The Gap Between Business and Code[1]

Brian Marick
marick@exampler.com

This paper argues that the continuing problems we have with requirements elicitation and transmission are not a result of poor skills or sloppy people. It's rather that the entire idea is based on dubious assumptions. A different assumption is introduced, namely that effectiveness can only be obtained by parties having iterative conversations over time, and that communication is meant more to provoke right action than to transmit understanding. Based on examples of this assumption in practice, suggestions for software development are sketched.

Biographical sketch

Brian Marick was a programmer, tester, and line manager for ten years, then a pure testing consultant for ten more. Since 2001, he's been heavily involved in the Agile methods. He was one of the authors of the Manifesto for Agile Software Development <www.agilemanifesto.org> and is past chair of the board of the Agile Alliance nonprofit. At this point, it's probably fairer to call him an Agile consultant and advocate than a testing consultant, though his Agile work is heavily informed by the lessons of testing. His testing work can be found at <www.testing.com>, and his agile work at <www.exampler.com> and <www.testing.com/cgi-bin/blog>. Many of his writings make it into the pages of Better Software Magazine, of which he is a technical editor.

[1] Because of poor time management on my part, this is not the final version of this paper. You can find that at <http://www.testing.com/writings.html>. Sorry about that. Based on past experience, I can say that you might find that version radically different from this one.

"... some feel they should not be given a headache when trying to understand meaning. Why? Why on earth should it be simple to explain how people create exquisite, infinite variations of meaning from elaborate squeaking and grunting rituals?"

- Nathan Stormer (impersonal communication[2])

As far as I know, no alchemist ever succeeded in turning base metals into gold. Why not? It wasn't for lack of trying: many people tried hard for many years, not without progress. Perhaps they were just using the wrong techniques – many of them thought so and devoted their lives to improvement and discovery. Perhaps they didn't take it seriously enough: it was widely thought that the purification of gold had to go hand-in-hand with the purification of the alchemist's soul.

Or maybe they were working from some bad assumptions. That's what I think.

We've been trying for many years to elicit and transmit requirements reliably. The results have been disappointing, though not without progress. Why? Maybe we don't know the right techniques - certainly we have people devoted to discovering new ones. And maybe we're not pure enough. Seriously: if the business people weren't so focused on what they see on the screen, and the programmers weren't so obsessed with the code behind it, maybe they would participate long enough and intently enough to get the requirements right.

Or maybe we've been working from some bad assumptions. That's what I think.

The way we think about requirements is shaped by two metaphors: the **conduit metaphor** and the **map metaphor**. If those metaphors are bad ones, no amount of skilled elicitation will produce golden requirements. Maybe we should give up and try something else. This paper is about the problems with those metaphors and the alternatives suggested by abandoning them.

Two Metaphors

The map metaphor is all about correspondence. Things in our heads – or words in our language – correspond to things outside in the world. You are always able to think the thought "chair" or say the word "chair", point at something in the world, and say whether that thing is a chair.

If you believe in this metaphor, the requirements problem amounts to building a map of the world inside which the program must perform well. This map allows one to mechanically solve any problem the program will encounter: that is, the solution can be computed using *only* the map and a fixed set of rules for working with it. Once given this map, the programmer's job is to translate it into some form a dumb binary computer can

[2] He wrote this in a comment to a blog posting, so it's not a personal communication. I did get permission to quote him.

compute with (given its fixed set of rules). This conversion does not change the map: it involves adding implementation details, not changing the nature of the program's world.

The conduit metaphor is about communication. It says that communication between two people consists of converting the map in someone's head into a message – likened to a physical object – which is shot over to the other person, who decodes it into an equivalent map. The conduit metaphor is deeply embedded into our offhand language, as is exhaustingly demonstrated by Reddy (1979), who provides many examples like this:

- You've got to get this idea *across* to her.
- I *gave* you that idea.
- It's hard to *put* this idea *into words*.
- Something was *lost* in translation.
- The *thought's buried* in that convoluted prose.
- I remember a book with that idea *in it*.
- That talk *went* right *over my head*.

This folk wisdom about communication affects our actions. When we have a hard time communicating, we use the metaphor to reason about the problem we're having and seek solutions (Lakoff and Johnson 2003). The requirements process demonstrates that. We seek ever better ways to make a written requirements document a faithful encoding of a map, so that any arbitrary qualified person will be able to decode it to the same result.

