A Manglish Way of Working: Agile Software Development

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 only 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.*

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.

Until recently, no one told such stories about the development of software products. Instead, the story of a software project was supposed to follow this template:

1. Work begins by producing a complete list of requirements the product must meet. A requirement is a truth-valued statement about the next released version of the product. Once all requirements have been met, the product has, by definition, solved the problem that prompted its creation.
2. Work continues with a specification of the product's interface. It describes everything that anyone with minimal qualifications would need to know to predict how the product would behave when given an arbitrary input. The specification also meets all the requirements.
3. The next step is an abstract design of the product's internals (often called the "architectural design"). This design, when realized, implements the specification.
4. There follows some number of progressively less abstract designs. Each one realizes the one before it.
5. Written last, the actual code (instructions to the computer) realizes the least abstract level of design and thus, transitively, satisfies the requirements.

The standard texts[1] asserted that completing steps like these in that sequence was the ideal way to develop software. They did concede that people err. People overlook requirements. They write requirements ambiguously, causing later writers to produce the wrong design. They think a design implements the specification when it does not. And so on. Practitioners were instructed to take failure into account, mainly by putting in place measures to detect and correct it as soon as possible.

However, these deviations from the story line were deemed inessential and uninteresting. Consider, for example, Parnas and Clements' widely cited "A rational design process: How and why to fake it" (1986). In it, they advocate removing all traces of the historical trajectory from the writeup of a software product's design. Suppose a requirement was ambiguous. Parnas and Clements would have us revisit the requirements document and rewrite it to remove the ambiguity. We should then propagate the consequences of that change to all the other documents. The resulting document set describes the product's history as a logical progression rather than actions in time–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." The error can now be forgotten.

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

Agile software development

Agile projects[2] run differently. The programmers are not guided by documents, but by a person who knows well a *domain* like bond trading or nursing. I call this person the *product director*.

The project progresses through a series of *iterations*, each from one week to one month long. At the beginning of each iteration, the product director tells a team of programmers and other technologists about the most important features currently missing from the product. The team responds with estimates of how long it would take to add each feature, and the product director selects a set of features that can be finished within the iteration. The team immediately begins writing code to implement those features, asking questions of the product director 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 the new features. The

---

[1] Two of the most popular texts have been Pressman's *Software Engineering: A Practitioner's Approach* (2004) and Sommerville's *Software Engineering* (2004). Note that older editions show the conventional ideal in purer form. The newer ones contain ever increasing admixtures of Agile and other unconventional thinking.

[2] For a description of the Agile methods in general, see Cockburn, *Agile Software Development (*2001). The general approach has many variants. The two most documented (and probably most common) are Extreme Programming and Scrum. For Extreme Programming, see Beck, *Extreme Programming Explained: Embrace Change* (1999) 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).

product director could conceivably stop the project at the end of any iteration, with no advance warning, and ship the product to end users.[3]

According to conventional thinking, such a project is doomed. After the first iteration, the team will have code that supports only the first set of features. In the second iteration, they'll have to implement features they hadn't anticipated. Because they hadn't planned for them, they'll find those features hard to wedge into the existing code. They'll be able to do it, but they'll inevitably leave some of the code worse than before–making iteration three harder still. Over time, the product will decay into what Foote and Yoder (2000) call "a big ball of mud." At that point, it will take heroic and lengthy effort to add a single feature. According to the textbook, the Agile project is like the hare in the fable: it'll look productive at first, but a properly run project will, tortoise-like, win the race.

Agilists, 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 more accommodating of change, gradually producing 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.[4]

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 good software design. Ron Jeffries, one of the three people most responsible for developing and popularizing Extreme Programming (Beck, 1999), puts it this way:[5]

> Beck [another of the three] 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.

---

[3] 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.

[4] 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.

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

> Yet the overall design of the system improves. I swear I'm not doing it.

Agile programmers also rely a great deal on constant communication. To reduce barriers to communication, teams typically work in bullpens instead of offices or cubicles. Most have daily meetings intended to tell each other what they did yesterday, what they plan to do today, and what help they need (Schwaber and Beedle, 2001; Beck, 1999). It is also common for programmers to program in pairs (Williams and Kessler, 2002). Each programmer might rotate through all parts of the product, pairing with each other programmer, thus gaining a generalist's knowledge of the whole rather than a specialist's knowledge of a part.

