Methodology Work Is Ontology Work

Brian Marick, marick@visibleworkings.com

I argue that a successful switch from one methodology to another requires a switch from one ontology *to another. Large-scale adoption of a new methodology means "infecting" people with new ideas about what sorts of things there are in the (software development) world and how those things hang together. The paper ends with some suggestions to methodology creators about how to design methodologies that encourage the needed "gestalt switch".*

In this paper, I'm going to abuse the word "ontology". In philosophy, an ontology is an inventory of the kinds of things that actually exist, and (often) of the kinds of relations that can exist between those things. My abuse is that I want ontology to be *active*, to drive people's actions. I'm particularly interested in *unreflective actions*, actions people take because they are the obvious thing to do in a situation, given the way the world is.

Here are some examples of ontologies.

Example 1: Emerson

Cornel West (1989) attributes a particular worldview to Ralph Waldo Emerson in his (West's) *The American Evasion of Philosophy*.

1. Emerson held that "the basic nature of things, the fundamental way the world is, is itself incomplete and in flux" (p. 15). Moreover, the world and humans are bound up together: the world is the result of the work of people, and it actively solicits "the experimental makings, workings, and doings of human beings" (p. 15).
2. Emerson believed that this basic nature makes the world joyous. It gives people an opportunity to exercise their native powers with success, because the world is fundamentally supportive of human striving.
3. And finally, Emerson believed that human powers haven't yet been fully unleashed, but they can be through the "genius of individuals willing to rely on and trust themselves" (p. 16).

I do not believe this passage means that Emerson was an optimistic guy and thought that things would pretty much work out in the end. I believe that, to him, it was a **fact** that the world beckons to humans to change it, and that the world just naturally fits together with individuals in a way that makes such change work, once individuals wiggle their way into the right relationship with it. A person who believes that will behave differently than someone who believes the world conspires against us. He'll behave differently than a strict materialist, someone who believes the world is fundamentally indifferent (as do, probably, most programmers).

Example 2: cows

My wife Dawn teaches veterinary students how to cure cows. Each sick cow is assigned a student, and each day that student has to decide—among other things—whether the cow is "bright" or "dull." Students learn the difference through exposure to a series of examples where they make judgments that Dawn then corrects. At the end of the process, students can reliably judge between bright and dull, though they cannot articulate any definition of the terms. In fact, the notion of defining them seems somewhat beside the point. Cows simply *are* either bright or dull, the way the student herself is either alert or sleepy, or the way a joke is either funny or lame. Any explanation of how she knows seems contrived and after the fact. It's as if the student's perceptual world has expanded. For purposes of this paper, I'd say there's a new kind of thing in her ontology: brightness. Moreover, the category has an effect in the world: a trained veterinarian is unable to ignore her perceptions, unable *not* to use them in her diagnostic work.

Example 3: programmers

> So, I don't start with a story like "The game has Squares." I start with something like: "Player can place a piece on a square."[...]
>
> What I am not doing is worrying about overall game design. [...] [Ideally], I let the design emerge.
>
> -- William Caputo[1]
>
> Beck has those rules for properly-factored code: 1) runs all the tests, 2) contains no duplication, 3) expresses every idea you want to express, 4) minimal number of classes and methods. When you work with these rules, you pay attention *only* to micro-design matters.
>
> When I used to watch Beck do this, I was *sure* he was really doing macro design "in his head" and just not talking about it, because you can see the design taking shape, but he never seems to be doing anything directed to the design. So I started trying it. What I experience is that I am never doing anything directed to macro design or architecture: just making small changes, removing duplication, improving the expressiveness of little patches of code. Yet the overall design of the system improves. I swear I'm not doing it.
>
> -- Ron Jeffries[2]

Whereas Emerson might have believed that design comes through the "genius of individuals willing to rely on and trust themselves," these two notable programmers believe designs come through the actions of not-necessarily-geniuses willing to rely on and trust a particular discipline, specifically what's nowadays called test-driven design and refactoring (Beck 2002, Astels 2003, Fowler 1999, Wake 2004).