A different metaphor

I happen to believe those metaphors are incorrect. Philosophers have spent better than two thousand years trying to make the map metaphor work well enough to survive the implications of quite straightforward statements (Rorty 1981):

- "To what thing in the world does the word 'unicorn' correspond?" We talk about things that don't exist.
- "I now pronounce you man and wife." Some statements aren't representations *of* the world; instead, they're actions *in* the world (Austin 1975).
- "That depends on what the meaning of 'is' is." All words admit of multiple interpretations. Given enough work, some would say that *any* word can be made to have an indeterminate meaning (Culler 1983).
- "Why is a bean-bag chair a chair while a three-legged stool isn't?" Categories do not have clear boundaries (Lakoff 1990).

Of these, the last two are most relevant to our work. Lakoff shows that many categories we use have fuzzy boundaries. In particular, they have *central and peripheral examples*. When I say the word "chair", you probably don't envision a beanbag chair; instead, you picture a more typical chair: seat, four legs, back. Other kinds of chairs are less "chair-ey" than that.

As another example, Lakoff asks us to consider the word "bachelor" and asks, "Is the pope a bachelor?" Your answer is probably something like "well, yes, technically, I suppose the pope is a bachelor" – but he's clearly not as *good* a bachelor as a 25-year old subscriber to *Maxim* magazine living in the youth-oriented center of some city. The simple definition of "bachelor" – an unmarried male – is surrounded by a penumbra of preferred connotations. The pope isn't a bachelor because there's no reasonable prospect of him marrying, and the canonical bachelor (in my culture, at least) is someone who's *not yet* married – someone who will, more likely than not, marry before too long. For that reason, it's a little bit odd to call an unmarried gay man a bachelor.

Or is it? It may be odd for me, in a country (the USA) where gay marriage is intensely controversial and far from the law of the land. Is it as odd in Canada, the Netherlands, and Belgium, which allow same-sex marriage? There, the standard image of the young male bachelor sowing his wild oats before settling down to a committed legal relationship might well become independent of sexual orientation.

Because of this, you and I might have very different maps of the world that we both encode in the word "bachelor". Our problem is not just that "it depends on what the meaning of 'is' is" – that single words can have multiple meanings, and that the cannily malicious speaker can mislead us by allowing us to assume one meaning while pointing to an alternate meaning to defend against the charge of perjury. It's that the multiple meanings of words are not pin-down-able. There are, in principle, at least as many meanings for a word as contexts people operate in.

It gets worse: Every word's dictionary definition is made using *other* words, so that – if we believe in the map metaphor – to really understand the definition of "bachelor", we'd have to understand the definition of every word used in "bachelor's" definition, and every word in *their* definitions, and so on. That multiplies the number of possible (mis)interpretations. Worse: the process doesn't bottom out: eventually, some definition of "is" will use the word "is", and we're caught in a circularity. On the old *Star Trek*, the computer would now burst into flame.

We don't burst into flame because, I believe, the map metaphor is only a tool. For certain problems, communication and social action are convenient if we all pretend that our words really do map into categories in the world. It's like using Newtonian mechanics when calculating slow-speed motion: it's not *true*, but it works well enough until you get into the hard (relativistic) cases. Situations of great precision – like causing a dumb binary computer to make judgments humans think are sensible, even though it's not naturally good at categories without hard margins – require us to use different tools.

So what's a better metaphor for communication? One, which I heard from Richard P. Gabriel, is A POEM IS A PROGRAM. That is, a poem is some text that combines with its input (the mind, memory, and experiences of the reader) to produce a result (an understanding, an emotion, a mental map). The good poet writes poems that produce intended (or surprising-but-good) effects in the intended audience. The effects are more predictable when the audience belongs to the same *interpretive community* (Fish 1980), people who have been trained to react to certain texts in certain ways. That's the sense in which art can be seen as a conversation among those people – both artists and critics – so deeply involved in the creation of the art that they teach each other, through example or commentary (Bloom 1997, Gombrich 1995).