An example

People in Agile projects err as much as those in conventional projects, but the attitude toward error 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.[6] It's the story of Advancers, in which a new concept emerged over a series of iterations through what programmers would term a conversation with the code and Pickering would call episodes of resistance and accommodation.

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 faster and more pleasant for bond traders to work with, important in a high-pressure environment. Second, users would be able to generate reports on a position (a collection of holdings) as of any date, a feature the competition lacked.

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. The team was taking longer to produce features, and they created more bugs.

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. Programmers often approach such tasks by applying *refactorings*. A refactoring is a change to the code that doesn't change its behavior (just as factoring out the variable *x*

---

[6] 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).

from *ax+bx* by turning it into *(a+b)x* doesn't change the meaning of the equation).[7] They converted the troublesome method 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." 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.

This team, however, left the method object in the program. The reasons are lost. Perhaps, as is often the case, the right way to split it up wasn't apparent.

Leaving a temporary object in a program turns a casual decision into a more serious one. Each kind of object must have its own name. A temporary object's name can be meaningless, but a name other programmers will encounter should communicate something to them.

Naming is an important topic in programming. Some of the objects programmers use have nothing to do with the world outside the program. But some of them do. In the latter case, programmers try to use names that make sense to domain experts. WyCash, for example, had Position and Portfolio objects. Collections of such nouns (object names) and verbs (method names) are referred to as "ubiquitous languages" (Evans, 2003). They function like the creoles Galison (1997) describes: although "position" means a different thing to a bond trader than it does to a programmer, the word allows them to coordinate their actions to each of their benefits.

The temporary name for the new object was "Advancer" because it was formed from a method that advanced Positions. When the programmers asked the domain experts for a better name, they didn't have one. "Advancer" wasn't an idea that bond traders used. So the programmers kept the name and continued to figure out what the object meant by seeing how it participated in program changes.

Advancers turned out, by accident, to be broadly useful. As the project dealt with the normal stream of changes, the programmers found they could get an intellectual grip on them by thinking about how to change existing Advancers or create new ones. They wrote better code faster.

For example, the program calculated tax reports. The government wanted reports described in terms of positions and portfolios, so the programmers implemented the calculations in Position and Portfolio objects. But there were always nagging bugs: someone would run a report on a novel portfolio, only to find that some of the numbers were wrong. Some time after Advancers came on the scene, the team realized they were the right place for the calculations: it happened that Advancers contained exactly the information needed. Switching to Advancers made tax reports tractable. Another gain in the team's capability, and a further understanding of what Advancers were about.

---

[7] Fowler's *Refactoring* (1999) is the canonical text on how to make code better without changing its behavior. There is a whole craft around refactoring. *Refactoring* talks of that, as do Wake's *Refactoring Workbook* (2003) and Kerievsky's *Refactoring to Patterns* (2004).

It was only years later 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 the latter idea, one that the experts did not know how to express, that Advancers capture. And once it's captured, the complexities of tax calculations collapse into (relative) simplicity. But at no point in the story did the programmers specifically set out to invent something new in the language of bond trading. They were only trying to generate the required reports while obeying rules of code cleanliness.

To summarize:

1. An unending stream of unanticipated changes caused programmers to revisit an area of code again and again, iteration after iteration, each time changing the area.
2. As is often the case, the code resisted change. A particular technology – refactoring – was used to make the code more accepting of change. Refactoring is a mostly mechanical process of moving bits of code around without changing the program's external behavior. Its goals have nothing to do with the world outside the program.
3. Nevertheless, some of those moving bits of code coalesced into an abstraction that was strikingly useful when programmers were given unanticipated business problems to solve.
4. Computer programs require that abstractions have names. Something so useful in solving business problems *ought* to be something business people have a name for. It appeared they did not. But once the team picked a name, it became radically easier to think about certain business problems.
5. Advancers presented themselves incrementally and emergently ("bottom up") without anyone at any point saying "We need a large conceptual leap to solve this constellation of business problems." But, after the fact, they were in fact a large conceptual leap, discovered through a process I find strikingly similar to Pickering's story of Hamilton and quaternions (Pickering, 1995, ch. 4).

Agile and the mangle

The story of Advancers is manglish. I can think of no good way to tell it except as a **trajectory through time**. That trajectory was affected by **chance and contingency**; had it not happened that there was particular messy code to clean up, no one might ever have thought of Advancers.

