Five Reasons to Fire your IT Department
(Or not hire agile developers)

Dr. David West
School of Business – New Mexico Highlands University
Department of Computer Science – University of New Mexico

Abstract: Agility is an old topic in the world of business. One of the major impediments facing an organization striving to be agile is information technology. It might appear that the burgeoning Agile Development movement is good news for the corporate world but only if those in that movement recognize and address with greater clarity and emphasis the ways that traditional development has failed. This paper outlines five factors that make IT a barrier rather than a resource. CEOs and managers should be asking agile developers how they intend to address these issues before making any adoption decisions.

Background

The world of business has been talking about agility for at least two decades. Popular and widely read (at least widely sold) works by Eric Toffler, Tom Peters, and many others argued vociferously for the position that business success was dependent on the ability to be agile, on rapid change, adaptation, and evolution – to "thrive on chaos."

Computers and information technology were thought to be the means whereby an organization could realize the goal of business agility. Alas, those hopes appeared to have been misplaced. Despite billions (over $500 Billion annually by 1996) of dollars spent on IT, few organizations are any more agile today than they were twenty years ago. Perhaps this should not be much of a surprise, given the almost unchanging (since declaration of the "software crisis" in 1968) statistics on the number of IT projects abandoned, over budget, late, and generating marginally useful results. (Or the oft debated lack of productivity gains documented by Robert Solow in the 1990s.)

It is not the case that IT has not tried to improve its track record. Apparent investments in new ideas (objects), new methods (Software Engineering), new concerns (quality and Total Quality Management), and new tools (UML, Java) are clearly evident. Today business managers are confronting the need to make decisions about another software development innovation – agile development. Is there any reason to expect that any of the various flavors of agile development will, in fact, deliver the means for an organization to achieve its own agility goals?

A possible answer to that question can be derived by examining five factors that have contributed to the failure of traditional software development to achieve its stated objectives. These five factors derive from ideas (assumptions and values) – they are not specific, easy to change, techniques or technologies. The business manager's evaluation

of the potential for success offered by agile development should be based on an analysis of how various agile methodologists and developers recognize and avoid the traps implicit in these five philosophical areas.

Thinking like a computer

Talking about the way he was taught to program, David Parnas recalls: (paraphrased) 'Think like a computer,' he said. 'Begin by thinking about what the computer must do first and write that down, then think about what the computer has to do next and continue in that way until you have described the last thing the computer must do.' This remains the way that almost all programmers and developers are taught to code and to design systems – think like a computer. Unfortunately, it does not work – at least it does not lead to the construction of software that can support organizational agility.

A consequence of developers thinking like computers is a communication gap between them and users. This gap is apparent and is wide. Immense amounts of effort – mostly misdirected – has been expended trying to bridge that gap with formal requirements definitions and highly legalized contractual relationships that help affix blame for miscommunication but do nothing to prevent it in the first place.

At first glance, it appears that agile methods have addressed this potential communication gap. Extreme Programming, for example, requires an "on-site customer" to assure continuous dialog as a system is being developed. But physical proximity and willingness to talk will not eliminate the communication gap – which is based on one party "thinking like a business" and the other "thinking like a computer."

A common way of thinking needs to be adopted. Since, as Parnas noted 20 years ago, thinking like a computer does not even help one become a better programmer, it is time for developers to abandon that unfruitful approach and adopt something more natural, something more consistent with the user's way of thinking. Eric Evans in one of the more recent voices advocating this type of thinking about development and design.

Even more fundamental are the metaphors and ideas that guide the division of the world into parts, parts that are easier to understand and interact with than the whole, parts that are composable, so that the same parts can be combined in different ways to solve different problems. The human "parts" that the businessperson works with every day have exactly this kind of composability.

The idea of an "object" as a unit of decomposition is precisely the kind of metaphor and idea required. But the objects must be defined consistent with "natural occurring disjunctions in the problem space" and not artificially defined disjunctions in the solution (computer thinking) space. Alan Kay, Kent Beck and Ward Cunningham, Rebecca Wirfs-Brock and others first articulated this kind of object idea. Dave West argues for its adoption in his book on object thinking.

Query your IT staff (and any agile practitioners you are thinking of hiring) and see what their conception of an object is. If they describe it in a way that you can understand, in a way that is reminiscent of an object-as-person metaphor then there is some hope. Odds are both groups will tend to answer you in terms of one of the two bedrock ideas of thinking like a computer: algorithms (telling the computer what to do) or data structures (what the computer is to do it to).

Agility does not, in and of itself, offer any assurances about how people think about the problem of constructing software. Agile developers can be just as guilty of thinking like a computer as the most traditional of practitioners.

Artifact Construction

Software developers, especially those most closely associated with the ideas of structured development and software engineering, have focused their attention almost exclusively on the construction of artifacts. Lip service is paid to the notion of context – of a system of interaction among human beings and other hardware and software artifacts – but the main focus of concern and interest is defining a "thing to be constructed" as thoroughly and unambiguously as possible and then "building to definition." If the product "meets specs" it is a success – it is not the developer's problem if it is unused because it makes people unhappy or disrupts the socio-political balance of forces in the office.

