Making sense of agile methods

Bertrand Meyer

Politecnico di Milano and Innopolis University

Some ten years ago, I realized that I had been missing something big in software engineering. I had heard about Extreme Programming early thanks to a talk by Pete McBreen at a summer school in 1999 and another by Kent Beck himself at TOOLS USA in 2000. But I had not paid much attention to Scrum and, when I took a look, noticed two striking discrepancies in the state of agile methods.

The first discrepancy was between university software engineering courses, which back then (things have changed) often did not cover agility, and the buzz in industry, which was *only* about agility.

As I started going through the agile literature, the second discrepancy emerged: an amazing combination of the best and the worst ideas, plus much in-between. In many cases, faced with a new methodological approach, one can quickly deploy what in avionics is called Identification Friend or Foe: will it help or hurt? With agile, IFF fails. It did not help that the tone of most published discussions on agile methods (with a few exceptions, notably [1]) was adulatory. Sharing your passion for a novel approach is commendable, but not a reason to throw away your analytical skills. A precedent comes to mind: people including me who championed object-oriented programming a decade or so earlier may at times have let their enthusiasm show, but we did not fail to discuss cons along with pros.

The natural reaction was to apply a rule that often helps: when curious, teach a class; when bewildered, write a book. Thus was "*Agile: The Good, the Hype and the Ugly*" born [2]. My first goal was to provide a concise tutorial on agile methods, addressing a frequently heard request for a comprehensive, no-frills, news-not-editorial description of agile concepts. The second goal was analytical: offering an assessment of agile boasts.

These boasts are impressive. The title of a recent Scrum book by one of the authors of Scrum promises "Twice the Work in Half the Time" [3]. Wow! I'll take a productivity improvement of four any time. Another book [4] by both of the method's creators informs us: "*You have been ill served by the software industry for 40 years—not purposely, but inextricably. We want to restore the partnership.*" No less! (Was any software used in producing that statement?) There is a certain adolescent quality to the agile literature ("no one else understands!"), but ideas are not born in a vacuum, and I quickly realized that the agile movement was best understood as evolution rather than revolution. While you would not guess it from some of the agile proclamations, the software engineering community did not wait until the Agile Manifesto [5] to recognize the importance of change; every textbook emphasizes the role of the "soft" in "software". I can point to my own 1995 book *Object Success* [6], a presentation of object technology for managers, which advocated designing for change and stressed the preeminence of code over diagrams and documents. For these issues as well as for some of the other agile ideas — the importance of tests, the necessity of an iterative process — the agile contribution was not to invent concepts but to convince industry to adopt them.

In looking at matters such as the support for requirements change, I encountered yet another discrepancy: between good intentions and concrete advice. It is great for the Agile Manifesto to "welcome change", but producing changeable software is a technical issue, not a moral one. It is hard to reconcile such lofty goals with the deprecation of software techniques that actually support change: information hiding, deemed ineffective in [7], and design for extension and reuse, pooh-poohed in [8] (full citations in [2], sections 4.4.4 and 4.4.5).

There are more eyebrow-raising agile pronouncements, but we should not let them obscure the major contributions of the agile school. After all, marketing buzz goes only so far; developers and managers, mostly driven by pragmatic considerations, have not embraced agile ideas — more accurately, *some* agile ideas — without good reasons. (For my part, I would not have spent the better part of four years reading more or less the entire agile literature, practicing agile methods and qualifying as a proud Certified Scrum Master, if I thought the approach was worthless.) But one needs a constantly alert IFF to sort out the best and worst agile ideas.

The final chapter of [2] summarizes that analysis of "the good, the hype and the ugly", with "good" augmented by a subcategory of the truly "brilliant". Here are a few significant examples in each category.

Let us start with the "hype", which can also be called "the indifferent": ideas that have been oversold even though their impact is modest. An example is pair programming, the practice of developing software in groups of two people, one at the keyboard and one standing by, speaking out their thought processes to each other. The XP method (Extreme Programming) prescribes pair programming as the standard practice. One of the uses of pair programming is as a source of publications: you can measure and compare the outcome (development time, number of bugs) of two groups working on the same topic, one with pair programming and the other using traditional techniques. Such studies are relatively easy to conduct in a university environment, on student projects in a software engineering course. There are many of them, which do not show any significant advantage or disadvantage of pair programming against other techniques such as code inspection (see for example [9]). The truth is that pair programming is an interesting practice, to be used on occasion, typically for a tricky part of the development requiring competence from two different areas; but there is no reason to impose it as the sole mode of development. It is also easy to misuse by confusing it with mentoring, an entirely different idea.

