**Jeffrey Palermo:**

1. I think the article is very good for people needing an introduction to these topics.

2. In DDD, ubiquitous language is a benefit, but a very minor one. The biggest benefit, IMO, is bounded context. This is really describing the core of what each application module should look like. Module is an overloaded term, but an application can have many modules (bounded contexts), and they run in the same memory space until you have to scale under load, but they break apart the application with hard vertical separation so that we lowely programmers can keep the complexity in check. Then when wondering how a change might impact the application, we need not look at the whole of the application, just the bounded context inside which the change is happening.

3. You cite Fred Brooks, but you leave out what may be his most useful lifelong lesson: Top-down architecture is still the way to go! Emergent architecture is getting a lot of buzz, especially in the agile space, which is driven by programmers, not architects. Emergent architecture falls down when used to mean emergent structure. At some level, the invariants of the architecture need to be set, fixed, and applied. Emergent architecture seems to communication that anything can change with little consequence. I, personally, don't like the term.

*[BDS] I agree on all counts, and we’re intending to use Emergent Arch not as Emergent Structure, but to emphasize flexibility within a set structure that enables inevitable change to occur, outside of the architecture invariants. I think Eric Evans inclusion Layered Arch, along with all of the Strategic Design patterns (BC, Evolving Order, System Metaphor, Pluggable Component Arch, etc.) assume some level of invariance to be useful. That said, I had been thinking before your comments that we might need to consider changing the term “Emergent Architecture” to “Agile Architecture.” Of course, that comes with its own baggage… Paul and I shall discuss. J In any case, we do need to underscore that we’re not proposing a bottom-up approach.*

4. Thanks for the mention of Onion Architecture.

*[BDS to PAR] What do you think about #2? I don’t agree that UL is minor, at all… didn’t want to argue b/c it’s a matter of perspective. I do agree that BC is important, but it’s not necc. “required” unless complexity reaches a certain threshold, whereas UL is pretty much essential in my opinion.*

*[BDS]Sent my reply on #3. Good points there, and I had already been thinking that we ought to consider changing the term.*

**Vaughn Vernon:**

Nice work of relating architecture to DDD. I think some essential parts you need are:

1. Hit SOA head on and make sure you contrast Transaction Script and Service Layer (Fowler, P of EAA) to Domain Model and DDD. Many will get lost if you don't spell out that most SOA efforts fly in the face of DDD, because the business logic/rules are captured in Application Layer services and the so-called "domain model" (if it exists at all) gets relegated to a C# (or Java) projections of the data model, resulting in an anemic domain model anti-pattern.

2. Contrast DDD's Aggregate, Entity, and Value Object to Active Record. I know this is a bit more nuts and bolts, but the contrast, if briefly drawn, will stand out to those who think in Active Record.

3. Do emphasize the important of SOA in the Application Layer, but as a thin layer that manages transactions and user security (authorization), but that delegates all business rules to the domain model. Discuss how this layer is used to glue various Bounded Contexts together in the enterprise (see #4).

4. Show how Strategic Design patterns from the blue book can solve architectural integration problems. For example, discuss Bounded Context, Open Host Service, Published Language, and Anticorruption Layer. These are all about how domain models live within an enterprise or extra-prise architecture.

http://tech.groups.yahoo.com/group/domaindrivendesign/message/16817

When you get the article into a more complete state I can give it another read.

**Eoin Woods:**

Thanks for sending me a copy of your article, I enjoyed reading it, agreed with much of what you say, and thought it reflected your obvious enthusiasm and expertise for these subjects.

The major limitation that I found in the article was that it requires the reader to have a lot of faith in the arguments being presented rather than backing up the many assertions and suggestions that you make with terribly convincing arguments (or even evidence). It may well be that many of your readers are DDD and emergent design users already, in which case this won't matter, but for the "average" reader it may well be worth making your points in more detail.

To take a tangible example, the first section of the article implicitly assumes that focusing on the domain model and specifically using a DDD approach will result in simplicity and ease of change. I think most experienced engineers would agree that it's critical to focus on the problem domain rather than the solution domain (and we were teaching that a long time before Eric Evans starting working in this area) but I didn't really come out of the section feeling that the necessity for DDD specifically had been made all that strongly. There seemed to me to be quite a number of other ways of getting those benefits.

The other high level point that I noticed was that the two themes of the article (DDD and emergent architecture) didn't appear to be tied together all that strongly. You can obviously practice one without the other so I was expecting the relationship between them to be drawn out earlier and more explicitly. Section 5 reads nicely but you don't really get an idea of your key point (that DDD and emergent design work together) until the last couple of paragraphs and even then, it's not quite clear how it's meant to work.

In summary I wonder if you could restructure the article around the relationship between the two more than the two separate dimensions. This might be a more interesting set of ideas that would you to delve into them in more detail?

A few specific comments on things in the paper were:

1. In section 2, it's true that messing up the domain model can doom a project, but there are a lot of other ways to do it too! Even with a decent domain model there are a lot of other things you need to get right.

