Looking for Love
(in all the wrong places)

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

Abstract: Onward! Seeks future directions. The Feyerabend project seeks to redefine computing. Thoroughly disillusioned with traditional and contemporary practices, a significant number are looking for a software development profession that we can love. If a person has a history of repetitive (only the names change) unsuccessful relationships they are advised to look at themselves in order to discover the roots of the problem that leads to them making the same mistaken choices over and over again. The same advice can be given to our profession – look inward to discover the true source of our problems. This paper introduces five factors that arise from reflective examination of computing and software development.

Background

Once upon a time (circa 1968) fear ran rampant. Computers and computer-based systems revealed their fatal attraction and the world was hooked. Although the degree of dependency was not yet obvious, it was clear that computing would dominate every aspect of future commerce. Alas, our collective ability to create software of sufficient quantity and quality was obviously suspect – instant "software crisis."

Thirty-six years and trillions of dollars later statistics on the number of software development projects abandoned, over budget, late, and generating marginally useful results have not changed to any significant extent. The software crisis continues. Yes it is true that we are surrounded with software and much of it is of the highest quality but the "good stuff" still represents a small portion of the total amount of software deployed and a still smaller amount of all attempts to create software.

It is not the case that computing has not tried to improve its track record. Apparent investments in new metaphors (biology, horticulture), new ideas (structure, objects, aspects), new methods (Software Engineering, Agile development), new concerns (quality and Total Quality Management), new processes (Capability Maturity Model), new environments (.Net), and new tools (UML, Java, Eclipse) are clearly evident. But these efforts too, have failed to produce any kind of crisis resolution.

From the inside, as a software professional or computer scientist, it seems as if great progress is being made, even if we have not yet reached our goals. From the outside, however, it appears that the professions is in an infinite loop – it keeps repeating the same cycle of innovation and disillusionment over and over, changing only the vocabulary. Personal computing did not learn from and evolve from mainframe computing – it merely recapitulated earlier knowledge and created "mini-Me" mainframes on every desktop. Agents and aspects are not advances from objects – merely a re-expression of ideas that were present in the early definition of objects and object decomposition.

In the world of human relationships a similar pattern is observed. The tendency of an individual to select partners that are, to an outsider, obviously nothing more than a variation on the last unsatisfactory and dysfunctional partner is well known. Equally well known is the fact that the cycle will continue to recur until the individual making the selection reflects on the reasons behind their choice.

Do the world of computing and the profession of software development share, and suffer from, an analogous problem? Are attempts to find new directions and reinvent computing doomed at the outset? Would a bit of self reflection reveal internal assumptions and values that predetermine our overt choices and actions in such a way that we cannot avoid repeating old mistakes yet again? Are we looking for answers in all the wrong places?

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 agenda of Onward! and efforts like the Feyerabend Project can be advanced by recognizing and avoiding the traps implicit in these five factors.

One: Thinking like a computer

David Parnas once described the way he was taught to write software. "Think like a computer," his instructor told him. "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 write that down. 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.

Yes, it is true that we no longer write one expression at a time – as Parnas was instructed to do – but we still think and solve our problems from the perspective of what it is that the computer must do. Grady Booch, in the August 2004 issue of *software development*, suggests we have made great advances because we are using higher and higher levels of abstraction to accomplish our work. Unfortunately, those abstractions continue to be of the machine and our thoughts of how to write programs is still being done in terms of what we think the machine must do next.

Thinking like a computer has several consequences:

First, there arises an unavoidable communication gap between developers and everyone else. This gap is readily apparent and is extremely wide. Immense amounts of effort 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 next to 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 high volume discussion is, at best, an indirect means for solving the communication problem.

Since, as Parnas noted 20 years ago, thinking like a computer does not even help one become a better programmer, it is not unreasonable to expect developers to abandon their artificial language and adopt something more natural, something more consistent with the user's way of thinking.

Second, the computer offers a vast expanse of infinite potential – a true *tabula rasa*. The computer provides an infinite "solution space" consisting of enumerable equivalent (in terms of matching an initial input to an ultimate output) alternative algorithms capable of solving a given problem. Even more problematic, there are no objective consistent criteria for determining which of the alternative solutions is "better." (While it is true that judging criteria can be established and algorithms scored according to how they meet those criteria – this merely begs the question. Why those criteria and not others?