The delivery of new software to an organization is highly reminiscent of the tragic-comedic events depicted in a movie, *The Gods Must be Crazy*, released several years ago. In the movie, a Coke bottle (an artifact) was dropped from a plane and discovered by a small group of *!Kung* bushmen. It was found by some to be useful, by others to be harmful. It disrupted family relationships and lines of authority. Things got so bad that a consensus decision was made to remove the bottle to a place where it could never be recovered.

Christiane Floyd and her colleagues, Bonnie Nardi and Vicki O'Day, and many others have pointed out the need to "socialize" our software. Floyd and her colleagues use the notion of software development as reality construction – creating a new physical, social, and even cognitive environment in which people have to live – with the introduction or modification of each new software artifact. Nardi and O'Day offer insights on the social structure and human roles that need to be established if any new technological artifact is to be successfully deployed in an existing complex, human inhabited, system.

Agile development has adopted the necessary foundations for a development process that is not centered on the construction of an artifact. Incremental iterative development with lots of feedback is a prerequisite. But again, it is not sufficient. Artifact centrism can be just as evident – one story card at a time – in agile development as it has been in traditional development.

Ask your IT staff (and any agile practitioners you are thinking of hiring) to describe for you the "affordances" of the software they are creating. If you get blank looks arrange

for a remedial course in Hiedeggerian philosophy. If you get an informed discussion of how people perceive and interact with the tools they use, with the occasional mention of Floyd, Pelle Ehn, Kristen Nygaard or the "Scandinavian School" of software design; you have a potentially great development team.

Formal Methods

The terms 'art' and 'craft' were banished from the world of software development decades ago. Instead, software was to be developed in a "scientific," "mathematical," "logical," and "engineering fashion. Even the process by which software development activities was managed was to be scientific and rational.

The agile literature is full of arguments against the formal mindset. Interestingly enough, so to is the literature of traditional software development. Leading traditionalists like David Parnas and Fred Brooks pointed out decades ago that method, process, and tools are far less important than people and people do not work according to formal methods. Robert Glass discusses this issue at length in his numerous books and columns.

But formalism is not a characteristic unique to software development. Western culture in general has been besotted with formalism since the "Age of Enlightenment". Rational is good, irrational is bad. Formal is necessary, informal is inadequate. Anything not science is superstition. Anything, including agile practices and methods, that comes into existence in this cultural context cannot help but be contaminated by the prevailing ethos.

It is astounding that so much effort – by the agile community itself – is being expended on efforts to formalize agility. From discussion about the need to test a particular kind of code, to debates about whether you can omit or add a practice and still be XP, to concerns about certification – too many people are attempting to remove all ambiguities and reduce agility to a predictable and, yes, formal way to develop software.

The test for your IT staff (and prospective agile staff) this time is to define what Kent Beck meant when he suggested that XP had three phases – "out-of-the-box, adaptation, and transcendence." At minimum you should get a response using an analogy like playing chess – by the book, a move at a time; using patterns adapted to current circumstances, and internalized gestalt awareness of the right move to make at each juncture of the game. An even better response would be based on how self discipline and internalization of the values, principles, and practices of XP create a mindset reminiscent of satori (enlightenment) that allows a deep understanding of each situation and recognition of the correct action in each instance.

Centralization and Control

Business has a love-hate relationship with the concepts of centralization and control. Software developers, it seems, have an unbridled lust for both.

Even though many alternatives are known, the typical business organization is organized according to a hierarchical control structure. It is also typical to see companies establishing "headquarters" where all employees can be centralized and better managed (controlled). At the same time, most successful business recognize that the need to escape the confines imposed by centralized control – especially when the business goals involve adaptation and innovation – agility.

Software developer's love of centralization and control is evident in the push for "integrated" applications, monolithic systems promoted by companies like SAP, and most prominently in the reverence shown for relational database systems. Even at the level of code, look for class names that include the word "manager" or "controller" or the extensive use of case statements and similar control structures.

There was a time, perhaps, when centralization and control were necessary for software – fifty years ago when computers where massive, slow and expensive. Today, the world of the Web and portable wireless platforms offers a powerful argument against both centralization and control. The coming world of ubiquitous computing will make this argument even more compelling.

The ends to which centralization and control are directed can be better achieved by communication and coordination. The most powerful of all the values and practices incorporated in agile methods are those directed towards communication. A shared vision (metaphor), a shared language, and constant use of both are more than adequate substitutes for centralization. (As long as you remember that human communication is dependent on the occasional exchange of pheromones – requiring close physical proximity - as it is on incessant email.)

Coordination can be effected by distributing "control" responsibilities across a community of participants instead of centralizing into a single controller. A traffic signal, for example, coordinates but does not control the flow of automobiles through an intersection. The traffic signal controls itself – changing states in an appropriate manner – and the automobiles (drivers) control themselves in response to broadcast communication by the traffic signal of its state.

