A Manglish Way of Working: Agile Software Development

DRAFT of Nov 23, 2004

Brian Marick (marick@exampler.com)

In the early years of this century, a style of software development dubbed "Agile" moved from an underground practice to one sufficiently respectable to be written up in the mainstream business press. From my perspective as an insider in that movement and a dilettante in science studies, I claim that the Agile style of work is readily and satisfyingly described by the terminology of The Mangle of Practice. *But that's not the main point of this chapter. The reason I am an Agile advocate is that the manglish style of work just suits certain people. Agile projects allow those people to be happy at work instead of bitter, cynical, and discouraged. Quite likely, others would like to work manglishly. I hope they will benefit from learning how it is we software developers get away with it.*

Let me distinguish between a *manglish story* and a *manglish person*. A manglish story is one that highlights a trajectory in time during which a prolonged interplay of resistance, accommodation, and chance leads to results that could not reasonably have been predicted from the story's initial conditions. It is, moreover, one in which everything is up for revision during that trajectory.

A manglish person is one who likes things that way.

Methodologists are people who tell others how software should be developed. Most of them are not manglish. Consider Parnas and Clements' widely cited "A rational design process: How and why to fake it" (1986). They advocate removing the historical trajectory from the writeup of a system's design. Instead, one should present it as if "we derive[d] our programs from a statement of requirements in the same sense that theorems are derived from axioms in a published proof."[1] One such derivation would contain these documents:

1. A list of requirements the system must meet. This is a complete set of truth-valued statements. If they are all true, the system has, by definition, solved the problem that prompted its creation.
2. A specification of the system's interface. This specification describes everything that anyone with minimal qualifications would need to know to predict how the

[1] This is reminiscent of Lakatos' "rational reconstructions" of scientific history in his *Methodology of Scientific Research Programmes* (1978): one teases out the rational, logical core of what happened. That's all that really matters.

system will behave given an arbitrary set of inputs. It also satisfies all the requirements.

3. An abstract design of the system's internals (often called the "architectural design"). This design, when realized, implements the specification.
4. Some number of less abstract designs. Each one realizes the one before it.
5. An argument that the actual code (instructions to the computer) realizes the least abstract level of design and thus, transitively, satisfies the requirements.

Until recently, the standard texts asserted that completing steps like these in sequence was the ideal way to develop software.[2] They do concede that people err. They'll overlook requirements. They'll write requirements ambiguously, causing later writers to produce the wrong design. They'll think a design implements the specification when it does not. And so on. Practitioners should take failure into account, mainly by putting in place measures to detect and correct it as soon as possible. Parnas and Clements take special care in the correction. Suppose a requirement was ambiguous. They will revisit the requirements document and rewrite it to remove the ambiguity. They will then propagate the consequences of that change to all derived documents. The error can now be forgotten, and the documents read as they would have in a perfect world (unless errors remain to be found).

Effaced history; no mention of emergence; no agency except that of the designer: not manglish.

Agile projects run differently.[3] The programmers are not guided by documents, but by a person who knows a *domain* like bond trading or nursing well. I call this person the *product owner*.[4]

The project progresses through a series of *iterations*, each from one week to one month long. At the beginning of each iteration, the product owner tells the team what features she wants delivered at the end. The team immediately begins writing code to implement those features, asking questions of the product owner along the way. Design consists of conversations, moving 3x5 cards around on tables (Beck and Cunningham 1989), and scribbling on whiteboards. Few, if any, design decisions are recorded, except in the code. At the end of the iteration, the team delivers a *potentially shippable* product containing

---

[2] For more, read Pressman, *Software Engineering: A Practitioner's Approach* (2004) or Sommerville, *Software Engineering* (2000). Note that older editions show the conventional ideal in purer form. The newer ones contain ever increasing admixtures of unconventional and Agile thinking.

[3] For the guiding values and principles of Agile development, see the appendix. For more on Agile methods in general, see Cockburn, *Agile Software Development* (2001). The general approach has spawned many particular refinements. The two most documented (and probably most common) are Extreme Programming and Scrum. For Extreme Programming, see Beck, *Extreme Programming Explained: Embrace Change* (2000) and Jeffries et al, *Extreme Programming Installed* (2000). For Scrum, see Schwaber and Beedle, *Agile Software Development with Scrum* (2001) and Schwaber, *Agile Project Management with Scrum* (2004).