A developer engaged in thinking like a computer has an open ended infinite tree of choices in front of her for each succeeding step in the program. Which of the multitude of options should be selected? There is no real answer to this question. Instead computing has relied on best practices (Programming Pearls – a multi-volume primer on algorithm construction is an exemplar), idiom and convention, arbitrary standards, and more recently, patterns.

The recent experience with patterns exemplifies the problem. Because the vast majority of those in the patterns community are seeking "patterns for thinking like a computer" -instead of an Alexandrian pattern language – the number of patterns is essentially equivalent to the number of algorithms, i.e. infinite. Patterns quickly become part of the problem instead of part of the solution. Design patterns, especially, merely document one idiosyncratic path through the maze of potential solutions.

"Better" abstractions merely provide new flashy facades that attempt to hide the complexity and ugliness of implementation behind a shiny new front. It is difficult, if not impossible, to use most such facades (whether in the form of compiled run-time libraries, or MDA) without a thorough understanding of what is behind them. The facades necessarily embody assumptions about the solution space, and those assumptions are seldom totally congruent with those made by the developer when using the facades.

The road not taken: several individuals over the past fifty years have suggested alternatives to "thinking like a computer." Alexander - both in *Notes on the Synthesis of Form* and again in *A Pattern Language*, Parnas in his essay On Decomposition, the Simula team (before Simula was corrupted into a commercial programming language), and the advocates of behaviorally defined objects like Kay, Beck, Cunningham, and Wirfs-Brock.

A Focus on Artifacts

Traditional (and most agile) software developers tend to focus 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 hardware and software artifacts – but the main focus of concern and interest is defining a "thing to be constructed" as unambiguously as possible and then "building to definition" - satisfying requirements. (Agile developers differ from this description, usually, only by executing the definition-build cycle in smaller increments and allowing greater latitude for a macro design – architecture - to "emerge".) If the artifact "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 tragicomic events depicted in a movie, *The Gods Must be Crazy*, released several years ago. In the movie, a Coke bottle (an artifact) was carelessly discarded from a private 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. There was no understanding of the artifact, its intent, its make-up. There was no way to modify the artifact that did not break it. Eventually, the artifact had to be discarded.

Christiane Floyd, Bonnie Nardi, Vicki O'Day, Luke Hohmann, Larry Constantine, Jerry Weinberg, John Seeley Brown and many others have pointed out the need to "socialize" our software – to put the artifact in context – and yet we insist that our job is merely to produce artifacts. (Sometimes those artifacts are working software, most of the time they are models and other forms of documentation.)

Peter Naur wrote a provocative article several years ago on "Programming as Theory Building," in which he suggested that development was flawed because it adopted a "production" model of the process of software development. We insist on thinking about our jobs as if they were manufacturing – the production of various artifacts.

The road not taken: Naur's ideas of theory building which are echoed in numerous post-modern treatments of software (like the work of Floyd, et. al.). The work of Lakoff on metaphor and embedded metaphor suggest several paths to explore software development from different perspectives than the prevailing manufacturing metaphor.

Formalism

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 of software development and management should 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. They also pointed out the obvious fact that 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.

It is time for software development to abandon the narrow concept of formalism and begin exploration of the ineffable that is expressed in art. It is time to explore "requirements" in terms of the purely qualitative (e.g. Clifford Geertz's ideas about "thick description") instead of the quantitative and formal straightjacket imposed today. It is time to figure out what Alexander meant by passing through the gate and what Kent Beck meant when he described the "third phase" of XP as "transcendence."

Bacon's Mistress

Fairly or not, Francis Bacon is blamed for creating a scientific method using "misogynist metaphors" and "treating Nature as a "mistress in need of control." Whatever its origins, and I suspect they long antedate Bacon, control is a dominant metaphor in software. Centralization is less dominant, but no less pervasive. Software developers, it seems, have an unbridled lust for both control and centralization.

