Enhancing DDD?

David West

In celebration of the fifteenth anniversary of Eric Evans’ *Domain-Driven Design*

I would like to communicate two things in this essay: first, the profound contribution that Eric Evans has made to software development; and second, pose some questions that might lead to explorations and possible enhancements. Before doing either of those, it is useful to provide some context.

**A Bit of History**

Computers were new then, and inspiring. Vannevar Bush, *As We May Think*, imagined a “Memex,” a hypertext type device to augment research and thought. Douglas C. Engelbart wrote of *Augmenting Human Intelligence*. Alan Kay envisioned the *Dynabook*. Steve Jobs (not a contemporary) had a vision of *a bicycle for the mind*.

Computers are ubiquitous today, and perplexing. What happened to all that potential? How did visionary utopias morph into Facebook as the exemplar of the advertising dominated dystopia described by Frederic Pohl and C.M. Kornbluth in *The Space Merchants* (circa 1955)?

Software was new then, and valuable. COBOL programs of 1-5,000 lines that took a few weeks to conceive, implement, and deploy reduced costs, or increased profits, by hundreds of thousands of dollars. Slightly larger software systems created strategic advantage worth millions. Work was enhanced by software that automated its most tedious and mundane aspects, allowing the worker to utilize her innate human abilities to a greater extent.

Software is pervasive today, and costly. Massive, convoluted, and complicated software inhibits business agility and innovation. Trillions of dollars are spent annually, ninety-percent of it on maintenance of legacy code, on software that is, at best, a commodity. Work is increasingly demeaning to the humans doing it as they are reduced to mere ‘machine parts’ with no freedom to think or act independently of how the computer instructs them.

Addressing all the whys and wherefores of how dystopian futures replaced utopian is beyond the scope of this essay, but how software and software development devolved is more tractable.

Let us begin with the LEO I — the world’s first business computer.

LEO, (Lyons Electronic Office), was built by J. Lyons and Company, a leading catering and food manufacturing company in the UK, in 1951. The same team built the hardware, programmed system software, and programmed a set of applications that included: payroll, order entry, inventory control, production scheduling, and management reports. The computer was even used to create customized tea blends — a rudimentary “expert system.”

A number of unexamined assumptions were made while constructing LEO that contributed significantly to the problems with software development that bedevil us today. Among them:

* Black Box programming. Programming was defined exclusively in terms of the computer, the machine. A precisely defined set of inputs entered the black box and a precisely defined set of outputs emerged. The programmer had to envision all the states and state transitions — within the box — that assured the correct correlation of inputs and outputs. Several decades later, Fred Brooks, in his “No Silver Bullet” paper, identified the challenge of mentally keeping track of all these states and transitions as THE essential problem of software development.
* Integrated, monolithic, systems. This one would be hard to avoid, given that there was only one computer available.
* Centralized hierarchical control. Epitomized by the ubiquitous ‘Program Structure Chart’ with a single master control module in charge of subordinate afferent, transform, and efferent modules.

The most egregious, albeit understandable, assumption concerned the domain; which was assumed to be no different, in essence, than the computer itself. That is to say that the domain — clerical tasks — was assumed to be a deterministic mechanical system of the same type as the computer itself. When the domain is purely clerical tasks, this assumption is not unreasonable. For the next 15-20 years, the era of “Data Processing,” few software development efforts challenged the assumption.

Little, if any, thought was given to a domain, some subset of the natural world (e.g. a business or business process). It was not until the early seventies when the era of Data Processing gave way to the era of Management Information Systems that any attention was paid to the domain.

In the early 1970s the ideas and practices of “Structured Analysis and Design” (SAD) dominated the practice of software development. SAD advocated:

* Step One: model the domain
* Step Two: determine what changes you wanted to enhance or correct issues in the domain.
* Step Three: brainstorm multiple ways in which the changes might be affected.
* Step Four: analyze which ‘solution’ was optimal.
* Step Five: model the chosen solution — including modularization of the software.
* Step Six: implement the model.
* Step Seven: deploy and evaluate the implemented solution and its impact on the domain

Seldom, if ever, were Steps One through Four actually performed. In part, because business management failed to see the value and therefore discouraged or forbid them as a “waste of time.”

The most important factor in the demise of the domain was “Software Engineering.” In 1968 a new profession and new academic discipline were defined. Software engineering was to be applied Computer Science in the same way that Structural engineering was applied math and physics.

To be a professional software engineer you needed to know everything possible about the computer. Programming was nothing more than the applied science of “algorithms plus data structures.” (Dykstra’s famous definition of programming.)