[4] If you read the literature, you'll see "Customer" more often than "product owner". That's confusing. In an Agile project producing shrinkwrap software (the kind you buy in stores), the Customer is *not* the customer - that is, not the person who pays money for the software. That's why I use "product owner".

the new features. The product owner could conceivably stop the project at the end of any iteration, with no advance warning, and ship the product to end users.[5]

According to conventional methodologists, such a project is doomed. After the first iteration, the team will have code that supports only the first set of features. When they start the next iteration - with features they hadn't anticipated in their (nonexistent) design documents - they'll find the new features hard to wedge into the existing code. They'll be able to do it, but they'll inevitably leave some of the code slightly worse than before - making iteration three harder yet. Over time, the product will decay into what Foote and Yoder (2000) call "a big ball of mud". They'll be able to add new features only after heroic and lengthy effort. A conventional methodologist would describe this project as like the hare in the fable: they'll look productive at first, but a properly run project will - tortoise-like - win the race.

Agile methodologists, in contrast, believe that decay is not inevitable. It is possible to make software truly *soft*, but you can't do it by designing for predicted changes. Instead, you should treat each change - predicted or not - as a prod to work the software into a form that is more accommodating of change, with a design tuned to the kind of changes that have actually been called for. Acting on supposed knowledge of the future is more a hindrance than a help.[6]

Even though the design is created opportunistically, not "top down" (in a logical progression from requirements to code), that's not to say that a good design happens without effort or skill. Agile programmers work with certain rules that are believed to lead emergently ("bottom-up") to a good system design. As one programmer puts it:

> 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

[5] I've known of two cases where the project was in fact stopped right in the middle, with very little warning, and the product shipped. Many products do need a little time at the end to put things in final order. For example, screen images have to be collected and put in manuals.

[6] In practice the team will usually know what features are coming in the next iteration and also have some idea of the features coming after that, but they won't intentionally design for them, and they may not even know them. One agile programmer (who wished to remain anonymous) wrote me that "our customer seems [to] make significant changes in direction every week. Yet, it seems that from his point of view, he is steering us toward a coherent product. So why can't the project team share in his point of view? Because he refuses to put together a release plan [a list of roughly what features will be added to the product before it's shipped to users]. He gives us an iteration worth of work (one week) at a time." I asked him if the lack of a release plan had an effect on their day-to-day work. He replied, "The source of frustration is that the team lacks perspective and a vision of the finish line. That said, the lack of release plan does not have a direct effect on our everyday work." I should note here that not everyone agrees that you should anticipate *nothing*. Many believe that certain changes are best planned for. What makes those people still Agile is their commitment to making that set as small as possible.

> 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.[7]

Agile programmers also rely a great deal on constant communication. Teams typically work in bullpens rather than offices or cubicles so that there are no barriers to asking questions or sharing information. Most have daily meetings intended to tell each other what they did yesterday, what they plan to do today, and what help they need.[8] It is also common for programmers to program in pairs.[9] Each programmer might rotate through all parts of the system, pairing with each other programmer, thus gaining a generalist's knowledge of the whole rather than a specialist's knowledge of a part.

Finally, Agility requires a certain amount of self-discipline and courage.[10] There are almost always several ways of implementing a new feature, and the easiest ones usually lead to code decay and, if taken often, to the Big Ball of Mud. Discipline is required to avoid the easy path and make the code better than you found it. (The close communication and lack of specialist "owners" of parts of the system produce peer pressure that reinforces discipline.) The business will always want the team to produce more, faster, but teams have a certain sustainable velocity; if they try to go faster, they will degrade the code and start going slower (just as the conventional methodologists predict). It takes courage to resist pressure.

So: it's hard to be Agile. It's as easy to err as in conventional project, but the attitude toward that is quite different. Error leads to correction; correction is a change; and any change is a chance for something novel to emerge. I'll illustrate that attitude with a story from Ward Cunningham, one of Agile software development's founding figures. It's the story of Advancers.[11]

Cunningham's team was working on a bond trading application called WyCash. It was to have two advantages over its competition. First, it would be more pleasant to work with. Second, users would be able to generate reports on a position (a collection of holdings) as of any date.

As the team worked on features requiring them to track financial positions over time, some code got messier and messier and harder to work with. Their velocity decreased, and they created more bugs.

---

[7] Ron Jeffries, Agile Manifesto authors' mailing list, July 19 2001.