I'd like to extend Gabriel's metaphor, which still seems to me to bow too much toward the map and the conduit. My extension is to take up the notion of certain cyberneticians

(Pickering, in press) that the brain is not about thinking in the sense of representing the world, but is rather a *performative* organ. Ross Ashby (1948) puts it well:

> ...to the biologist the brain is not a thinking machine, it is an acting machine; it gets information and then it does something about it.

So I'm agnostic about whether a poem (or a requirements document) is a program: I don't know what happens after it enters the brain. What it does may sometimes be well analogized by the running of a program – that is, thinking that way might allow us to make better predictions about what will happen when someone reads a requirements document – but sometimes it might not. Rather than taking one approach to the limit, we'll need what James Bach (1999) calls "diverse half measures": a number of measures, each inadequate on its own, that in combination do better than any one of them possibly could.

In the absence of a single theory about what the brain is doing, let's concentrate on what it does things with: the inputs. Those can be divided into three kinds: statements (verbal or written), actions (smacking the programmer upside the head), and the setting (passive inputs gathered by the receiver, not projected by the sender).

I'm going to skip talking about setting, despite its importance. Drug users can elaborately arrange their environment (Leary et. al. 1963, Zinberg 1986) to maximize the effects they experience. The same person, taking the same drug in the same dosage, might experience very different effects in one setting than another. I expect that's relevant to debates such as offices vs. cubicles vs. bullpens, but I don't have anything useful to say about it.

I will also combine statements and actions. At least some statements *are* actions. Consider "I now pronounce you man and wife" above or "I christen thee *The Luisitania*". They are what Austin (1975) calls *performatives*. Derrida thinks all statements can be treated as performatives (Nilges, in press; Derrida 1988). I'm not sure of his argument, but that seems reasonable to me. Consider the statement by a person of influence that "there are no good chairs here". That's a statement about the world, but it's also likely a "speech act" that will cause someone with lower status to fetch a chair. Furthermore, it causes that person to make a specific interpretation of "good chair" – not as a universal, but as a set of objects that will cause the high status person pleasure when it's brought to her. A hearer is always actively (if implicitly) interpreting statements in terms of what the consequences of particular interpretations will be, which – given that we're primarily social animals – means that we wonder what *they'll* do should we do *X* in response to message *Y*.[3]

If all statements are actions, it doesn't make much sense to treat them separately. That allows us include physical actions as a part of speech, which seems more abstract. The

[3] Dennett (1992) proposes that consciousness was created through this mechanism. As social beings, we must become adept at making good – reasonably accurate, easy to use – models of other people's behavior. Consciousness is what results when that ability is turned on its possessor. (Greg Egan's novel *Diaspora* contains a narration of an individual's creation of consciousness by this means.)

written statement "the input field must accept all unicode characters" has a different effect than do the same words spoken and accompanied with a vigorous pointing gesture.

Learning from the bottom up

Here's an example of causing the right behavior without any obvious use of a map or a conduit.

> 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 usually err on the side of bright. It's Dawn's job to correct them. She and a student will stand near a misclassified cow, and Dawn will say, "This cow is dull. See—it's not cleaning its nostrils."[4]
>
> From this conversation, the student will create a rule, a little bit of a map of the world:
>
> > (1) If a cow looks bright, but it's not cleaning its nostrils, it's dull.
>
> That rule might work for a while, but eventually Dawn and the student will be standing beside a cow, and Dawn will say, "That cow is dull." The student will say, "But it's cleaning its nostrils!" To which, Dawn will reply, "But its ears are droopy."
>
> Now the student adds a new rule:
>
> > (2) Droopy ears mean dull, and perky ears mean bright.
>
> But those two rules don't work either. It might take a while for the student to be able to reliably judge between bright and dull. What's interesting is that, when she does, *she's lost the rules*. She can't articulate any complete set of rules that define bright or dull. In fact, the notion of decision rules 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 – not conceptual – world has expanded.

Several interesting things are happening here.

1. This is an example of what Lave and Wenger (1991) call *legitimate peripheral participation*. What the students are doing is legitimate because what they do *matters*; this is a real cow that will either go home or to the cooler. It's not a classroom exercise.

   The students are peripheral in that they begin by having unsupervised responsibility for very little. (As my first boss put it, "we're going to put you somewhere where you can't do much damage.") As they gain experience, their decisions become more central to the success or failure of the case.