Further, the Advancers story illustrates how 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 programmer Bill Caputo on the topic:[8]

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

> 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**.

There's also what seems to me a **decentering of agency**. When programmers encounter resistance, it's not uncommon for them to say "the code is trying to tell us something. Let's try it." (Beck, 1996, p. 6) 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 director.[9]

I 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 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's *Refactoring* (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.[10]

---

[9] See Jeffries' *Extreme Programming Adventures in C#* (2004a) for a book-length example of decentered agency.

[10] 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 product 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 whether fixing the build is the most important to do. Everyone can see it is. The lava lamp story is given in more detail in Clark's *Pragmatic Project Automation* (2004).

So we have a **dance of agency**, where humans accommodate the resistance of code and, by doing so, **tune** themselves and the code so that they can make requested changes predictably and cheaply enough. But when it comes to **interactive stability**, there seems to be a difference here from Pickering's stories. In those stories, the stable point is one at which the bubble chamber can be used to observe the tracks of new particles, free quark experiments can be performed to the experimenter's satisfaction, or Hamilton has a tool that can be used to solve mathematical problems (albeit not the one he started with). It's a point at which people can use a created object as a reliable enough tool. But in an Agile project, the product is supposed to be usable throughout. Even after very early iterations, users could get some value from it. That value increases with each new iteration. It increases the iteration before the product is released, and it increases in the iteration after (the first iteration building toward the next release). Is that steady pace compatible with Pickering's concept of interactive stability?

I believe it is, once you make two perspective shifts. The first is that the product's code is not the interesting result of the dance of agency. Rather, it's the assemblage of programmers + code + product director. In the course of the project, they become tuned to each other such that none of them can be easily replaced. If one of them is replaced, performance will drop drastically until all three components retune to each other.

The second shift is to forget that the word "stability" implies a lack of movement. There is an active or dynamic stability that I associate with someone balancing a broom during a mild earthquake on a fitfully windy day. The stability is a consistent (stable) *capacity to adjust swiftly and appropriately to a world full of chance*. It is this kind of interactive stability that, it seems to me, Agile projects have. They have arrived at that capacity by tuning themselves to a constant stream of deliberately unexpected change requests, explicitly assuming that the dance of resistance and accommodation will never end. Reflecting this back to one of Pickering's case studies, it is as if Glaser's bubble chamber were Nobel Prize-worthy not because he enabled experimentalists to create particle tracks they never could before, but because the existence of the bubble chamber made yet more experimental instruments strikingly easier to create.[11]

Getting away with it

I've completed my argument that Agile projects are intentionally manglish. Project members choose the mangle because they enjoy it. I'm not surprised when I hear programmers and product owners refer to an Agile project as the best one they've ever worked on. But how do they get away with it? Business doesn't leap to mind as the most fertile ground for manglish behavior.

The two key moves, it seems to me are to **become a personalized black box**[12] and to **reinforce the base ontology.** I'll explain the adjectives "personalized" and "base" later,

---

[11] Glaser and the bubble chamber might very well have done that by contributing to the experimentalist tradition in particle physics (Galison, 1997). And I imagine the invention of quaternions allowed new mathematics to appear. But I suspect that neither Glaser nor Hamilton gave much thought to whether solving their current problem would make the currently unknown *next* problem easier to solve. Agile projects aim at that.

[12] I use "black box" in Latour's sense: "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

but first let me give a thumbnail sketch of the base ontology. In it, businesses are machines for converting money into more money. A business is composed of three parts:

1. The part named *PLANNING* balances predictions of the external world's future and the abilities of the company. Given this moment's understanding of possible futures, what use of today's surplus money will produce products that yield the highest risk-adjusted expected return? I'm lumping into PLANNING corporate executives, the *product sponsor* (who perhaps proposed the project to the executives, is the keeper of the budget, and will be held responsible if the project fails), and the product owner (who often works for the sponsor).
2. *DEVELOPMENT* converts money, time, and the desire for products with particular new features into actual products.
3. *DELIVERY* moves products into the hands of people who will give money in return. (I include here advertising and sales as well as order fulfillment.)

PLANNING can consider DEVELOPMENT a black box if the latter has three properties:

1. It can answer the question "How much would this new or changed feature cost?"
2. The answer must often be much less than the feature's value, and
3. The answer is almost always close to right.