[8] The "daily standup meeting" is best described in the Scrum literature: Schwaber and Beedle (2001) and Schwaber (2004). It is also widely used in Extreme Programming (Beck 2000).

[9] Pair programming is most common in Extreme Programming. See also Williams and Kessler, *Pair Programming Illuminated* (2002).

[10] Discipline and courage are most emphasized in XP and Scrum, least emphasized in Crystal (Cockburn 2004).

[11] I base this story on conversations with Ward in 2003 and 2004. But see also http://c2.com/cgi/wiki?WhatIsAnAdvancer (accessed November 2, 2004).

Much of the problem was due to a particular *method*. You can think of a method as an imperative sentence in the formal language used to construct a program. In a bond trading application, there might be methods like "buy bond" or "alert the user when the price reaches *x*." The subject of the sentence is a named bundle of data called an *object*. So a complete instruction in a program might be "Hey! You, position! Add this bond to yourself." Before a method can be used, it has to be defined in terms of other methods. So "buy bond" might be defined as something like "Bond, what's your price? Money market account, remove that number of euros from yourself. Brokerage, add this number of euros to yourself, noting that it's for this bond…" For the WyCash method in question, the definition was of Heideggerian opaqueness.

At some point, Ward's team made a concerted effort to simplify the method by turning it into a *method object*. A method object is a new named bundle of data that responds to only one imperative sentence: "do whatever it is that you do". A new kind of object has to have a name; they picked Advancer, because it came from the method that advanced positions. For technical reasons that don't concern us here, method objects are useful when modifying overly complex code. They're often an intermediate step - you convert a bad method into a method object, clean it up by splitting it into smaller methods, then move those to the objects where they really belong.[12] But this team saw that Advancers were a useful new concept in the domain of bond trading, even though it wasn't a concept that real bond traders had. The programmers could get an intellectual grip on a new requirement by thinking about how they'd change or create special kinds of Advancer. They could write better code faster. [13]

Advancers later helped out in another way. The program calculated tax reports. What the government wanted was described in terms of positions and portfolios, so the calculations were implemented by Position and Portfolio objects. But there were always nagging bugs. Some time after Advancers came on the scene, the team realized they were the right place for the calculation: it happened that Advancers contained exactly the information needed. Switching to Advancers made tax reports tractable. Another speedup.

It was only in later years that Cunningham realized why tax calculations had been so troublesome. The government and traders had different interests. The traders cared most about their positions, whereas the government cared most about how traders came to have them. It's that latter idea that Advancers captured, but conversations with experts couldn't tease it out - even tax experts didn't know how to express it that way. It only came out through what agile programmers would term a conversation with the code and Pickering would call an episode of resistance and accommodation.

---

[12] You can find more about method objects in Fowler's *Refactoring* (1999), the canonical text on how to make code better without changing its behavior. There is a whole craft around "refactoring" - how do you know when a cleanup is warranted, which cleanups to make and when. *Refactoring* talks of that, as do Wake's *Refactoring Workbook* (2004) and Kerievsky's *Refactoring to Patterns* (2004).

[13] Agile programmers don't lightly create new words. Most of the time they try to use words from the business domain. I suspect these languages are what Galison calls creoles in his *Image and Logic* (1997): although "holding" means a quite different thing to a bond trader than to a programmer on a bond trading system, it allows them to coordinate their actions to each of their benefits. Evans's *Domain-Driven Design* (2004) has a good discussion of this mediating language from the point of view of programmers. It's what he calls the project's "ubiquitous language".

I earlier claimed that Agile programmers are manglish. This episode shows that. There is no way to tell the story of Advancers except as a **trajectory through time**. That trajectory is affected by **chance and contingency**; had it not happened that there was messy code to clean up, no one might ever have thought of Advancers. Certainly other people had written bond trading applications without making that conceptual leap.