[4] Trust me. You don't want to know how cows clean their nostrils.

The students participate in a larger group activity, where they have frequent opportunities to talk with both experts and learners like themselves.

2. The process is highly iterative. Early in our courtship, I realized that it was no wonder that Dawn was so much more competent at her job than I was at mine: in the time I spent working toward a single product release, she would see hundreds of cases through from start to finish. The feedback on her work was immensely faster than on mine. Further, the example shows that a student gets expert, useful feedback not just at the end of the case, but continually throughout it.
3. This learning is driven by examples. We can say the students generalize from them (while remaining agnostic about whether the generalizations are expressible in any language). That's different from the "top-down" approach in which students are taught general rules and then how to apply them to particular cases. The bottom-up approach seems highly risky: how can you be sure students trained at Illinois will make the same judgments as students from Colorado State? You can't, I suppose: but, nevertheless, they do.
4. Learning is an explicit goal of all involved. The students know that every case is about learning veterinary medicine, not merely about curing one cow. That affects the way Dawn speaks to a student when making a correction. It also causes the professors to organize explicit sessions devoted to review and extension of learning ("rounds").

I hasten to say that not everything is learned this way. For example one *can* give rules that distinguish between the diagnostic categories "alert" and "depressed". At some point, physicians recognized that it made practical sense to distinguish those states – they generalized from examples – and were able to make those states visible even to amateurs. Would that everything were so simple. Certainly much software is not.

Implications

The job of software development is to produce a product that people will pay money for, plus an organization that will be funded to build the next product. I'm assuming that it is impossible to communicate what the successful product will be, even were people forbidden to change their minds. What's needed is an iterative approach that allows frequent correction:

> "I think this is a satisfactory product."
>
> "Yes, so far as it goes, but it also needs *X*."
>
> "It now has *X*, so it's a satisfactory product."
>
> "No, that's not really an *X*. To make it an *X*, do this."
>
> "Now it's an *X*!"
>
> "It's a good enough one for now. Now it needs a *Y*."

Just as Dawn doesn't evaluate her student's preparation to evaluate a case, but rather the evaluation itself, a project team's output should be an actual working product at frequent intervals. It must be evaluated by the same person or people who will evaluate the final product.[5] This desire for iteration, unsurprisingly, fits with the Agile methods. (Unsurprisingly, because I'm an advocate for those methods.)

Legitimate peripheral participation has apprentices learn by starting with activities that are simultaneously simple and low risk. As they learn, the risk and difficulty grow together. I don't know, but I suppose that the complexity factor is about easing learning while the risk factor is about protecting the master's livelihood from the apprentice.

In Agile software projects, the emphasis is often on making the early iterations the most valuable. Each successive iteration should provide less value. When the next iteration would not be worth the cost, you stop the project. That's a risk-reduction strategy: it minimizes the risk that the project sponsor will get impatient and disrupt or cancel the work.

The lesson of legitimate peripheral participation is that the early iterations should also be easy. In a project that's starting from scratch, it seems fairly easy to put the valuable features in early. It's more difficult in legacy code, which suggests that the project must balance ease and risk. The project should also keep in mind that it's explicitly a learning project, that one output of each iteration should be the team's increased ability to make decisions pleasing to their "professor". It appears that Agile practice is learning how to strike a balance among many factors when scheduling; see, for example, the discussion in Cohn (in press).

The need to produce greater capacity from each iteration produces a question: is it better for schedules to be broad or narrow? Suppose a product has three constituencies. Which leads to most efficient learning of the business domain: satisfying each constituency in each iteration, thus getting knowledge of all parts of the domain, but knowledge of any given part at a slow rate? Or concentrating first on one domain, thus learning that part of the domain well but the others little or not at all? I don't know, so I change the subject.

In our field, we underrate examples. We – especially those of us with programming roots – love abstractions. Like the stereotypical mathematics text, with its repetitive sequence of definition-theorem-proof, we have a tendency to assume that those who come after us needn't follow the laborious process of coming to the right definition.