---

[1] testdrivendevelopment Yahoogroups mailing list, March 9 2003.

[2] Agile Manifesto authors' mailing list, July 19 2001.

However, I claim that Caputo and Jeffries do share much of a worldview with Emerson. In their world, the basic nature of a program is to be incomplete and flux, to solicit the workings of human beings. Given the right practices, social organizations, workspace arrangement, and tools, the software will be fundamentally supportive of human striving. Software has it in its nature to be *soft*, if only we know the right way to accommodate that nature.

They are subscribers to the Agile ontology. That's distinct from the standard "software engineering" ontology, which is much more dour. In it, entropy is a fundamental thing in the world. Software development is a struggle against entropy that will inevitably fail as the product devolves into Foote and Yoder's "big ball of mud" (2000).

Whether the Agile ontology is *true* or not is not at issue in this paper. What I'm concerned with is how people are moved from one ontology to the other. In a personal communication, Caputo says that he believes his worldview was always compatible with the Agile methods. But many others didn't start that way. They became enthusiastic adopters of the Agile methods despite starting somewhere else. How did that happen? How can promoters and designers of future methodologies make it happen again?

Part of the answer must come from a detailed history of the Agile methods, which does not yet exist. In the meantime, I want to draw analogies with another field, one that also undergoes sweeping changes, one that most software people feel is admirable and worthy of emulation. I mean science.

Kuhn (1962) popularized the idea that science is not a matter of steady progress. Instead, its history contains major disruptions that he calls revolutions. Both Kuhn and Feyerabend (1975) emphasize that scientists on either side of a revolutionary divide cannot communicate. The technical term for that is *incommensurability*.[3] For example, Galileo and the Aristotelians he debated meant different things by velocity. In retrospect, we can gloss the difference as between instantaneous and average velocity, but that doesn't capture the understandings of the participants. To the Aristotelians, velocity - movement, change - was intimately bound up with a whole range of topics. The same theory that accounts for the falling of a ball and the movement of a pendulum should also account for the rising of flames and the growth of trees.[4] Galileo discards many of those topics - he says nothing about fire - thereby altering a whole network of meanings. There is no accessible shared meaning that allows either party to express their theory in the other party's terms.

The same is true of methodologies. I am the moderator of the agile-testing Yahoogroups mailing list. It's an enormously frustrating experience. To oversimplify, there are two factions. One is the "conventional" testers, to whom testing is essentially about finding bugs. The other is the Agile programmers, to whom testing is essentially not about bugs.

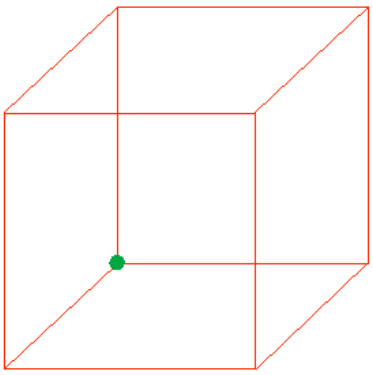
---

[3] See chapter 5 of Hacking (1983) for a nice description of different kinds of incommensurability.

[4] To an Aristotelian, objects try to reach their natural position. For a ball or a pendulum bob, that's down. For fire or a growing tree, it's up. To an Aristotelian, pendulums are much less interesting than balls, because a pendulum bob is a "tortured ball" that's prevented from falling in the natural way. To Galileo, a pendulum was a revealing case; to an Aristotelian, it obscures the essential. (This may be an example of what Pickering (1995) calls *machinic incommensurability*.)

Rather, it's about providing examples that guide the programming process. Whenever this point is made, it slips away. Whenever one definition is granted for the sake of discussion, the dispute sneaks back in through its implicit connection to other definitions and other issues. This is an ontology conflict, and conversations are not a particularly productive way of resolving such.