Further, agile programming work is **performative, not representative**. A programmer tries something, the program resists or not, in one way or another, and the programmer **accommodates that resistance**. Certainly representations and abstractions are created - Advancer is one such - but, for many programmers, they are more likely to follow performance than precede and drive it. Here's one programmer on the topic:

> 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**.[14]

The accommodation of resistance, the working of code, is emergent. Moreover, this working of code continues throughout the project in a **dance of agency** in which the growing product is constantly **tuned** to the wants of the market. However, because the market tends not to be as stable as, say, subatomic particles, there is often no permanent product-level interactive stability. The history of a product is often one where it is interactively stabilized against the set of requirements and users known at the time, then released into the wild, whereupon new requirements are thrown up as users tune their behavior to the new product, see new opportunities, and demand new behaviors. It is not unknown for a team to make a shift as substantial as Hamilton going from three to four dimensions or Glaser going from cosmic rays to particle accelerators.[15]

There's also what seems to me a **decentering of agency**. It is not uncommon for programmers to say "the code is trying to tell us something. Let's try it." when they encounter resistance.[16] Note also the Ron Jeffries quote above, which ends with "I swear I'm not doing it" - though he is of course doing *some* of it. The code doesn't change itself. I see a dance of agency among four agents: Ron Jeffries, the code, rules that Ron follows and interprets, and the flow of requirements coming from the product owner. See also Jeffries' *Extreme Programming Adventures in C#* (2004a) for an extended example.

Over a number of iterations, there arises one form of **interactive stability**: the programmers and code are so tuned to each other that they can correctly predict how

---

[14] William Caputo, testdrivendevelopment Yahoogroups mailing list, March 9 2003.

[15] Personal communication from Nick Alesandro, a product owner: "We had built a prototype of an online EDI translator app and were on the way to fleshing it out into a real app. Then in January-ish 2004 we changed to the current model (a transaction lifecycle visibility app). Some core components could easily be repurposed to the new app, but others were irrelevant. The switch was fairly smooth (all things considered) for the dev team, although a bit disruptive for [me]." That smooth switch for programmers is the unique promise the Agile methods make to business. Personal communication from Ron Jeffries: A programmer on an project building a payroll system said of their product owner, "If Marie started giving us air traffic control [features], she could turn this thing into an air traffic control system and we'd never notice." (Both Ron Jeffries and Chet Hendrickson believe the quote is from Chet, though neither is entirely sure.)

[16] That quote is from Beck's *Smalltalk Best Practices Patterns* (1996), page 6.

many features they can finish in an iteration. (They have a known and constant velocity.) It's not fanciful to consider team and code a single unit, since neither the programmers, the code, nor the product owner can be replaced without requiring another complete process of tuning.

And I will finally even claim that **material agency** plays a role. It seems absurd to talk about it: computer code is hardly material, being nothing but patterns of electrical charge. But let me contrast three quotes. The first is from Pickering:

> The world, I want to say, is continually *doing things*, things that bear upon us not as observation statements upon disembodied intellects but as forces upon material beings. (*Mangle*, p. 6)

The second is from a programmer, J.B. Rainsberger (2005). The moment he describes is midway through a test-first[17] development session:

> In the process of writing these tests, I felt a familiar twinge that usually indicates the onset of a design problem.

And, finally, we have Kent Beck and Martin Fowler, who popularized the now-ubiquitous phrase "code smells" in Fowler (1999, p. 75):

> If it stinks, change it.
> - Grandma Beck, discussing child-rearing philosophy

What sensations are closest to "forces upon material beings"? Pain and smell. We do not stand back, disembodied, and observe them. Instead, it feels as if they act upon us. A hot stove *makes* our hand jerk back. A foul stench *makes* us retch at the thought of eating that rotten food. Using this sensory jargon helps train programmers to fix code without hesitation or second-guessing. It moves right action from the intellectual plane to the plane of reflex.[18]

I've completed my argument that Agile projects are self-consciously manglish (though they do not use that terminology). They're manglish because project members enjoy it. It's not surprising to hear programmers and product owners refer to an Agile project as the best project they've ever worked on.

---

[17] In test-first development, the programmer first devises a simple example of what some chunk of code should do and implements it as a test that the code either passes or fails. Since the code doesn't exist yet, it fails. He next writes a tiny bit of code that makes the test pass. There is now one good example of what the code does, and code that does nothing but match that one example. He then picks another example, probably slightly more complicated, and goes through the process again, slightly expanding the code. The process continues until there are no more examples. See Beck 2002, Astels 2003, and Hunt&Thomas 2003.

[18] There are many examples of Agile projects relying on perception to cause - or at least reinforce - right behavior. Here's a favorite. Agile teams typically rebuild the system at very frequent intervals so that incorrect code changes will be discovered quickly. One team uses two lava lamps to signal the state of the build. While "the build is good", a green lava lamp bubbles. When "the build breaks", the green lava lamp turns off and a red one turns on. It takes about twenty minutes for bubbles to start rising in a lava lamp, so the instant the red lamp goes on, the race is on to fix the problem before the bubbles start. That's a completely arbitrary deadline, but it sidesteps any thinking about what's most important to do. The lava lamp story is given in more detail in Clark's *Pragmatic Test Automation* (2004).