I suggest that people explaining business domains reduce the number of speeches that begin "A bond is a…" and increase the number that begin "Suppose you have $5,000 and you want to buy the simplest kind of bond. You *could…*" My preference for examples is to begin with step-by-step descriptions akin to use cases (Cockburn 2000), though I generally prefer more "implementation detail" than is considered wise in use cases. While I recognize that implementation detail that comes too early can lead to design decisions made thoughtlessly, there are counterbalancing advantages to detail:

- It's not uncommon for a completely arbitrary choice to reveal something about the business domain, especially when people choose outlandish, fun examples

[5] The more people who have influence over whether the product was worth the money, the harder that is. You quickly run into problems because you have to select mouthpieces for a large population. Nevertheless, the team will learn what's wanted faster and better if their finished work is evaluated more frequently.

(Buwalda 2004). As a simple example, someone thinking of an unusual example for a ZIP code field would surely use Canadian postal codes (which, unlike US codes, contain letters, and are used by people who get annoyed – albeit politely – when programs assume all the world's the US) or countries that have no postal codes (a problem if it's a required field).

- Anyone who's read about the handling of a sports car and then given one of those cars a test drive should know that you learn more – or perhaps just differently – from a test drive. That's especially true if you and the magazine reviewer don't already share common interpretations of words. There's a difference between talking *about* something (at a remove) and using it. The use of specific detail makes it easier for people to imagine use.

These sort of step-by-step examples are often used as tests, sometimes tests written before the code, in test-driven design fashion (Beck 2002, Astels 2003). I've come to believe that those kinds of tests are often a bad practice. What seems to work better is to use the collection of examples in what feels like a "boiling down" process, so that the resulting tests contain only those facts necessary for the most concise examples possible. (See Mugridge and Cunningham 2005.) Such tests are an interesting combination of abstraction and specificity: unlike normal requirements, they contain exact details, but only the details necessary for understanding.

When such examples are constructed in conversation (as they should be, so that people can ask questions), the examples themselves are likely the only thing that need be written down (on a whiteboard). When people refer to them later, it's likely they'll need a summary – probably abstract – of what the examples are all about. Those summaries might well begin "a bond is…" In that sense, I have a motto: *conventional requirements are merely commentaries upon examples.*

Ubiquitous languages as creoles

Another story, again based on personal communication:

> Ward 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. The team was taking longer to produce features, and they created more bugs.
>
> Much of the problem was due to a particular method (chunk of code) that was large and opaque.At some point, Ward's team made a concerted effort to simplify it by turning it into a *method object*. A method object is one that responds to a single command: "do whatever it is that you do".[6] A new kind of object has to

[6] 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

> 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 classes 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. Perhaps they already knew it was useful.
>
> Because it *was* 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. It seemed as if Advancers must correspond to something in the business world (must map to something "out there"). So the programmers asked the experts what Advancers were "really" called – but the experts didn't have a name. "Advancer" wasn't an idea that bond traders had. So the programmers kept the name and continued to figure out what it meant by seeing how it participated in program changes.
>
> For example, 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 in their instance variables. Switching to Advancers made tax reports tractable. Another gain in the team's capability, and a further understanding of what Advancers were about.
>
> 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 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.

This story reminds me of two related ideas from science studies. The first is what Star and Griesemer (1989) call *boundary objects*. They have several important properties:

- If *x* is a boundary object, people from different communities of practice can use it as what Chrisman (1999) calls a *common point of reference* for conversations. They can all agree they're talking about *x*.
- But the different people are not actually talking about the same thing. They attach *different meanings* to *x*. An Advancer to a bond expert is a novel way of talking

---

around refactoring. *Refactoring* talks of that, as do Wake's *Refactoring Workbook* (2003) and Kerievsky's *Refactoring to Patterns* (2004).

about the history of positions. To a programmer, it's a chunk of code that allows certain tasks to be done in certain ways.

- People use boundary objects as a *means of coordination and alignment* (Fischer and Reaves 1995). Advancers are a way for business people to tell programmers what to do with increased confidence that they'll be pleased by the results. They allow different people to *satisfy different concerns simultaneously*.
- Despite different interpretations, boundary objects serve as a *means of translation*. If it becomes important that a programmer understand more about bond trading, the business person can use Advancers to create telling examples.
- Boundary objects are *working arrangements*, adjusted as needed. They are not imposed by one community, nor by appeal to outside standards (Bowker and Star 1999).