What is, then? Kuhn gives some help. He likens revolutionary theory changes to gestalt shifts of perception, such as the way a Necker cube suddenly switches from one orientation to another. Kuhn also talks of how the perception of a new "paradigm" is instilled in students in a way reminiscent of the way my wife instills a perception of "bright cows" in her students: they are drilled in example after variant example, comparing each new example to others, especially to the exemplars found in textbooks. The paradigm is acquired through practice, and new students enter the field knowing it. The old paradigm dies off with its adherents. However, this is perhaps not so helpful in the software field, where the textbooks seem to get written after the methodology is already established.



I think another student of science, Imre Lakatos, gives more help than Kuhn does. He devised his "methodology of scientific research programmes" (1978) to answer the question of when it is *rational* for a scientist to pursue a new research program. He was seeking a middle ground between those he viewed as irrationalists because they provided no solid rules for the advancement of science (Kuhn, Feyerabend), and rationalists like Popper (1934) who provided solid, simple rules whose only disadvantage is that they don't work and can't work.[5] He aimed to do a "rational reconstruction" of the history of scientific change and extract rules of rationality that would work.

To my mind, he failed. But the failure was of a productive sort. His rules aren't *sufficient*, but they're *convincing*. That is, I think they describe a process by which scientists come to switch paradigms (or ontologies), even if that switch is, at bottom, as irrational as it is rational. In this paper, I'm going to assume that some of these same rules will also convince software developers. (I'm using here only the most obviously applicable of Lakatos's rules.)

Lakatos on attracting ontology change

Lakatos claims that at the heart of any scientific research programme is a "**hard core** of two, three, or maximum five postulates" (Motterlini 1993, p. 103). For example, Newton's theory has a hard core of three laws of dynamics plus a law of gravitation.

[5] Popper holds that a rational scientist puts forth a testable hypothesis. If an experiment refutes it, the rational scientist abandons it. No hypothesis is ever proven; at best, it's tentatively accepted because it's not (yet) been refuted. Lakatos points out that, by this rule, Newton was a lousy scientist, since his laws of motion were easily refuted. See Laktos's "lectures on scientific method" in Motterlini 1999. And also see below.

The same might be true of successful methodologies. The Manifesto for Agile Software Development (2001) has four core values. Extreme Programming (Beck 2000) has the same number.

I don't know why a limited number is helpful, though I can speculate that they're easier to remember. They also fit our bias toward simplicity, elegance, and getting a lot of emergent payback from a small conceptual investment.

The lesson to methodologists is that the core postulates should be developed with care. Most important, they should be *developed* and *expressed*, not left implicit (as they so often are).

Lakatos also concentrates on **novel, striking success**. Scientific "wins" are not just a matter of avoiding experimental refutation. What convinced holdouts of Newton's theory of gravitation? According to Lakatos, it was Edmund Halley's successful prediction (to within a minute) of the return date of the comet that now bears his name. What tipped scientific opinion toward Einstein's theory of general relativity? The famous experiment in which the bending of light was observed during a solar eclipse.

What's the equivalent in software development? To grab mindshare, methodologists should use a surprising method to achieve unexpected success with a long-intractable problem. Consider "requirements churn." The traditional solutions revolve around eliminating the churn or planning an infrastructure tuned to likely changes. The Agile methods strikingly invert the dynamic by treating churn as good rather than bad: they assert that a team and a product can be "trained" into accepting changing requirements by making them accept changing requirements from the very beginning. I've met people - programmers and product owners - for whom the result was as striking as pointing a telescope where Halley said to and seeing a comet. They've bought into the programme; they want to use Agile methods on other projects, just as excited scientists wanted to use general relativity in all sorts of new places.

Next, Laktos says a scientific research programme must be **progressive**. It should continue making successful predictions that follow from elaborations of its hard core. Newton's theory went from success to success. General relativity didn't stop being useful after Eddington's 1919 observation of the eclipse.