Those properties given, what happens within the black box to produce the answer and build the software is of no interest to PLANNING, so those within the box can arrange its interior however they like.

The problem with conventional software development is that the answer *DEVELOPMENT* provides is almost always wrong: it's asked to estimate all of the project tasks at the moment of least knowledge, the beginning of the project. Agile teams, in contrast, make firm estimates for only the next iteration's tasks and only just before it begins. At that moment, they can bring to bear what they've learned in previous iterations.

Traditional *DEVELOPMENT'S* inaccuracy means the outside must break open the black box and meddle with the workings to make it more reliable. PLANNING still wants a black box, so the work is delegated to a new entity, not part of the base ontology. That's *MANAGEMENT*, whose job is to be an interface between PLANNING and DEVELOPMENT, presenting the three key properties to the former and directing the latter.

That doesn't usually work either, but the Taylorist command and control doctrine in business is so strong that it's hard to conceive of an alternative. MANAGEMENT is taken to be as fundamental a thing in the world as features, money, the future, PLANNING, and DEVELOPMENT. But it is actually a logically derivative role. *In order to* make the business a money-generating machine, MANAGEMENT is needed. Were there an alternative, business would happily exist without it, but MANAGEMENT wouldn't exist if the base ontology were rejected. It's that sense in which I call that ontology "base".

---

input and output." (*Science In Action*, 1988, 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)

One way that Agile projects get away with being manglish is to "route around" MANAGEMENT. DEVELOPMENT's explicit message to PLANNING is that the role of mediating manager is unneeded. The business can run better without it. PLANNING should view the team as **a reliable tool easily wielded**. "Reliable" here means that it's a black box with the three properties above. "Easily wielded" emphasizes the second of the three. As noted, PLANNING is concerned with the predicted best use of money *at this moment*. One month after a project starts, it's likely that the best use of the unspent money will seem different than it did at the start. The world will have changed in some unexpected way or PLANNING will have learned something new. The business would be a better machine if PLANNING could redirect DEVELOPMENT at any moment. Conventional projects have great inertia; the answer to "can you change this planned capability now?" is very often more than the change is worth. Agile projects promise that it will seldom be.

But making promises is easy. The hard part is getting the listener to believe. Agile projects use two noteworthy mechanisms to signal to PLANNING that they're trustworthy.

Progress is demonstrated 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 have an end-of-iteration ritual in which they show that iteration's advances to anyone they can convince to watch, most particularly the product sponsor and salespeople (Schwaber and Beedle, 2001). But visibility is more continuous than once every few weeks. If you walk into an Agile bullpen at any moment, you'll likely see what Jeffries (2004) calls "big visible charts" and Cockburn (2001) calls "information radiators". Among them will be a graph tracking expected against actual progress and a bulletin board studded with index cards, each of which represents one of this iteration's tasks. Some of the cards will have been X'd off with a thick marker stroke, meaning that the task is done. Someone who wanders past the bullpen on the way to the bathroom will be able to see both short-term and long-term progress at a glance. Some projects set up a computer whose display allows anyone who happens past to browse the product's automated tests and 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.

In fact, it's common for projects to solve purely internal problems by devising a chart that demonstrates the problem, putting it up where anyone can see it, and relying on everyone's desire to see the chart look steadily better to make the problem go away.

I call this **exhibitionist Panopticism**. Like the panopticism described in Foucault's *Discipline and Punish* (1977), it serves to create a subject. Programmers become the kind of programmer who makes the right decisions without being nudged by a chart. It's exhibitionist in that the subjects are actively choosing what the outside can view *and* choosing to show more than anyone outside asks for.[13] By so doing, they simultaneously make the walls transparent and make the project more of a black box. The constant

[13] The exhibitionism isn't restricted to outsiders. Programmers are traditionally protective of "their" code (though it legally belongs to their employer). I've found as a consultant that many programmers are reluctant to show their code to anyone else. They know it isn't as good as it should be. But pair programming, especially when pairs rotate, brings all code into everyone's view. That disciplines programmers to write code "they could show to their mother".

availability of information reduces the business's desire to meddle with the project, so the team retains autonomy to be manglish in the domain of programming.

Also significant is that Agile teams **honor and incorporate the outside ontology**. 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. That scorn sometimes represented an artisan's scorn for no measure of value other than "filthy lucre", sometimes a belief that they know what the market desires as well as PLANNING does. But when Agile programmers talk about "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.