2. In section 3, when talking about the example conversations, you say that the conversations are the same apart from "the use of language to describe the domain". I think I completely disagree! In the first conversation, the expert is trying to discuss the problem domain, whereas the architect is determined to discuss the solution domain. They're having different conversations. You don't need to be a DDD advocate to spot that this is going to be a problem! (As you say the problem is too little time understanding the domain ... a problem we've been aware of since the day of De Marco and Yourdon structured analysis :-)

3. Towards the end of section 3 you say ubiquitous language is not about embracing either technical or business domain terminology. Surely it \*is\* about embracing business domain terminology, but improving it by using it consistently and defining it rigourously?

4. The problem with "code as the domain model" is of course validating it with domain experts. (Unless you use Naked Objects ... before Dan points this out! :-)

5. Types of complexity - isn't the second one also accidental complexity but it's accidental problem domain complexity, not accidental solution domain complexity? (After all isn't "non-essential" the same as "accidental"? :-)

6. In part 5, you say that lots of architects specify as much of the architecture as possible up front. Actually I think the problem is that many architects don't specify the \*architecture\* very clearly but instead try to define lots of internal component details. This is what annoys teams; they're missing what they really need (components, responsibilities, interfaces and interactions) and have been given what they were expecting to do themselves (component design).

7. In part 5, I wasn't sure about the phrase "Over-specification creates inflexible boundaries and results in brittle software" ... it depends what you mean by "over-specification". I'd say \*under\* specification causes those problems (i.e. not defining the components, responsibilities, interfaces and interactions clearly).

8. Totally agree about the architecture working embedded in the team. I do exactly this with my architecture team.

9. In part 5, paragraph 5, you talk about being "focused on creating a loose boundary that is tightly coupled to the domain model" .... I'm not sure what you mean by this. This is an example of what sounds like an interesting idea that would be great to focus on so you can explain it in detail.

10. In part 5, perhaps explain "onion architecture", I'd not heard of this (but I'm reading Jeff Palermo's site with interest now!)

**Dan Haywood:**

On "The Architect as Language Steward": I didn't pick this up from Evans' book. I appreciate that this article is targeting architects, but I think making this a responsibility of the architect is wrong. It's meant to be the entire team's responsibility. My preference is to use agile practices - eg scrum master along coupled with "way of working" agreements and agile retrospectives. Also regular demos to the customer/stakeholder will help ensure that the UL is nurtured. Our experience is that using the Naked Objects Framework for prototyping and/or deployment is also a powerful way to build up a UL across the team.

On "What's a Model?" : Perhaps you should talk about "Model Driven Design", since is the other central piece to Evans DDD book.

"Beyond being a simplified representation of a thing, a model..." - I think this is the wrong parallel to draw upon. Evans' use of "model" is the actual thing, not a representation of the thing. Modelling is a design activity in that one is decides what is relevant to include and what is not; but what you end up with IS the code.

"It's important to note here that a model is conceptual rather than an artifact" - well, it's not a single artifact diagram, sure. But the set of artifacts do express the model. The model isn't anything other than what's in that set of artifacts. I think this is what you are saying, but that word "conceptual" is possibly overloaded and likely to be misunderstood

"This means that diagrams, documents, automated unit tests and storytests" - on "storytests" - you could also mention that Cunningham and Mugridge talk about the UL in their FIT book.

"List of artifacts that can be used to express a model" - better: "taken together"

"3) Accidental Complexity - non-essential complexity introduced by designs, frameworks and code that bleed into the domain model " - should a domain model be annotated with semantics from adjacent layers? There are pros and cons. I'm in favour of it.

"Eric Evans proposes strategies for dealing with each of these types of complexity in Part IV of Domain-Driven Design" - does he? I taxonomy of complexities is all well and good, but I think that Evans book only really is concerned with (1). I can't recall anything he says that relates to (2) or (3) per se.

". The architect should hold the role of strategic designer on a team,..." - yes, there is a strategic role for the architect to fill, but it's more to do with defining how different bounded contexts relate. I think a conversation on BCs is probably what you should be getting into here instead.

"V. Emergent Architecture" - : if you're going here, doesn't this bring in TDD, YAGNI and other agile practices?

"Design for change really means understanding which coarse-grained aspects of a system are likely to change and then isolating those which are likely to change from those that aren't. " - : This wouldn't be an agile viewpoint. YAGNI.

"Layered Architectures are typically leveraged to achieve the kind of ordering and isolation described here, but they tend to enable, if not encourage, the kind of accidental coupling ..." - in what way?

On Onion Architecture: : I think the hexagonal architecture (Alistair Cockburn) predates this, and is essentially the same thing. I'm a fan of it though, nice to see you going in this direction. I describe Naked Objects in these terms too, see http://starobjects.sourceforge.net.

"...and Data Access should be implemented elsewhere..." - Mainstream DDD is easier with an ORM. No ORM makes it hard to keep data access concerns out of the domain.

". Your Domain Model, which you should expect to change at a faster rate than any other part of the system, can evolve even as infrastructure concerns like Data Access are implemented and tested." - : It would be nice if that were true, but in practice the UI tends to be more fickle. Also, the domain model almost certainly cannot change without a change to the RDBMS schema. So, really it changes no faster than the slowest moving part of the system (the data).