Similarly, software methodologies should continually throw up new ways of working. For example, the XP practice of having programmers write tests before modifying a class is now being extended. People are writing whole-product tests (examples of requirements) before starting the coding. That's a straightforward extension. What's more interesting is how this extension is tying into other practices. These tests are being used as a way of improving and reinforcing the face-to-face communication that drives an Agile team. (Tests give the team something to talk *about*.) The inevitable difficulties in implementing tests that both "speak the customer's language" and also drive the product are best solved by blurring the boundaries between "testing" and "programming", reinforcing both the XP emphasis on "Whole Team" (Jeffries 2001) and also the general Agile tendency to favor generalists over specialists.

Continuing extension - continuing greater success - attracts people. Everyone likes a winner. More importantly (for this paper), it reinforces ontology. When people on a

project have a problem and the coach says "Whole team, guys. It's not his problem, it's our problem" and the team thereupon makes progress, it makes the concept "Whole Team" more concrete, more real, more something that's used reflexively. Moreover, problem-solving that depends upon different pieces of the ontology reinforces the way it all hangs together.

Scientific programmes also progress by **resolutely ignoring counterexamples**. For example, Newton did not discard his theory when he found it did not correctly predict the observed motion of the moon. When Le Verrier discovered the motion of Mercury's perihelion was faster than predicted by Newton, people shrugged and waited for Einstein to explain it.

Research programmes can even proceed despite their obvious falsity. Rutherford's model of the atom (mostly empty space, electrons orbiting a nucleus) violated Maxwell's equations, which were believed to be rock solid. They were certainly much more compelling than the new data Rutherford's model was intended to explain. But Rutherford's programme essentially said, "We'll figure out how to reconcile with Maxwell later." (The solution was quantized orbits - the "Bohr atom".) Lakatos says, "Theories grow in a sea of anomalies, and counterexamples are merrily ignored." (Motterlini, p. 99). This ignoring is essential, since a replacement theory cannot, especially in the beginning, hope to predict everything that a long-established theory can.

For this reason, a methodology should begin with a restricted scope, and its developers should not worry that they don't have an answer for every problem that rival methodologies claim to solve. They should not engage the rivals on those grounds. Rather, they should explicitly put them off for the future. In the now, they should concentrate on striking achievements and progressive expansion of the methodology's scope of applicability.[6]

The preface to *Extreme Programming Explained* does the right thing: "XP is designed to work with projects that can be built by teams of two to ten programmers, that aren't sharply constrained by the existing computing environment, and where a reasonable job of executing tests can be done in a fraction of a day." (Beck 2000, p. xviii)

Not only does eschewing universal claims let the methodologist and methodology users define success, rather than having it defined for them, it also gives scope for making the methodology progressive. Early work in Agile methods focused on completely co-located teams. More recently, work has been done on remote teams, but in an Agile context that emphasizes the roles of trust-building and constant, fluid conversation (Sepulveda 2003; Hunt, Thomas, and Sepulveda 2004).

Bringing in perception

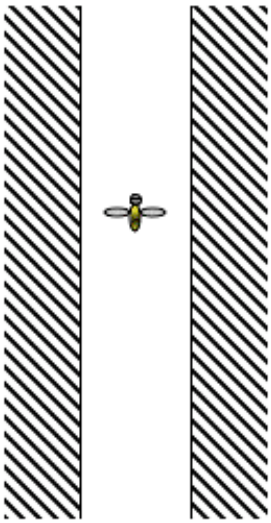
Lakatos, like Popper and many other students of science, preferred theorists to experimentalists. His methodology of scientific research programmes is all about the relationship of theories to one another. Experiment merely serves to provide

[6] Compare also to Gabriel's "Worse is Better": "The concept known as 'worse is better' holds that in software making (and perhaps in other arenas as well) it is better to start with a minimal creation and grow it as needed." (Gabriel web)

confirmations or to throw up anomalies to be dealt with whenever the research programme gets around to it. Experiment is not creative: the action takes place in the world of thought, not the physical world.