That doesn't diminish their artisan's values; it appears to me those values are as strong or stronger in Agile projects than in conventional ones. It's just that the most reliable way to be allowed to exercise their values is to honor the business ontology. And the best way to honor it is to incorporate it in addition to artisanal and manglish values.

Incorporation is especially important because of the prevalence of **close, frequent, informal contact** with the product owner. Businesses aren't really machines, and teams are composed of people. In an ideal Extreme Programming project (Beck, 1999), the product owner's main physical location is in the bullpen with the programmers. Although she is typically nominally part of PLANNING, she's a visitor to them and a resident of the team. She can tell whether talk of business value is pro forma or sincere. For example, the programmers might start a conversation about how a feature could be simplified to reduce its cost. If the conversation starts and ends with what bits of the feature can be lopped off to save time, the programmers have not incorporated the base ontology. If instead the conversation begins with careful quizzing about the sources of value in the feature and only then moves on to how 80% of the value can be gotten for 20% of the effort (as often seems possible), the product owner can feel confident in the team.

Thus, the output of the black box is not just visible, tactile, frequent new product capability; it is visible, tactile, frequent new capability *accompanied by* a *PLANNER* who can vouch for the manglish people inside the box. It's that personal contact and trust, together with the drumbeat of progress, that gives license to the odd behavior of the team: the untidy bullpen, the constant chatter, the strange names (Extreme Programming? Scrum Master?[14]) and the crude profusion of whiteboards, sticky notes, and index cards.

---

[14] 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 one weekend and moved cubicle walls, desks, and equipment himself. On Monday, he said he would resign if the bullpen were taken away. It stayed. Apocryphal or not, that story exemplifies the attitude a Scrum Master is supposed to have.

Generally, the role of a manager in Agile projects is to remove impediments in the path of the team, facilitating agreement (rather than dictating it), having the formal authority to eject people from the team, and being an advocate for career growth and skill development.

The stirring conclusion

Agile software development has made surprising inroads in the six years since the phrase was coined. It's an example of a group of people deliberately working in ways that can be readily mapped onto Pickering's model - and being successful enough doing it that they 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 working. 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.[15]

[15] My thanks to Johannes Brodwall, Bob Corrick, Steve Freeman, Keith Guzik, and Ron Jeffries for their helpful comments on an earlier draft. Special thanks to Keith Guzik for pushing me to think about what really gets interactively stabilized on an Agile team.

References

Beck, Kent. (1996)
Smalltalk Best Practice Patterns.

Beck, Kent. (1999)
Extreme Programming Explained: Embrace Change.

Foote, Brian, and Joseph Yoder. (2000)
"Big ball of mud" in *Pattern Languages of Program Design 4*, Harrison, Foote, Rohnert eds.

Clark, Mike. (2004)
Pragmatic Project Automation: How to Build, Deploy, and Monitor Java Apps.

Cockburn, Alistair. (2001)
Agile Software Development.

Evans, Eric. (2003)
Domain-Driven Design: Tackling Complexity in the Heart of Software.

Foucault, Michel. (1977)
Discipline & Punish: the Birth of the Prison.

Fowler, Martin. (1999)
Refactoring: Improving the Design of Existing Code.

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

Humphrey, Watts S. (1989)
Managing the Software Process.

Rainsberger, J.B. (2005)
"Injecting testability into your designs" (tentative title). *Better Software* 7,4

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

Jeffries, Ron et al. (2000)
Extreme Programming Installed.

Beck, Kent, and Ward Cunningham. (1989)
"A laboratory for teaching object-oriented thinking." In *Proceedings of OOPSLA '89*,

Kerievsky, Joshua. (2004)
Refactoring to Patterns.

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* 12,2

Pickering, Andrew. (1995)
The Mangle of Practice: Time, Agency, and Science.

Pressman, Roger. (2004)
Software Engineering: a Practitioner's Approach (6/E).
Jeffries, Ron.(2004)
"Big Visible Charts."
http://www.xprogramming.com/xpmag/BigVisibleCharts.htm.
Schwaber, Ken. (2004)
Agile Project Management With Scrum.
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. (2004)
Software Engineering (7/E).
Wake, William C. (2003)
Refactoring Workbook.
Williams, Laurie, and Robert Kessler. (2002)
Pair Programming Illuminated.