Cynicism." In: IEEE Computer, vol. 34, no. 12, p. 4. IEEE Computer Society (2001).

[9] Stafford, T., Webb M.: Mind hacks. O'Reilly Media (2004).

[10] Project Management Institute: A guide to the project management body of knowledge. Project Management Institute (2008).

[11] Vanderburg, G.: "A simple model of agile software processes – or – extreme programming annealed." In: Proceedings of the ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA) 2005. ACM (2005).

[12] Conboy, K., Fitzgerald, B.: "Organizational Dynamics of Technology-Based Innovation: Diversifying the Research Agenda." In: McMaster, T., Wastell, D., Femeley, E., DeGross, J. (eds.) IFIP International Federation for Information Processing, vol. 235. Springer (2007).

[13] McBreen, P.: Questioning Extreme Programming. Addison-Wesley (2003).

[14] Stephens M., Rosenberg. D.: Extreme Programming Refactored: The Case Against XP. Apress (2003).

[15] Abrahamsson, P., Salo, O., Ronkainen, J., Warsta, J.: Agile Software Development Methods: Review and Analysis. Technical Research Centre of Finland, Vtt Publications (2002).

[16] Boehm,B., Turner, R.: Balancing Agility and Discipline: A Guide for the Perplexed. Addison-Wesley (2004).

[17] Koch, A.: Agile Software Development: Evaluating the Methods for Your Organization. Artech House (2005).

[18] Brinkkemper, S.: "Method Engineering: Engineering of Information Systems Development Methods and Tools." In: Information and Software Technology, vol. 38, no. 4, pp. 275–280. Elsevier (1996).

[19] Cronholm, S., Goldkuhl, G.: "Meanings and Motivates of Method Customization in Case Environments: Observations and Categorizations from an Empirical Study." In: B. Theodoulidis (eds.). Proceedings of the Workshop on the next Generation of Case Tools (NGCT) 1994, pp. 67–79. University of Twente, Enschede, NL (1994).

[20] Grundy, J., Venable, J. : "Towards an Integrated Environment for Method Engineering." In Brinkkemper, S. Lyytinen, K. Welke, R. (eds.). Proceedings of the IFIP TC8 Working Conference on Method Engineering: Principles of Method Construction and Tool Support 1996, pp. 45–62. Chapman & Hall, London (1996).

[21] Harmesen, F., Brinkkemper, S., Oei, H.: "Situational Method Engineering for I.S. Project Approaches." In: Verrijn-Stuart A. A., OUe T. (eds.) Methods and Associated Tools for the Is Life Cycle, pp. 169–194. Elsevier Science (1994).

[22] Smolander, K., Tahvanainen, V., Lyytinen, K.: "How to Combine Tools and Methods in Practice: A Field Study." In: Steinholtz, B., Solvberg, A., Bergman L. (eds.) Lecture Notes in Computer Science, Proceedings of the Second Nordic Conference (CAiSE) 1990, pp. 195–214. Springer (1990).

[23] Tolvanen, J., and Lyytinen, K.: "Flexible Method Adaptation in Case: The Metamodeling Approach." In: Scandinavian Journal of Information Systems, vol. 5, no. 1, pp. 51–77. Information Systems Research in Scandinavia Association (1993).

[24] Beck, K., Andres, C.: Extreme Programming Explained: Embrace Change, 2nd ed. Addison-Wesley Professional (2004).

[25] Janes. A., Succi, G.: "To pull or not to pull, pp. 889–894." In: Arora, S., Leavens, G.T. (eds.) Companion to the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) 2009, ACM (2009).

[26] Florac, W.A., Carleton, A.D. Measuring the Software Process: Statistical Process Control for Software Process Improvement. Addison-Wesley Professional (1999).

[27] Basili, V.: "Quantitative Evaluation of Software Methodology." In: Proceedings of the First Pan Pacific Computer Conference, vol. 1, pp. 379-398 (1985).

[28] Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice. Addison-Wesley Longman (2003).

[29] Gamma, E., Helm, R., Johnson, R.E.: Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley Longman (1994).

[30] Ambler, S.: Process Patterns. Cambridge University Press (1998).