Later strands of science studies have placed much more emphasis on scientific practice, and they have much to offer.[7] Here, however, I'll step even further from rationality and look at perception.

Suppose you're a bee navigating down a tunnel. You don't want to crash into either side. How can you do it? Well, consider what happens as you drift toward the right side. Features on the right wall will appear to go by you faster, features on the left slower. So just have that specific perception trigger changes in your wing flapping that shift you to the left[8]. You don't need a "world model" with any accuracy; rather, to be a successful bee, you need a diverse set of perceptions that are well tuned to the tasks you, as a bee, need to perform. You are, in fact, more successful if the perception bypasses the brain and simply causes you to do the right thing.[9]



Methodologists should be alert for opportunities to align perception with ontology-building. Consider, for example, FitNesse, a wiki used to organize and execute Fit tests.[10] Some teams put a machine in the project bullpen with the FitNesse pages always visible. This encourages random people to wander past, glance at the current status of the acceptance tests, even run a few themselves. This is a variant form of Cockburn's (2001) information radiators or Jeffries' (2004) Big Visible Charts. It reinforces various parts of one Agile ontology: that feedback is an essential part of the world, a part that should be as quick and continuous as possible; that tests are for communication between programmers and business experts; and that communal workspaces generate learning. The team members *see* feedback all around them, the same way my wife sees bright cows.

Moreover, the constant perception of test status short-circuits rationality—and rationalizing. If the rule is that tests always pass, the sight of red on the test-display machine triggers an immediate impulse to fix the problem. That perception is stronger than a mere recitation of fact: "a test is failing." Perceptual patterns have the same (perhaps greater) effect in the absence of a specific rule. In a project that's tracking the number of passed vs. failed tests, the chart on the next page (Jeffries 2004) should help produce a visceral reaction and desire to make the badness go away. (If you're reading a

[7] Pickering 1992, Latour 1988, or Latour and Woolgar 1986 are good sources.

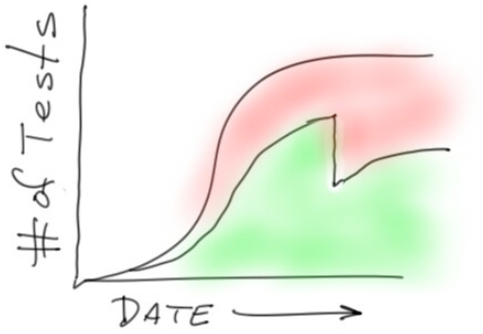
[8] Srinivasan et al. 1991. See also http://cvs.anu.edu.au/insect/

[9] Philip Agre (1997) writes well on the deficiencies of world models.

[10] For FitNess, see http://www.fitnesse.org; for wiki see Leuf 2001 or Rupley 2003; for Fit, see http://fit.c2.com.

black-and-white copy, the top band is red and the bottom green.) That might counter some of the usual "we'll have to fix that someday, but… not today" rationalization.

Ontologies are normally what Heidegger called *ready-to-hand*.[11] Just as one doesn't think about how to hold a hammer when pounding nails, one shouldn't have to think about the methodology, its ontology, and its rules during the normal pace of a project: one should simply act appropriately. These sorts of intellectual reflexes don't just happen. They are trained in, and it's part of the methodologist's job to arrange perceptions so that they reinforce the methodology.[12]



My position

Methodologies do not succeed because they are aligned with some platonic Right Way to build software. Methodologies succeed because people *make* them succeed. People begin with an ontology—a theory of the world of software—and build tools, techniques, social relations, habits, arrangements of the physical world, *and* revised ontologies that all hang together. In this methodology-building loop, I believe ontology is critical. Find the right ontology and the loop becomes progressive. I hope that this paper helps methodologists see better how to create and transmit ontologies.