If your IT staff (or prospective agile team) compromise or inhibit (like the all to common practice of forbidding one part of the IT group from communicating with others, including users and managers) the flow of communication then send them to the boardroom where Donald Trump can fire them. You can also ask a technical question – what pattern do you use most often in your code. If you get any other response than "the observer pattern (also known as "publish and subscribe") then be suspicious, they are probably still too enamored of hierarchical control.

Scale

Humans do have a fascination with big things. Businesses do strive for continual growth. Bigger is, if not better, often necessary. A coordinated and standardized air traffic

control system (and accompanying software) is a big task – one that is desirable to undertake. Problems with bigness arise because we not only desire it in many ways, we are also afraid of it. Bigger things are harder to understand and we are afraid mere mortal humans will fail in the attempt. We are also afraid because we do not understand how things get to be big – or at least how they can get big and still work instead of collapsing as a consequence of their bigness. Uncontrolled growth is described as "cancerous."

Software developers frequently confront issues of scale. A harsh criticism of a solution to a software problem is the comment, "but it doesn't scale." The larger the system under development the more pressure there is to adopt extremely formal and bureaucratic techniques and management strictures in order to assure success. Agile methods, and especially the highly prominent variant extreme programming, are attacked and dismissed because they "can't scale."

It is fear not evidence that supports the claim that agile methods do not scale.

There is no formal process for growing a business, why should there be one for "growing" the software it requires? Businesses grow by taking on one challenge at a time, by hiring one person at a time. To grow successfully an organization needs a vision – a mission statement in prosaic terms – and a set of values to guide that growth. It does not need a "plan," "architecture," "process," or "maturity level." (At most, the vision and values are augmented by the existence of an espalier (to use a term advanced by David Gelernter) – a "training form" of the sort used to encourage the growth of roses to fill a space.)

Software is written one line at a time by a single pair of programmers. Whatever the scale of a project its creation requires small teams working independently and constantly communicating. All large-scale software is created "organically" in a manner consonant with agile values and practices.

What does not scale is the ability to pre-plan, pre-design, and mechanically execute those plans and designs. Businesses do not have "hive minds" of the sort popularized in science fiction, directing the growth of the company nor does a large software development team.

Successful business growth is an emergent phenomenon – the eventual outcome is unpredictable from perfect knowledge the operative, and local, techniques, constraints, and initial conditions. The outcome is always a bit of a surprise. If we use business reality as an analogy for software it means our software systems will also emerge and their final form will be a bit of a surprise.

But the goal of software is to support and augment the natural abilities of the humans using that software in the environment they occupy. The goal of development should be continual parallel evolution – not the construction of some pre-conceived artifact. This is especially true for large scale systems supported large organizations and socio-political communities.

Agile methods provide the foundation for precisely this kind of parallel evolution of software and organization. Both client (user, organization, society) and server (software) co-evolve in an incremental and iterative fashion.

The final question for your IT staff (and prospective agile team) involves the value of courage. Do they have the courage to "walk the talk?" Evolvability, adaptability – the ability to respond to change – are espoused values for software engineering as well as for agile methods. Does your staff (your prospective agile team) truly embrace change?

Conclusion

Agility is a goal (a requirement really) for both business and software development. Traditional approaches to software have failed to deliver on their promises – especially those involved evolution, adaptation, and close support of business requirements. Agile methods are the most recent expression of approaches known to deliver the promises made but not delivered by traditional methods.

There is no guarantee that agile methods, however fervently advocated or practiced, will have a long-term track record better than traditional approaches. At least five reasons for questioning the ability of agile methods to deliver have been presented in this paper. Each reason is based on a criticism of traditional development attitudes and values – attitudes and values that can corrupt agile development just as effectively as any other type.

Two challenges conclude this paper. The first, to the agile community: consider and respond to the concerns raised above, and articulate how agile values, principles and practices can assure a different outcome this time around. The second to the business community: reflect on the essence of business and how that essence is or is not consistent with your relationship with your IT organization. If you find it wanting – probably for one or more of the aspects of development discussed in this paper – take action to realize the agility you claim to strive for in both your business and your means of IT support.

References

Dittrich, Yvonne, Christiane Floyd, and Ralf Klischewski (eds). Social Thinking – Software Practice. MIT Press. 2002.

Evans, Eric. Domain Driven Design: Tackling the Complexity in the Heart of Software. New York: Addison-Wesley. 2003.

Floyd, C., H. Zullighoven, R. Budde and R. Keil-Slawik. Software Development and Reality Construction. Springer Verlag. 1992.

Glass, Robert. L. Facts and Fallacies of Software Development. Addison-Wesley. 2003.

Parnas, David Lorge. "Software Aspects of Strategic Defense Systems." *American Scientist* 73 (1985) PP. 432-440.

West, David. *Object Thinking*. Redmond, WA: Microsoft Press. Microsoft Professional. 2004.