A professional software engineer began work with a set of “complete and unambiguous” requirements and then built a piece of software that provably satisfied those requirements. All of the tools and techniques utilized by the software engineer were focused facilitating the programmer’s understanding of what was going on inside of the computer and/or structuring and inter-relating software modules within the context of the computer.

By the mid 1970s, the Domain was Dead.

**Life Underground**

Only to the Mainstream was the Domain lost.

In 1985, Peter Naur tried to tell us why and how the domain was critical to software development. Railing against the prevailing “ software engineering production model” of software development, Naur insisted that developers were engaged in a process of theory building. A theory of, *“****an affair of the world*** *and how the software will handle and support it.”* [emphasis mine]

Lost in the rush towards Object-Oriented Programming in the 1980s and 1990s were critical aspects of the Object Idea. In OOP, objects were nothing more than animated data structures, a means of modularizing a programs source code. In contrast the Object Idea was to use the criteria of “responsibilities” to modularize and understand the Domain and to create a common vocabulary, a common ontology, that would map domain modules to software modules.

User stories, ala Kent Beck and Extreme Programming, were yet another attempt to bring the Domain back into the development process. “On-site Customer” and the practice of “Exploratory, Iterative, Development” brought the Domain front and center while simultaneously providing practices and principles that supported Peter Naur’s vision of theory building.

Challenges presented by “Ultra-Large Scale Systems” and “Complex Adaptive Systems” have taken center stage in the past decade. As our understanding of these systems increases, so to is the conviction that natural, Domain, systems are qualitatively different from the simple deterministic system of computer plus software. The assumption made with the building of LEO — that the domain was just as deterministic and simple as the computer — is false.

These, and numerous other, advocates for the domain were, seemingly, ignored by mainstream developers. Practitioners made the attempt, but barriers like preventing programmers from talking to users — only business analysts could do that, and they could only talk to system architects — kept the domain at bay.

And, of course, for software engineering the domain was irrelevant. Users provided specifications / requirements and developers coded to satisfy requirements. Job done.

**Comes the Dawn**

My first impression, opening Eric Evans’ *Domain Driven Design*, was pleased amazement that someone was actually paying attention and was aware of the underground existence of the Domain.

Martin Fowler’s Introduction and Eric’s own words assert that a robust domain model is essential for the success of any software project. They even asserted that there was an entire community within software development that understood the essential value of domain knowledge and domain models.

Aaah, but how to conceive and construct such models?

Eric proceeds to provide us with principles and practices, techniques and tools, to do just that. Readers of this essay will already be familiar with the contents of the original book and subsequent developments of DDD, so it is not necessary to attempt a recap of specifics here.

The amazing thing about *DDD* was not the patterns or the practices, it was the quiet way it put the lie to a fundamental tenets of software engineering: the lie that programmers did not need to have an understanding of domains, that everything they needed to know was a set of requirements that the code must satisfy.

At the time *DDD* was written the prevailing trend in software development organizations, especially large one, was establishment and enforcement of communication silos. Business Analysts were the only ones allowed to communicate with users or domain experts — Architects were the only ones allowed to communicate with Business Analysts — Systems Analysts and Data Base Administrators were the only ones allowed to communicate with Architects —Designers were the only ones allowed to communicate with System Analysts / Data Base Administrators — and Testers were the only ones allowed to communicate with Designers.

Coders were limited to receiving one-way communications from Designers, in the form of specifications, and Testers, in the form of bug reports.

By 2004 it was becoming clear that an integral and essential aspect of Agile development — stories written by domain experts / users in their own vernacular and On-Site Customer — were not going to be allowed by management. Indirect communication filtered through a “product owner” isolated the development team from domain experts and ‘users’.

Not only did Eric challenge prevailing wisdom regarding the domain, he demonstrated precisely why an understanding of the domain was essential to successful implementation.

A tertiary contribution, was filling in some of the ‘design gaps’ of iterative, incremental, (Agile) approaches to development. For example: XP provided clear directives for building and using domain models but relied on the code (as noted by Eric) as the sole implementation model. Kent Beck and the early practitioners of XP were already expert designers and they had deep implicit knowledge of how to transition from a user story to code. But they never spelled out the nature of that “magic” in any way that would allow less adept developers to follow their example.

Eric provided some intellectual “middleware” that addressed this problem.

It would be difficult to understate the importance of DDD and Eric’s contributions — both with the original book, papers written, and via his active involvement in discussion forums and conferences.