References

Agre, Philip (1997)
Cognition and Human Experience.
Astels, David (2003)
Test-Driven Development: A Practical Guide.
Beck, Kent (2000)
Extreme Programming Explained: Embrace Change.
— (2002)
Test-Driven Development: By Example.
Cockburn, Alistair (2001)
Agile Software Development.
Dreyfus, Hubert (1991)
Being-in-the-World: A Commentary on Heidegger's Being and Time, Division 1.
Feyerabend, Paul (1975)
Against Method.
Foote, Brian and Joseph Yoder (2000)
"Big Ball of Mud", in *Pattern Languages of Program Design 4*, ed. Harrison, Foote, and Rohnert. Also http://www.laputan.org/mud/mud.html (accessed May 2004).
Fowler, Martin (1999)
Refactoring: Improving the Design of Existing Code.

[11] Heidegger is notoriously opaque. Winograd and Flores (1987) apply his ideas to software design. Dreyfus (1991) is an extended commentary. It's not *as* opaque as the little Heidegger I've struggled through.

[12] This is not to say that reflex should replace thought, just that thought has its place—which is not every place. When there's a *breakdown* (another Heideggerian term), the methodology (including its underlying ontology) should become *present-to-hand*, subject to reflection and change. But getting people across the gestalt shift barrier seems to me harder than getting them to debate methodologies. (To put it mildly.)

Gabriel, Richard P. (web)
"Worse is Better", http://www.dreamsongs.com/WorseIsBetter.html (accessed July 2004).
Hacking, Ian (1983)
Representing and Intervening: Introductory Topics in the Philosophy of Natural Science.
Hunt, Andrew, Dave Thomas, and Christian Sepulveda (2004)
"Remote Control", *Better Software Magazine,* Vol. 6, no. 4, April 2004.
Jeffries, Ron (2001)
"What is Extreme Programming" http://www.xprogramming.com/xpmag/whatisxp.htm (accessed May 2004).
— (2004)
"Big Visible Charts" http://www.xprogramming.com/xpmag/BigVisibleCharts.htm (accessed May 2004).
Kuhn, Thomas (1962)
The Structure of Scientific Revolutions.
Lakatos, Imre (1978)
The Methodology of Scientific Research Programmes. Philosophical Papers, Volume 1.
Latour, Bruno (1988)
Science in Action: How to Follow Scientists and Engineers Through Society.
— and Steve Woolgar (1986)
Laboratory Life: The Construction of Scientific Facts (2/e).
Leuf, Bo and Ward Cunningham (2001)
The Wiki Way: Collaboration and Sharing on the Internet.
Manifesto (2001)
"The Manifesto for Agile Software Development", http://www.agilemanifesto.org (accessed July 2004).
Motterlini, Matteo ed. (1999)
For and Against Method.
Pickering, Andrew ed. (1992)
Science as Practice and Culture.
— (1995)
The Mangle of Practice: Time, Agency, and Science.
Popper, Karl (1934)
The Logic of Scientific Discovery.
Rupley, Sebastian (2003)
"What's a Wiki?" http://www.pcmag.com/article2/0,4149,1071705,00.asp (accessed May 2004).
Sepulveda, Christian (2003)
"Agile Development and Remote Teams: Learning to Love the Phone". In *Proceedings of Agile Development Conference 2003*. http://www.christiansepulveda.com/papers/remote_agile_adc2003.pdf (accessed May 2004).
Srinivasan, M.V., M. Lehrer, W.H. Kirchner and S.W. Zhang (1991)
Range perception through apparent image speed in freely-flying honeybees. Vis. Neurosci. 6, 519-535.
Wake, William (2004)
Refactoring Workbook.
West, Cornel (1989)
The American Evasion of Philosophy: A Genealogy of Pragmatism.
Winograd, Terry and Fernando Flores (1987)
Understanding Computers and Cognition: A New Foundation for Design.