The epitome of structured development was the hierarchical top-down control architecture of the program structure chart. At the top of the pyramid was the central "master" control module. Below and subservient were afferent, transform, and efferent modules. Information flowed up and then down the hierarchy dispatched according the best judgment of the higher up control modules. Control (almost always) flowed down the hierarchy. This love of central control is still pervasive in the "main plus subroutines" architecture mandated by most currently popular programming languages (like Java). Even ostensibly object languages like Smalltalk could not eliminate the vestiges of control - the "Controller" being one of the three legs of the MVC architecture at the heart of the original Smalltalk environment.

Software developer's love of centralization and control is also evident in the push for "integrated" applications, monolithic systems promoted by companies like SAP, the ubiquitous relational database management system, and even packages like Microsoft Office. At the level of code, class names that include the word "manager" or "controller" or the extensive use of case statements and similar control structures reveals the assumed need for centralized control.

There was a time, perhaps, when centralization and control were pragmatically justified - e.g., fifty years ago when computers where massive, slow and expensive. An argument might be made that centralized control are pragmatic necessities in the design and construction of dense integrated circuits. But in terms of application software, today, those arguments are highly suspect. In fact, 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. A common language (one based on thinking like an object as noted above) provides the necessary means of communication. Relying on messaging to provide channels of communication is a natural consequence of thinking like an object – one that can essentially eliminate the need for direct control so popular in today's software. (Popular, only because it is more "efficient" in terms of machine utilization.)

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.

It is not necessary to create a single monolithic "integrated" program in order to put a spreadsheet in a document. Hypertext is a clear alternative with essentially equivalent results and an orders-of-magnitude simpler implementation.

Communication and coordination are dependent on standards whether the parties involved are humans working together using a natural language, or software objects sending messages in order to collectively complete an assigned task. The standards, however, should be derived from the natural world. Unfortunately most standards employed today reflect "thinking like a computer" and are therefore arbitrary in nature and adopted more for reasons of competitive advantage than communication.

Scale

Humans do have a fascination with big things. Businesses do strive for continual growth. "Bigger is better," "mine (company skyscraper, salary) is bigger than yours (company, skyscraper, salary)," 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 not only from our desires but also from our fears. Megalophobia is found in most aspects of software development. We justify our fear, partly, by noting the obvious fact that human beings are somewhat limited in their cognitive abilities. It is "necessary" to decompose large complex systems into simpler ones. No one can deal with a large and complex system in his or her head. Didn't Miller prove his "magic number seven plus or minus 2?"

The trouble with this justification is that it ignores a huge body of counter-evidence. Human beings like complex things, deal with a dynamic and highly complex environment every day of their lives. Readers of Neal Stephenson's *Cryptonomicon* juggled, in their heads, a lot more than seven plus or minus two details. It is true that bigger things can be harder to understand and mere mortal humans might fail in the attempt. But these are possibilities not certainties.

Software developers frequently confront issues of bigness, *aka* 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." Unfortunately, in this sense of scaling – nothing scales – just ask any general how often extensive war plans survive actual battles.

There are numerous alternatives for dealing with scale. One is to abandon the metaphor of construction in favor of horticulture. Grow software instead of engineer it. If insecurities persist regarding the eventual "big thing" to be grown adopt the least intrusive means of guiding the outcome – use an espalier as advocated by David Gelernter.

While it is true that the volume of communication required for fifty people to work cooperatively is greater than that required by five – this does not mean a qualitatively different kind of development is required. It means that a qualitatively different means of communication is

required. Happily we have an existence proof of just such a communication medium – culture. We need to understand a lot more about culture and how it works to effectively coordinate large populations. One possibility is to explore how "culture" serves as a ubiquitous and dynamic "blackboard" space used by individuals as a source of clues about individual decisions and actions – all without the imposition of any overt, formal, defined process.

Conclusions

Perpetuation of the status quo in computing and software development is not an option. Continually pursuing the "next big thing" which turns out to address one of Brooks' "accidental" factors that make software hard is a waste of time.

Only reflective examination will reveal the roots of our discontent. Once revealed, we can pursue strategies for circumventing them.

Unfortunately, abandonment of cherished assumptions and values will be required.

Fortunately, we have already discovered many of the seeds from which a new discipline of software development might develop: iterative incremental exploratory development; simulation, behavior defined objects, decomposition based on natural disjunctions in the problem domain, and several of the agile practices are but some of them.

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.