Star and Greisemer are using physical objects as an analogy. Galison (1997) uses language. In his study of how experimental and theoretical physicists work together, he describes them creating what he calls "creoles" by analogy to the trading languages developed at shared boundaries of cultures.

Galison adds, I think, an extension to the semi-common project practice of creating what Evans (2003) calls a *ubiquitous language*. Such a language is composed of nouns and verbs spoken by the project team and also found in the text of the program (as class and method names). Those nouns and verbs allow the same coordination as boundary objects do.

Galison steers us away from thinking of the ubiquitous language as being discovered in the business domain. Instead, it's mutually created, over time – just as Advancers were. And, like Advancers, the words can come from either domain – the business domain or the programming domain. What matters is successful coordination and effective learning.

In a way, creoles take us full circle. No matter what really happens within brains, the ubiquitous language lets everyone involved in a project make a shared (enough) map of the world. It's not transmitted down a conduit from person to person – it's created word by word, example by example, as people converse and correct.

References

Ashby, Ross. (1948) "Design for a brain", *Electronic Engineering*, 20 (Dec 1948), 379-83.

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

Austin, J.L. (1975). *How to Do Things With Words*. (2/e)

Bach, James. (1999) "Heuristic Risk-Based Testing", *Software Testing and Quality Engineering*, November.

Beck, Kent. (2002) *Test-Driven Development: By Example*.

Bloom, Harold. (1997) *The Anxiety of Influence: A Theory of* Poetry.

Bowker, G., and S.L. Star. (1999) *Sorting Things Out: Classification and its Consequences*

Buwalda, Hans. (2004) "Soap Opera Testing", *Better Software*, February.

Chrisman, Nicholas. (1999) "Trading Zones or Boundary Objects: Understanding Incomplete Translations of Technical Expertise", 4S San Diego, 1999.http://faculty.washington.edu/chrisman/Present/4S99.pdf

Cockburn, Alistair (2000) *Writing Effective Use Cases.*

Cohn, Michael. (in press) *Agile Estimating and Planning.*

Culler, Jonathan. (1983) *On Deconstruction: Theory and Criticism after Structuralism.*

Daniel Dennett. (1992) *Consciousness Explained.*

Derrida, Jacques. (1988) *Limited Inc.*

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

Fish, Stanley. (1980) *Is There a Text in this Class?: The Authority of Interpretive Communities.*

Fischer, G., and B.N. Reeves. (1995) "Creating Success Models of Cooperative Problem Solving", in Baecker et. al. (eds), *Readings in Human-Computer Interaction: Toward the Year 2000.*

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

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

Gombrich, E. H. (1995) *The Story of Art.* (16/e)

Kerievsky, Joshua. (2004) *Refactoring to Patterns.*

Lakoff, George (1990) *Women, Fire, and Dangerous Things: What Categories Reveal About the Mind.*

— and Mark Johnson. (2003) *Metaphors We Live By.* (2/e)

Lave, Jean and Etienne Wenger. (1991) *Situated Learning: Legitimate Peripheral Participation.*

Leary, T., G. Litwin, and R. Metzner. (1963) "Reactions to psilocybin administered in a supportive environment." *Journal of Nervous and Mental Disease*, 137.

Mugridge, Rick and Ward Cunningham. (2005) *Fit for Developing Software: Framework for Integrated Tests.*

Nilges, Edward. (in press) "Deconstruction for Programmers". To be published in the 2005 *OOPSLA Companion.*

Pickering, Andrew (in press). *The Cybernetic Brain in Britain.*

Reddy, M.J. (1979) "The conduit metaphor – a case of frame conflict in our language about language." In A. Ortony (ed.), *Metaphor and Thought.*

Rorty, Richard. (1981) *Philosophy and the Mirror of Nature.*

Star, S.L., and J.R. Griesemer. (1989) "Institutional Ecology, 'Translations', and Boundary Objects: Amateurs and Professionals in Berkeley's Museum ofVertebrate Zoology 1907-39", Social Studies of Science, Vol. 19.

Wake, William C. (2003) *Refactoring Workbook.*

Zinberg, Norman. (1986) *Drug, Set, and Setting.*