My own appreciation of his work had led me to ask questions; questions I would love to have the opportunity to discuss with Eric and the many experts in DDD that attend conferences like the Domain-Driven Design – Europe conference in Amsterdam. It is my naïve belief, and hope, that thinking about this kind of question might lead in directions fruitful for enhancing DDD.

**Modeling**

Both Eric (in the first few pages of section I) and Martin Fowler (in his foreword) assert the need for both a Domain Model and an Implementation Model and for consistency between them. This is an important observation.

Just how important can be illustrated by three prior failures; one with objects and the second with Agile.

For responsibility-driven object projects the domain model was a set of CRC (Class, Responsibility, Collaborator) Cards. Great effort was expended in creating a set of cards that could be used to verbally walk-through a set of stories about object interactions in pursuit of some goal. Care was taken to assure that each card listed only those responsibilities appropriate for the object and that responsibilities were distributed across the set of objects in an optimal way.

But then what? How did you get from a 3x5 card with a list of domain natural responsibilities [e.g. “provide age upon request” on a Person card] to code [e.g. (SystemClock today – self DOB) asInteger]? In my own work, I invented and utilized an “Object Cube” that extended the CRC card to include, a Knowledge Required, a Protocol, and an Events section, all created from the domain perspective. The addition of these elements to the domain model allowed the programmer to directly write the code to implement the objects.

When Kent Beck (and Ward Cunningham) introduced Extreme Programming, the domain model was the user story, or, more accurately the collection of stories in the Product Backlog. Code was the implementation model. Never explained was how you got from a User Story to Code. And when management disallowed the practice of On-site Customer and, to a great extent, stories actually written by domain experts, the results were predictable.

In typical software engineering, the domain model was an extensive set of “requirements.” Multiple implementation models (e.g. the set advocated by UML) were generated with tenuous, if any, connection to an actual model of the domain. Software was written to “satisfy requirements.” The vast volume of outsourced and offshored software that demonstrably, even provably, ‘satisfied requirements’ while remaining unusable is testimony to the failure of this approach. [Not to pick on outsourcing or offshoring, but projects that remained in-house had many opportunities to cheat by going around impediments to seek fuller, more semantic, understanding of the domain than was ever available to those in remote locations.]

Eric’s book explicitly and implicitly addresses this issue by having the developer / development team create the domain model in cooperation with and in conjunction with domain expert(s).

This approach brings to mind two interesting and inter-related questions.

First, is the domain model truly a ‘domain model’? Or, rephrasing, would the domain experts, absent the influence and direction of the developer, have created the same, or substantially the same model?

I read a lot of business books. Most of them propose models to facilitate the understanding of the business domain. None of these models, excepting PERT charts, have any semblance to any of the standard UML models.

The related question concerns the utility of the domain model, constructed from the perspective of the developer, that Eric asserts is essential for the ongoing success of the development team in their work; is it equally valuable to the domain expert for increasing their understanding of the domain, how it works, and how it might be evolved? Or, as I believe to be the case, that when she is not working with a software team, the domain expert uses other, more familiar, models within the domain?

A corollary question: is it possible for any formal or even semi-formal model to capture the true complexity of any natural domain? (Exempt from this question are internal system domains like device drivers and purely clerical, number crunching, domains like the earliest examples of payroll and order entry, ala LEO in 1951.)

Natural domains, are, most often, complex, highly dynamic, ambiguous, and self-contradictory. An implementation system may be convoluted and complicated but it cannot be complex. Implementation systems are, necessarily, unambiguous, precise, and quantifiable. Is any semi-formal modeling process, grounded in the perspective and world view of the developer, adequate?

None of these questions are intended to fault the kind of models advocated in Eric’s book nor should the be construed in any way to diminish the immense contribution he has made to software development.

Instead they should be seen as asking if we might enhance domain modeling on behalf of software development by exploring the use of story, visualization - like iconic Tibetan Mandalas, metaphor, and the simple system models of General Systems Theory.

**Language**

As noted in the previous section, DDD insists on domain models and implementation models with significant consistency between them. Is not source code an implementation model? Perhaps it is the ultimate implementation model. As Eric noted, in most Agile approaches, the source code is the only implementation model.

Eric also makes an important point concerning language and the assertion that the team should share ***one*** language. By this he means that the domain expert and the developer use the same language. It also argues that the multiple specialty argots of tester versus DBA; Java programmer versus JavaScript programmer; framework expert and tool maven; etc. be unified in some important way in a common language embedded in the domain and implementation models.

Should any ‘common language’ extend past any visual models to include the source code itself? The question is not raised in Eric’s book and I do not know if it has come up in discussions, forums, blogs, etc. since the book was published.

My personal answer is yes!