Other examples of the "indifferent" kind of agile advice include the role of open spaces (while office layout deserves attention, some of the best software was developed in garages), self-organizing teams (different projects and different contexts will require different project management modes), and the charming invention of "planning poker".

More worrying are the agile recommendations that fall into "the Ugly". Perhaps the most damaging is the widespread rejection of "Big Upfront Everything": upfront requirements, upfront design; for a typical example, see [10]. The rationale is understandable: some projects spend too much time in general preliminary discussions, a phenomenon known as "*analysis paralysis*", and we should strive to start some actual coding early. It does not justify swinging the pendulum to the other extreme. No serious engineering process can skip an initial step of careful planning. It is good to put limits on it, but irresponsible to remove it. The project failures that I tend to see nowadays in my role as a consultant (or project rescuer) are often due to an application of this agile rule; we don't need no stinkin' requirements phase, we are agile, let's just produce user stories and implement them as we go. A sure way to disaster.

User stories themselves also fall into the Ugly category. More precisely, user stories are a good way to *validate* requirements, by making sure these requirements handle common-sense scenarios of user interaction. But they are not a sufficient basis to specify requirements. A user story describes one case; piling up case over case does not give you a specification. What it will give you, in fact, is a system that will handle the user stories — the exactly planned scenarios — and possibly nothing else. In [2] I cite at length the work of Pamela Zave from AT&T [11], who has for decades studied feature-based design of telecommunication systems. The problem with individual features is that they interact with each other. Such interactions, often subtle, doom any method that tells you to build a system by just implementing feature after feature. Everything will look fine until you suddenly discover that the next user story conflicts with previously implemented ones, and you have to go back and rethink everything. While no universally accepted solution exists to the question of how best to write requirements, object-oriented analysis, which looks past individual scenarios to uncover the underlying *data* abstractions (see chapter 27 of [12]) is a good start.

Rejections of upfront tasks and reliance on user stories for requirements are examples of the harmful advice that you will find proffered —sometimes in the same breath as completely reasonable ideas — in agile texts. As I come to the Good and the Brilliant it is useful to include an example of a technique that, depending on how you use it, qualifies as part of either the worst or the best. I mentioned that, at the beginning of the last decade, agile methods had found their way into industry but not academia. Students learn their trade not just from courses but also from summer internships in industry. In one of my first classes as a newly minted professor in 2002, I gave a lecture on software design; a third-year student came to me afterwards and asked why I was still teaching such nonsense: everyone knows, he said, that nowadays no one does design; we just produce "the simplest thing that can possibly work" and then *refactor*. I was stunned (not having realized how far XP ideas had percolated). He was wrong: no magic process can, through refactoring, turn bad design into good. Refactoring junk yields junk. Understood this way, refactoring would fall into the Ugly category. And yet refactoring also belongs to the Good by teaching us the lesson that we should never be content with a first software version simply because it works. Instead, we should apply the systematic habit of questioning our designs and looking into what could be done better. In other words, whatever my student thought, the right approach is to work hard, upfront, on producing a good design — *and* later on to refactor it.

Some other positive contributions of agile methods are for everyone to see, since agile ideas have already exerted a major influence on the practice of software development. Most visibly, no project in its right mind would today go into the scheme (which looks crazy, but not so long ago was the norm) of splitting the task into large chunks leading to separate subprojects, and trying to reconcile them months down the road. The splitting is easy; it is the reconciliation that can be a nightmare. Divergent assumptions, often implicit, preside over the design of the various components and propagate into the depths of each of them, rendering them incompatible. The only remedy is to catch such divergence right away. While iterative development is not an agile invention, agile made it the default, and particularly promoted the use of *short* iterations. ("*Short*" has evolved to mean increasingly shorter. Just a few years ago, promoting six-week sprints sounded audacious. Today we are hearing about one-week or sometimes one-day sprints. This is not even taking into account the spread of "DevOps" processes with their rapid fine-grain interleaving of development, testing and deployment.) Continuous integration and continuous testing are natural complements to this core idea. These and a number of other agile precepts are truly Good.