To be allowed to work in a manglish way, Agilists must convince the people who pay their salary. How do they do that? I've seen a few ways that seem specific to Agile projects.

All software projects become "black boxes" in Latour's sense[19], but an Agile project is a black box tuned to be responsive to change. The network in which it lives (the business) desires three things of it:

1. That it be able to answer the question "How much would this new or changed feature cost (in terms of time)?",
2. That the answer often be much less than the feature's value,
3. And that the answer almost always be close to right.

The team can be viewed as a **reliable tool that is well-suited to use by its wielder.** In Pickering's terms, the team provides disciplinary agency. There is some resistance outside the team. Either the product owner or someone else makes an imaginative leap: perhaps the resistance could be removed or even enrolled if *this* change were made to the product. Now the business cedes agency to the programmers, who will throw up an estimate nearly as incontestable as the results Hamilton got when multiplying two equations. The product owner judges the estimate. If it's too big, she either simplifies the idea or finds a new one. If the estimate is acceptable, she directs the programmers to throw up an implemented feature (again, much like algebra threw up results to Hamilton). Now agency again goes elsewhere: to the business, which may use the feature to spark new ideas; or to the outside world, which will react to the feature in whatever way it does.

That's the outside of the black box. What's invisible inside it is the ontological character of the team. It **makes no demands on the worldview of those who use it**. It allows the outside world to be - and to consider itself to be - exactly as manglish as it desires. Even though an Agile team will thrive most when its environment is a business that adapts quickly to its environment, Agile teams also succeed in "torpid" businesses.[20]

Not only do Agile teams make no demands on the outside's worldview, they **incorporate the outside worldview into their own**. When I first became involved in Agile development, the most surprising thing to me was how intent team members were about pleasing the businesspeople. There's a long tradition of programmers being scornful of them and even of the product's users.[21] But when Agile programmers talk about

---

[19] "The word **black box** is used by cyberneticians whenever a piece of machinery… is too complex. In its place, they draw a box around which they need to know nothing but its input and output." (Latour, *Science In Action*, pp. 2-3). "[Once a composite object has been assembled into a black box], it is made up of many *more* parts and it is handled by a much *more* complex commercial network, but it acts as one piece." (ibid, p. 131)

[20] Sometimes Agile projects "disguise" themselves by following Parnas and Clements' advice to the extreme: they produce all the documents that are supposed to be inputs to their work after the fact, when it's easiest. Other projects are suffered to survive because of their success (and, it has seemed to me, because they have an uncommonly forceful and devoted manager). The organization tolerates them but doesn't let them infect anyone else with their strange ideas.

[21] See Raymond, *The New Hacker's Dictionary* (1991) for the long history of programmers referring to users as "lusers", not coincidentally pronounced the same as "losers". A common quip in computer circles is "there are only two professions that refer to their customers as 'users'… and one of them is illegal." That's

"delivering business value", they don't speak with a cynical or ironic inflection. Committed Agilists are *extremely* reluctant to work on tasks that they cannot tie directly to a visible result that the product owner has deemed of value.

The cynical might call this an instance of Stockholm Syndrome, but I suspect it is a way of protecting **a necessary separation between the inside of the box and the outside.** The product owner, as representative of the business, has final say over the value of a feature; the programmers, as representative of the code, have final say over the cost. The essence of a beginning-of-iteration planning session is for the product owner to present a stack of feature cards in priority order. The programmers select from the top however many they can finish in the iteration. End of planning.

For the business to consider the team a well-suited tool, they must believe the estimates. If they believe that the programmers are padding estimates with "gold plating"[22], they will demand that the programmers find a quicker way to do it. The programmers can do that, but only by making the code worse. The rot isn't visible, though, so the product owner is likely to insist again on "aggressive" estimates. The project is now on that trajectory toward a Big Ball of Mud.

