Methods need theory

Ivar Jacobson
Bertrand Meyer
Preprint of article published in Dr. Dobbs Journal, 6 August 2009. There is still a page at https://www.ddj.com/architect/219100242 but the text itself seems to have disappeared.

For someone in search of a software development method, the problem is not to find answers; it's to find out how good the proposed answers are. We have lots of methods — every year brings its new harvest — but the poor practitioner is left wondering why last year's recipe is not good enough after all, and why he or she has to embrace this year's buzz instead. Anyone looking for serious conceptual arguments has to break through the hype and find the precious few jewels of applicable wisdom.

In this note we argue that work on software development methodology must undergo a profound transformation. It should renounce its current reliance on fashion and political-style propaganda, turning instead to a serious scientific endeavor based on theory and experimental validation.

1. Fashion, politics or engineering?

Software methodology is a strange field. In principle it should be a science-based engineering discipline, but in practice it looks at times like the fashion industry, and at time like politics.

As in the fashion industry, every new year brings a new style, and in their hurry to keep up people throw away the good with the bad. Instead of learning from their own experience, they start with something they believe must be better since everyone else says it's better. Contrary to the complaint — common among innovators — that large companies are slow to embrace change, many are eager to try out new things; the true problem is that they abandon them even more quickly, discarding expensive investments in processes and tools before having had a chance to apply them seriously. Everything is marketed to the trend setters and early adopters. We accept this for clothing, but with the size of our software investment and its importance for society this is wasteful and irresponsible.

As in politics — more accurately, bad politics — the emphasis is not on substantial solutions to hard problems, but on slogans, propaganda and emotions. Ideas are not presented through careful discussions of pros and cons, but marketed like brands; they are spread by gurus delivering the Holy Word. Each approach tries to ignore its neighbors; if it has to acknowledge them, it is usually to berate them. The practitioner is left to wonder whom to believe.

In the end few new ideas ever get applied on a large scale, so that little changes in what counts: quality, productivity and lead time in the development of large systems. For all the hoopla in software methods over the past forty years, only a handful of major innovations — structured programming, object technology, design patterns, UML, … — have truly affected how the industry works.

These are signs of immaturity. The discipline needs to become adult.

2. Agility

The latest wave to sweep the industry is "agile". Agile methods have made a number of significant contributions and reminded us of the central role of people in software engineering. Some of the agile lessons are likely to remain in future methods.

At the same time, the agile field provides a vivid example of some of the phenomena cited above. For a movement that values people over process and tools, agility has given us many processes and tools advertised as “new”, without making the effort to describe what is truly different and what is a restatement of known ideas. Practitioners are understandably confused. Regardless of the value of the ideas, it is striking to see how they are promoted: the elegantly written foundational document of the approach is a “manifesto” (http://agilemanifesto.org/) , long on emotional appeals in the first person plural (“*We* have come to value…”) and short on facts. This style may be effective to capture attention, but as ideas mature it should yield to more traditional (even if also more boring) forms of argumentation.

In engineering and science, the originators of a new technique — eager, like any other human beings, to promote their invention — are careful to work out where its application is inadequate or unproven Few software methodologists bother with such intellectual safeguards. Too many exaggerate the differences with previous approaches. For one paradigm shift (such as object technology), how many supposed breakthroughs are adaptations of known ideas? There is nothing wrong with incremental improvements; much of the progress in science and engineering happens that way. But it makes no sense to disguise every increment as a revolution.

3. Your current method: a cocktail of practices

The method you are using for your development, commercial or locally developed, new or old, known or obscure — is really just a combination of elements from the methodology literature, spiced up with some domain- or business-specific extensions. The basic ingredients are individual *practices*.

If we separate the ingredients from the cocktail we can empower people to build the methods they need; such methods, being devised in a modular fashion, will be able to evolve quickly as a result of lessons learned and in response to the demands of a quickly changing industry.