[31] Coplien, J.O.: Organizational Patterns of Agile Software Development. Prentice Hall (2004).

[32] Sillitti A., Succi, G., Vlasenko, J.: “Understanding the impact of pair programming on developers attention: a case study on a large industrial experimentation.” In: Proceedings of the International Conference on Software Engineering (ICSE 2012), pp. 1094-1101. IEEE Press (2012)

[33] Torchiano, M., Sillitti, A.: “TDD = too dumb developers? Implications of Test-Driven Development on maintainability and comprehension of software.” In: The 17th IEEE International Conference on Program Comprehension, pp. 280–282. IEEE Computer Society (2009).

[34] Fronza, I., Sillitti A., Succi, G.: “Does Pair Programming Increase Developers' Attention?” In: Proceedings of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering. ACM 2011

[35] Beck, K., Grenning, J., Martin, R. C., Beedle, M., Highsmith, J., Mellor, S., van Bennekum, A., Hunt, A., Schwaber, K., Cockburn, A., Jeffries, R., Sutherland, J., Cunningham, W., Kern, J., Thomas, D., Fowler, M., Marick, B.: “Manifesto for Agile Software Development”, http://agilemanifesto. org, accessed on April 13th, 2012.

[36] Gauch, Hugh G., Jr.: Scientific Method in Practice, Cambridge University Press (2003).

[37] Duggan, J., Stang, D. B., Iyengar, P., Murphy, T. E., Young, A., Norton, D., Driver, M., Kenney, L. F., James, G. A., Beyer, M. A., Schulte, R. W., Natis, Y. V., Gootzit, D., Karamouzis, F., Scardino, L., Blechar, M. J., Newman, D., Feiman, J., MacDonald, N., Feinberg, D., Valdes, R., Light, M., Cearley, D. W., McCoy, D. W., Thompson, J.: Hype Cycle for Application Development, 2007. Garner Research (2007).

[38] Il fatto quotidiano: “14.02.2012: Olimpiadi moderne, da Barcellona ’92a Pechino 2008 un affare solo sulla carta.”, available on http://www.ilfattoquotidiano.it/2012/02/14/olimpiadi-moderne-barcellona-pechino-2008-affare-solo-sulla-carta/191192/, accessed on July 30th 2012.

[39] Schneid, H.: “04.06.2012: Skylink: Der rot-schwarze Millionenskandal.” Die Presse, Wien Panorama, available on http://diepresse.com/home/panorama/wien/763175/Skylink_Der-rot schwarze-Millionen skandal, accessed on July 30th 2012.

[40] Sillitti, A., Janes, A., Succi, G., Vernazza, T.: “Monitoring the development process with Eclipse” In: Proceedings of the International Conference on Information Technology: Coding and Computing, pp. 133-134, vol. 2, no. 2. IEEE (2004).

[41] Sillitti, A., Janes, A., Succi, G., Vernazza, T.: “Measures for mobile users: an architecture” In: Journal of Systems Architecture, vol. 50, no. 7, pp. 393-405. Elsevier (2004).

[42] Naur, P., “Programming as theory building.” In: Microprocessing and Microprogramming, vol. 15, pp. 253-261. Elsevier (1985).

[43] Taylor, F.W.: The Principles of Scientific Management. Harper & Brothers (1911)

[44] Herzberg, F.: “One more time: how do you motivate employees?” In: Harvard Business Review (January-February), pp. 53-62. Harvard Press (1968).

[45] Boehm B.: “A Spiral Model of Software Development and Enhancement” In ACM SIGSOFT Software Engineering Notes, vol. 11, no. 4, pp. 14-24, ACM (1986)

[46] West, D.: Object Thinking. Microsoft Press (2004)

[47] Die Welt: “23.08.11: Kosten für Elbphilharmonie bei 500 Millionen,“ available on http://www.welt.de/regionales/hamburg/article13561526/Kosten-fuer-Elbphilharmonie-bei-500-Millionen. html, accessed on July 30th 2012.

[48] Fronza, I., Sillitti, A., Succi, G., Vlasenko, I., Terho, M.: “Failure Prediction based on Log Files Using Random Indexing and Support Vector Machines.” In: Journal of Systems and Software, Elsevier 2012