I mentioned that some of the Good deserves an upgrade to Brilliant. Let me bring up just two examples. Both are ideas that figure in the agile literature, although with less emphasis than others which, to me at least, are less significant. The first deserves a substantial discussion but I will just state it: no branching. Repeat after me: *branching is evil*. The second appears in Scrum texts but without a name; I call it the Closed-Window Rule. It states that the list of tasks for an iteration — a "sprint", which, as noted, is short — the list of tasks for that iteration cannot grow. It does not matter who requests an addition: queen, hero or laborer, everyone will be told "no". The proposed functionality will have to wait until the next sprint. There is an escape mechanism (an "exception" in programming language terms): if the addition is truly essential, you can cancel the sprint and start afresh. This possibility will address truly urgent cases, but is so extreme as to be used only rarely. The beauty of the Closed-Window Rule is that it brings stability to software projects, preventing the constant influx of supposedly good ideas that disrupt the development. Some of those may not look *so* good when you wake up sober the next morning; the Closed-Window Rule fosters a process of attrition and selection in which only the fittest ideas survive. These survivors will not have that long to wait: the rule would be unworkable with the long steps of old-style project development, but with a typical one-month sprint the average delay will be two weeks, during which the ideas will get the opportunity to mature. Few suggestions of added functionality are so critical that they cannot wait two weeks.

Benefitting from agile methods is a matter of spotting and rejecting the Ugly, ignoring the Hype, and taking advantage of the Good and Brilliant. Industry, which usually has its feet solidly in the ground, understood this situation early: despite the absolutist claims of some agile proponents (adopt every single one of my precepts, or else…), every project that I have seen embraces a subset of the chosen method's ideas, rejecting those that do not fit its culture or its needs. Agile methods are not a panacea, and, like most human endeavors, have their dark side, which has not prevented them from improving the practice of software development in very concrete ways. They do not, however, invalidate the knowledge of software engineering accumulated over the preceding decades. Some of their beneficial insights contradict specific elements of this traditional wisdom, but for the most part they complement and expand it.

Agile is not a negation of what came before. It is one more brick in the patient construction of the modern software engineering edifice.

References

1. Barry Boehm and Richard Turner: *Balancing Agility and Discipline: A Guide for the Perplexed*, Addison-Wesley, 2003.
2. Bertrand Meyer: *Agile: The Good, the Hype and the Ugly*, Springer, 2014. Also note the EdX online course (MOOC) "Agile Software Development" at www.edx.org/course/agile-software-development-ethx-asd-1x-0.
3. Jeff Sutherland: *Scrum: The Art of Doing Twice the Work in Half the Time, Random House*, 2015.
4. Ken Schwaber and Jeff Sutherland: *Software in 30 Days: How Agile Managers Beat the Odds, Delight Their Customers, And Leave Competitors In the Dust*, Wiley, 2012.
5. Agile Manifesto, at agilemanifesto.org.
6. Bertrand Meyer: *Object Success: A Manager's Guide to Object-orientation, Its Impact on the Corporation and Its Use for Reengineering the Software Process*, Prentice Hall, 1995.
7. Mary and Tom Poppendieck: *Leading Lean Software Development*, Addison-Wesley, 2010.

8. Ron Jeffries, Ann Anderson and Chet Hendrickson: *Extreme Programming Installed*, Addison-Wesley, 2001.
9. Matthias Müller: *Two controlled experiments concerning the comparison of pair programming to peer review*, in *Journal of Systems and Software* 78, 2005, pages 166-179.
10. Mike Cohn: Agile Design: Intentional Yet Emergent, blog article available at www.mountaingoatsoftware.com/blog/agile-design-intentional-yet-emergent.
11. Pamela Zave, collection of research papers on requirements and other topics, available at www.research.att.com/people/Zave_Pamela/custom/indexCustom.html.
12. Bertrand Meyer: *Object-Oriented Software Construction*, 2nd edition, Prentice Hall, 1997.