4. Developing a theory

The economic pressure is such that current characteristics, — the fashion-industry-like versatility, the politics-like use of propaganda — will not completely go away. But methodologists who are concerned about the viability of software engineering should bring reason to the discipline and work in a different way.

What we are missing is the cornerstone of science and engineering: a theory, and its validation. Here are the steps that we can take to turn software methodology into a serious endeavor.

1. Model the nature of methods

   Software development is a human activity, but it consists of well-defined steps with well-understood relations between them. At least the definition and understanding exist implicitly, in the minds of experienced professionals. This is not enough; we need a strong theory of software development. Formal methods give us the right tools to perform this modeling; object-oriented languages with contracts can also serve that purpose. Without a precise, unambiguous model of the tasks and constraints of software development we cannot improve things significantly. The model should be method-independent (describing the problem, not the solution); and it should not only include definitions and axioms but — the part too often missing in formal models — theorems stating fundamental properties of all systems and all viable methods

2. Find the Kernel – the Mother of all Methods

All methods, beyond the hyped differences, share many properties. Using the theory as a basis, we should describe the properties that any method should satisfy if it is going to be effective for developing quality software. After all, they are all used to develop software, and they all acknowledge that there are certain things that we always need to do when we develop software. We always write code, we always test it in some way, we always think about requirements (documented or not), we always have a backlog (explicit or implicit), and we always have a plan on a paper or in our heads. Using the theory as a basis, we should describe the properties that any method should satisfy if it is going to be effective for developing quality software.

We need to find the “essential core” or “kernel” of software development, which cannot be made simpler. For example, by studying about 50 methods, including XP and Scrum, we have identified a kernel with some 15+ elements, things we always do or always produce.

3. Describe each interesting method using the kernel.

   With the kernel in place, all methods can be described and compared. We can harvest the implicit practices from all widely used and proven methods or processes, such as architecture, components, iterations, scrums etc. Some practices will overlap, e.g. use-case driven development and feature-driven development. Some practices will complement one another, e.g. use-case driven development and design-by-contract.

   The kernel clears away the cosmetic differences between methods, such as the same thing being called by different names in different methods. For example RUP talks about iterations whereas Scrum talks about sprints, but they mean pretty much the same thing. By doing this we can clearly see the real differences between different methods.

Since the kernel is agnostic in relation to any specific practice, we can simply find out what is the actual difference between different practices, not just on the surface but in depth. This decreases the element of religion in which every method is embedded.

5. Moving on

The above represents a clear call to action which we hope will be adopted (after any suitable adaptations) by all those who agree with us that the software industry needs to reach maturity.

Perhaps the most important step is to realize that what unites the different schools of thought is more important than what separates them. In particular:

- *Agile vs process*: the differences are exaggerated. The goals are the same on both sides: producing excellent software smoothly. All processes need agility, since in software more than anywhere else change is the rule and stability the exception. All agile methods, if applied to enterprise-critical projects, need a process, including specification and design.

- *Formal vs informal*: software practitioners must realize that any form of progress includes a dose of formal methods, and that they have nothing to fear from them. All engineering relies on math: can one imagine an electrical or mechanical engineer refusing to learn and apply mathematical techniques? Formal methods have their limits — no one claims that they solve everything — but are far beyond “academic” and have repeatedly proved their value when used appropriately. Whether we realize it or not, they are already pervasive in some areas (type checking as present in modern programming languages is a form of proof, and hardware design is increasingly based on mathematical techniques). As the IT industry moves towards a more professional mode of operation it will increasingly rely on selectively applied mathematical techniques.

Only by ignoring superficial differences and applying all ideas, regardless of their origin, shall we provide the industry with what it is entitled to expect from the experts: scientifically sound, practically useful methods and tools.

These are our views. Are we chasing windmills? Or is it possible to clean up the methodology mess and develop the sound basis that the industry needs? One thing is certain: progress can only result from the collaboration of many people. There are many forums to discuss these issues, starting with our blogs (http://ivarblog.com/, http://bertrandmeyer.com). Tell what you think, and help software engineering move on to the adult phase.