A constant, seemingly obsessive concern with delivering business value is a way for programmers to **signal alignment** with the product owner's interests. The box remains black, the workings remain invisible, but the product owner knows that the mechanism inside is the kind that will actively stay tuned to her. So she is less likely to intrude on their responsibilities.[23]

Business value is delivered in a special way: it is **visible, tactile, and frequent**. When a feature is done, the product owner most likely tries it out. Many agile projects, particularly those that use Scrum, have an end-of-iteration ritual in which they demonstrate that iteration's advances to anyone they can convince to watch (most particularly the *product sponsor*, who is the person who decided to spend the company's money on the whole project). Some projects set up a computer whose display allows anyone who happens past to browse the product's automated business-facing tests[24] and

---

only funny if the listener accepts that drug dealing and software construction have more in common than just a word. Attitude, perhaps?

[22] Gold plating is code that's not needed for the problem immediately at hand. Sometimes it's code that will be handy when that feature you *know* will come arrives. Other times, programmers use it to make a dull task interesting. The problem with prospectively useful code is that the feature often never comes, so the code is a waste. Or the feature comes in such a different form that the code just gets in the way. The acronym used to remind people of that is YAGNI, for "you aren't going to need it". For an article discussing gold plating, see Cohn (2005).

[23] In practice, there can be negotiation about both value and cost. If the feature is too expensive, the product owner and programmers might jointly find a way to trim it down to one that has slightly less value but substantially less cost. Or a feature might be so valuable that the product owner is willing to take on what's sometimes called "technical debt" (the phrase is due to Ward Cunningham) by accepting that time saved now by hurried work will have to be paid back with interest later. The programmers remain authoritative on the "interest rate" - the impact of the proposed shortcut.

[24] I distinguish between business-facing and technology-facing tests. A business-facing test is written in the language that a business expert would use, so it is meaningful to passers-by. A technology-facing test is written in the internal language of the system, so it is not.

run whichever they please. A passed test means that more recent changes haven't broken some earlier feature: it continues to have its business value.

This drumbeat of progress, as continuous as possible, gives license to the odd behavior of the team: the untidy bullpen, the constant chatter, the strange names (Extreme Programming? Scrum Master?[25]) and the crude profusion of whiteboards, sticky notes, and index cards.

At this writing, Agile software development has only a foothold in corporations. It appeals to programmers with a manglish bent, product owners who enjoy the idea of driving continuous evolution of a product, and executives who view their role as actively adapting to a market. But it has trouble with managers who see their job as making plans, directing workers, and tracking progress against plan. They do not appreciate the loss of control. So it's all too common to hear of successful Agile teams disbanded by a new Vice President of Engineering who is aghast at the disorder he sees and decrees a more "mature" process.[26] The Agile community also has difficulty justifying itself to those who "manage by numbers". Such people, typically senior executives, are used to demanding precision. Their job is to weigh opportunity cost: given a proposal to spend *x* euros on project *p* for monetary benefit *y*, they must decide, first, whether the predicted *x* and *y* are believable and then whether there is a more profitable way to spend *x*. To do that, they need *x* and *y*. An Agile team is not obviously geared to supply those numbers. Instead, they say that, if the business picks a direction, the team will produce a steady stream of deliverable features in that direction, and that the team will proceed at the highest velocity they are capable of. But they cannot *guarantee* what value *x* euros will deliver.

Nevertheless, the Agile methods have made surprising inroads in the four years since the phrase was coined.[27] It's a promising example of a group of people deliberately working in ways that can be readily mapped onto Pickering's model - and being successful enough

---

[25] The Scrum Master is Scrum's equivalent of a project manager. The twist is that she is a master of *Scrum*, not of the team. Indeed, one of the hardest jobs a Scrum Master has is to keep her mouth shut and let the team figure out their own solutions (Ken Schwaber, personal communication). The Scrum Master's role is supportive, not directive. One story, possibly apocryphal, is that a team was not allowed a bullpen because the company standard was cubicles. The team said they needed a bullpen to work effectively, so the Scrum Master came in on the weekend and moved cubicle walls, desks, and equipment himself. On Monday, he said he would resign if the bullpen were taken away. Apocryphal or not, that story exemplifies the attitude a Scrum Master is supposed to have.

[26] "Mature" is a term of art, due to the Software Engineering Institute's Capability Maturity Model, which describes five levels of maturity. Each higher level of maturity involves mastery of certain broad practice areas that (supposedly) contribute to more predictable and controllable results. See Humphreys' *Managing the Software Process* (1989) and SEI's *The Capability Maturity Model: Guidelines for Improving the Software Process* (1995).