Early in my career I wrote application programs in COBOL. (System level programs were written in Assembler.) We often had domain experts, bankers in that instance; sit in on ‘code reviews’. We also had domain experts, bank auditors, read and review code before it went into production as a means of detecting or preventing fraud. COBOL made this easy.

Decades later, coding in Smalltalk, it was also possible for domain experts to read and comment on, sometimes enhance, source code. While it is true that domain experts were seldom exposed to large parts of the class library, and never the tiny-C implemented kernel code, they did see the code that implemented objects and the code (usually in Workspace) that exposed object interactions in pursuit of application goals.

I could cite numerous examples similar to one where a domain expert, participating mostly as an observer, responded to a “does not understand” error message, by waving a CRC card and telling the coders, “wait, we forgot to account for this collaboration before we sent that message. Without the collaboration we do not have an object in place to receive the message we are trying to send.

Smalltalk was expressly designed to be a shared language — between human and the computer — that would allow domain experts to express their needs without having to think about how the computer would, internally, satisfy those needs. Simula, before it became Simula I and a programming language, had the same goal; to allow the domain expert to “discuss” the ‘what’ and ‘why’ of a program without ever concerning themselves with the ‘how’.

I am not suggesting any kind of “end user programming” by saying these things as there is a lot of ‘how’ knowledge required to successfully develop software. But, I am suggesting, strongly, that source code be part of the implementation model and that the common language shared between domain expert and developer should include much of the programming language.

While this is clearly possible with languages like COBOL and Smalltalk (and for more scientific domains, FORTRAN) I can categorically assert that it is not possible in languages like C, C++, LISP, or any functional language. It is improbable with Java.

**The Heart of Software**

***“The heart of software is its ability to solve domain-related problems for its user. All other features, vital though they may be, support this basic purpose.”***

Eric Evans, Domain Driven Design, page 4

At the heart of this simple statement is another gauntlet thrown at the feet of traditional, mainstream, software engineering. Another example of Eric challenging prevailing wisdom.

In fifty years of practicing, observing, and teaching software development: I can count on two hands the number of times any approach, method, or theory of software development has asserted the critical importance of software solving domain problems.

Acceptance and Usability testing and User Experience (UX) design might be construed as indirect measures of the heart of software but not more. [That is a discussion for another time and place.]

As much as I agree with Eric’s words and their underlying premise: questions arise.

Should our domain model extend to include the forces and constraints that gave rise to the problem in the first place? If it does not, how can we differentiate between A solution and An Optimal solution? Does it matter?

A corollary: without an understanding of the forces at work in the domain, will we have any way to anticipate both advantageous and disadvantageous reactions of the domain system to the introduction of this new software artifact? Again, does it matter?

As an anthropologist I have extensively studied “technology and cultural change.” A culture is a complex system, so to is a business, so to are all the myriad domains in which and for which we develop software.

The introduction of any change in such a system forces the system out of equilibrium and it changes, in unpredictable ways, in order to establish a new state. Some examples from culture: the invention of clay pots singularly contributed to the movement from hunter-gatherer societies to villages; and, the introduction of the automobile led to massively distributed cities and the sexual revolution.

From the history discussed above, LEO totally transformed the world of business and, eventually, gave rise the massive data stores and associated concerns about privacy and targeted marketing.

Surely, we cannot, and do not want, our domain model to be extensive enough to account for the entirety of the natural system where we will introduce our software. However, my personal belief is that our domain models must incorporate some understanding of how our well-crafted solution affects some of the elements in the domain - specifically and emphatically, the People.

Our software may very well “solve a domain related problem for its user,” but at what cost to the user?

Will our solution totally displace the user? I can remember a time when the development of new software was cost justified by the number of workers that would be fired once it was implemented.

Or will it simply make the user’s work a living hell? Sometime you might ask a gate attendant trying to reschedule a 250 irate passengers of a cancelled flight, just how much the software she is mandated to use “helps” her and makes her job easier and more pleasant.

These questions might be distilled to a general one, “to what degree to ethics, morality, and respect for human beings mandate the extension of our domain model into areas not obviously connected to development of the software itself.”

**Challenge**

Fifteen years ago, Eric pulled together a collection of inter-related software development issues and means for addressing those issues under an umbrella concept stressing the primacy of domain understanding and domain modeling.

This book and his work putting it together should be regarded as a major and important effort. It should also be seen as a challenge.

Can the community that has enthusiastically adopted DDD transcend simple implementation of the ideas and practices advocated in 2004 and work with him, and each other, to enhance and extend; to build on the foundation provided to us.