[27] The phrase was coined in February 2001. A group of proponents of what were then called "lightweight methods" gathered to find out what they had in common. The term "lightweight" was not considered very flattering, so "agile" was chosen instead. The group also produced "The Manifesto for Agile Software Development" (www.agilemanifesto.org). See the appendix.

to find tolerance and even support from an unexpected quarter, the business world. I am sure that the ways in which Agility gains support do not apply to all manglish ways of living. Nevertheless, I hope this article suggests some promising directions for one next step in "manglish studies", which is to develop an analytical framework that goes beyond the descriptive to the performative: one that helps those with a manglish bent change their world.

Appendix: The Agile Manifesto

The Agile Manifesto was written in two stages. The first page (statement of values) was finished at the original meeting. The principles that follow were created in an email conversation.

Manifesto for Agile Software Development

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

Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan

That is, while there is value in the items on
the right, we value the items on the left more.

Kent Beck	James Grenning	Robert C. Martin
Mike Beedle	Jim Highsmith	Steve Mellor
Arie van Bennekum	Andrew Hunt	Ken Schwaber
Alistair Cockburn	Ron Jeffries	Jeff Sutherland
Ward Cunningham	Jon Kern	Dave Thomas
Martin Fowler	Brian Marick	

© 2001, the above authors. This declaration may be freely copied in any form, but only in its entirety through this notice.

Principles behind the Agile Manifesto

We follow these principles:

Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.

Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.

Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.

Business people and developers must work together daily throughout the project.

Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.

The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

Working software is the primary measure of progress.

Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.

Continuous attention to technical excellence and good design enhances agility.

Simplicity--the art of maximizing the amount of work not done--is essential.

The best architectures, requirements, and designs emerge from self-organizing teams.

At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

References

Astels, David (2003)
Test-Driven Development: A Practical Guide.

Beck, Kent (1996)
Smalltalk Best Practices Patterns.

— (2000)
Extreme Programming Explained: Embrace Change.

— (2002)
Test-Driven Development: By Example.

Beck, Kent, and Ward Cunningham (1989)
"A laboratory for teaching object-oriented thinking", in *Proceedings of OOPSLA '89* and the special issue of *SIGPLAN Notices*, Vol. 24, No. 10 (October 1989).

Clark, Mike (2004)
Pragmatic Project Automation.

Cockburn, Alistair (2001)
Agile Software Development.

— (2004)
Crystal Clear: A Human-Powered Methodology for Small Teams.

Cohn, Mike (2005)
"No title yet", in *Better Software Magazine*, Vol. 7, No. 2 (February 2005).

Cunningham, Ward (1996)
"EPISODES: A Pattern Language of Competitive Development". In *Pattern Languages of Program Design 2*, edited by Vlissides, Coplien, and Kerth.

Evans, Eric (2004)
Domain-Driven Design.

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.

Galison, Peter (1997)
Image and Logic: A Material Culture of Microphysics.

Humphreys, Watts (1989)
Managing the Software Process.

Hunt, Andy, and Dave Thomas (2003)
Pragmatic Unit Testing in Java with JUnit.

Jeffries, Ron, Ann Anderson, and Chet Hendrickson (2000)
Extreme Programming Installed.

Jeffries, Ron (2004a)
Extreme Programming Adventures in C#.

— (2004b)
"Big Visible Charts" http://www.xprogramming.com/xpmag/BigVisibleCharts.htm (accessed November 2004).

Kerievsky, Joshua (2004)
Refactoring to Patterns.

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.

Parnas, David, and Paul Clements (1986)
"A rational design process: How and why to fake it", *IEEE Transactions on Software Engineering*, Vol. 12, No. 2 (February 1986).

Pickering, Andrew (1995)
The Mangle of Practice: Time, Agency, and Science.
Pressman, Roger (2004)
Software Engineering: A Practitioner's Approach (6/e).
Rainsberger, J.B. (2005)
Title not chosen yet, *Better Software Magazine,* Vol. 7, No. 3 (March 2005).
Raymond, Eric (ed.) (1991)
The New Hacker's Dictionary.
Schwaber, Ken (2004)
Agile Project Management with Scrum (2004).
Schwaber, Ken and Mike Beedle (2001)
Agile Software Development with Scrum.
Software Engineering Institute (1995)
The Capability Maturity Model: Guidelines for Improving the Software Process.
Sommerville, Ian (2000)
Software Engineering (6/e).
Wake, William (2004)
Refactoring Workbook.
Williams, Laurie and Robert Kessler (2002)
Pair Programming Illuminated.