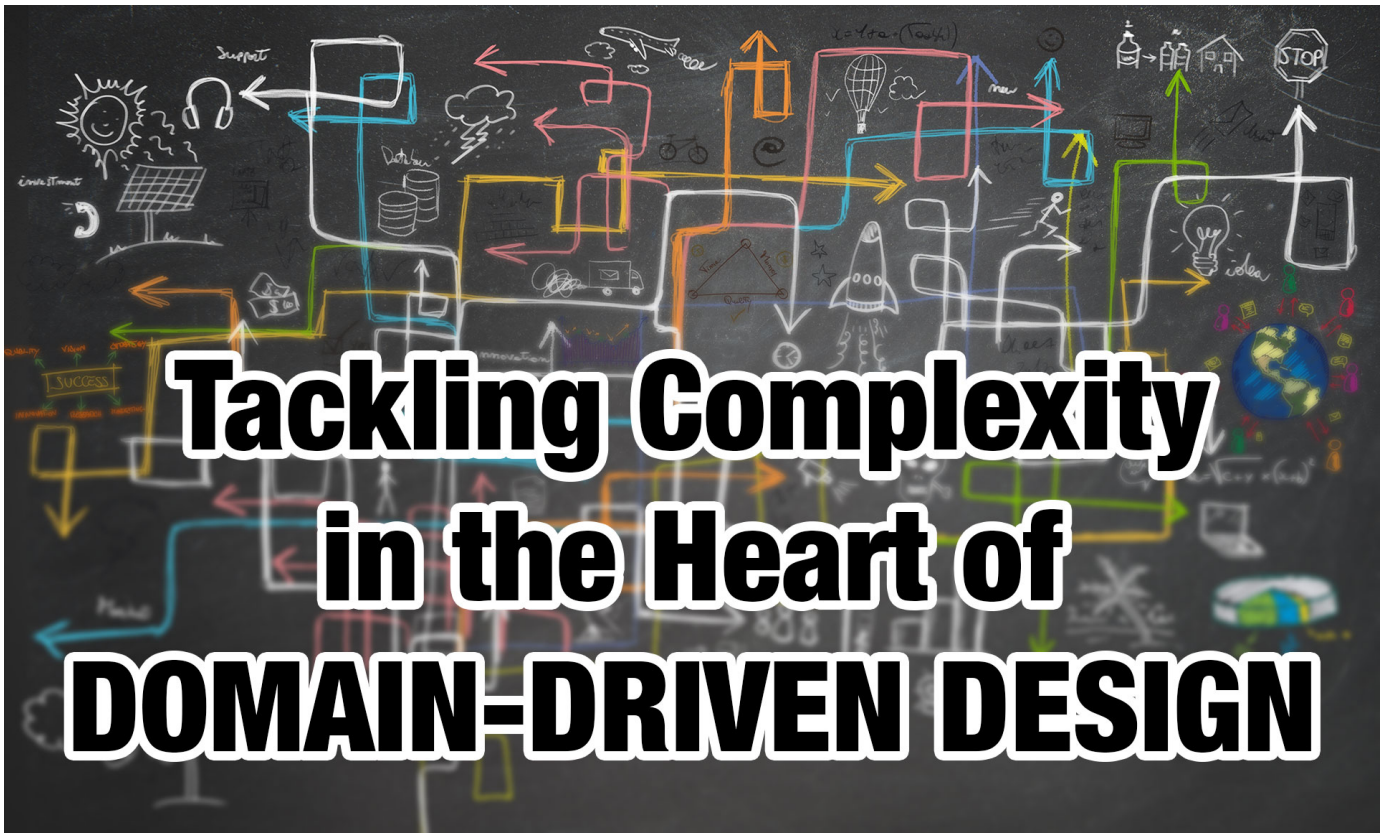
Tackling Complexity in the Heart of DDD

Posted on April 5, 2016 | 7 min (1391 mots)



Let's do a little experiment: try to explain the gist of Domain-Driven Design to someone who has no clue about it. This, especially doing it succinctly, is not easy. Heck, I struggle with it myself. Bounded contexts, entities, domain events, value objects, domains, aggregates, repositories… where do you even start?

To find the order in the apparent chaos, I want to analyze the DDD methodology from a rather unusual perspective — by applying Domain-Driven Design to Domain-Driven Design itself. After all, this methodology is intended to deal with complex domains, isn't it?

Let's start by identifying the core domain: what is DDD's main competitive advantage, and what are its means of achieving it?

The Core Domain: Ubiquitous Language

In “Domain-Driven Design: Tackling Complexity in the Heart of Software”(the Blue Book), Eric Evans argues that poor collaboration between domain experts and software development teams causes many development endeavors to fail. DDD aims to increase the success rates by bridging this collaboration and communication gap.

To allow fluent sharing of knowledge, DDD calls for cultivation of a shared, business-oriented language: Ubiquitous Language. This language should resemble the business domain and its terms, entities, and processes.

The Ubiquitous Language should be extensively used throughout the project. All communication should be done in the Ubiquitous Language. All documentation should be formulated in it. Even the code should “speak” the Ubiquitous Language.

Many methodologies strive to reduce risk and increase success rates of software projects, but since Ubiquitous Language is DDD’s means of achieving it, I consider it as the Core Domain of Domain-Driven Design.

Defining a Ubiquitous Language is not a trivial thing to do. Since software doesn’t cope well with ambiguity, each Ubiquitous Language term should have exactly one meaning. Unfortunately, that’s not how human languages work — often words have different meanings in different contexts. To overcome this hurdle and support the process of cultivating a rigorous language, another DDD pattern is employed: Bounded Context.

Supporting Sub-Domain: Bounded Contexts

To prevent terms from having multiple meanings, DDD requires each language to have a strict applicability context, called Bounded Context. This pattern defines a boundary, inside of which the Ubiquitous Language can be used freely. Outside of it, the language’s terms may have different meanings.

Although the Bounded Context pattern is an essential part of Domain-Driven Design, I consider it a Supporting Sub-Domain, since its purpose is to support the formation of a Ubiquitous Language, the Core Domain.

As I mentioned earlier, the code should also "speak" the Ubiquitous Language of the Bounded Context in which it is implemented. But how do you implement a business domain in code? There is no one-size-fits-all pattern for implementing a business domain. Multiple options are available, and that's our next stop. Be warned: sacred cows are about to be hurt...

Generic Sub-Domain: Domain Implementation

These patterns provide different ways of implementing the business domain's logic:

1. Transaction Script
2. Active Record
3. Domain Model
4. Event-Sourced Domain Model

Each of these patterns suits a different level of domain complexity. The pattern you choose should be expressive enough to reify the Ubiquitous Language in code. It is crucial to point out that this decision is not set in stone. As the business evolves and the Ubiquitous Language's complexity grows, the implementation pattern can be upgraded to a more elaborate one.

The aforementioned four patterns of business-domain implementation are the ones I am currently familiar with.

Indeed, there are may be others than I am currently unaware of.

New ones may be invented in the future.

Their implementation differs greatly in various programming paradigms.

Some best fit a certain programming paradigm but are complex to implement in others.

With all this volatility in mind, are they an essential part of Domain-Driven Design?

Since the Domain-Driven Design methodology cannot encompass all business domain implementation patterns, this know-how can, and should, be borrowed from other sources. For example, the Transaction Script, Active Record, and even Domain Model are described in Martin Fowler's "Patterns of Enterprise Application Architecture" book. By definition, the ability to rely on "off-the-shelf" solutions makes them a Generic Sub-Domain. Yes, even the Domain Model pattern.

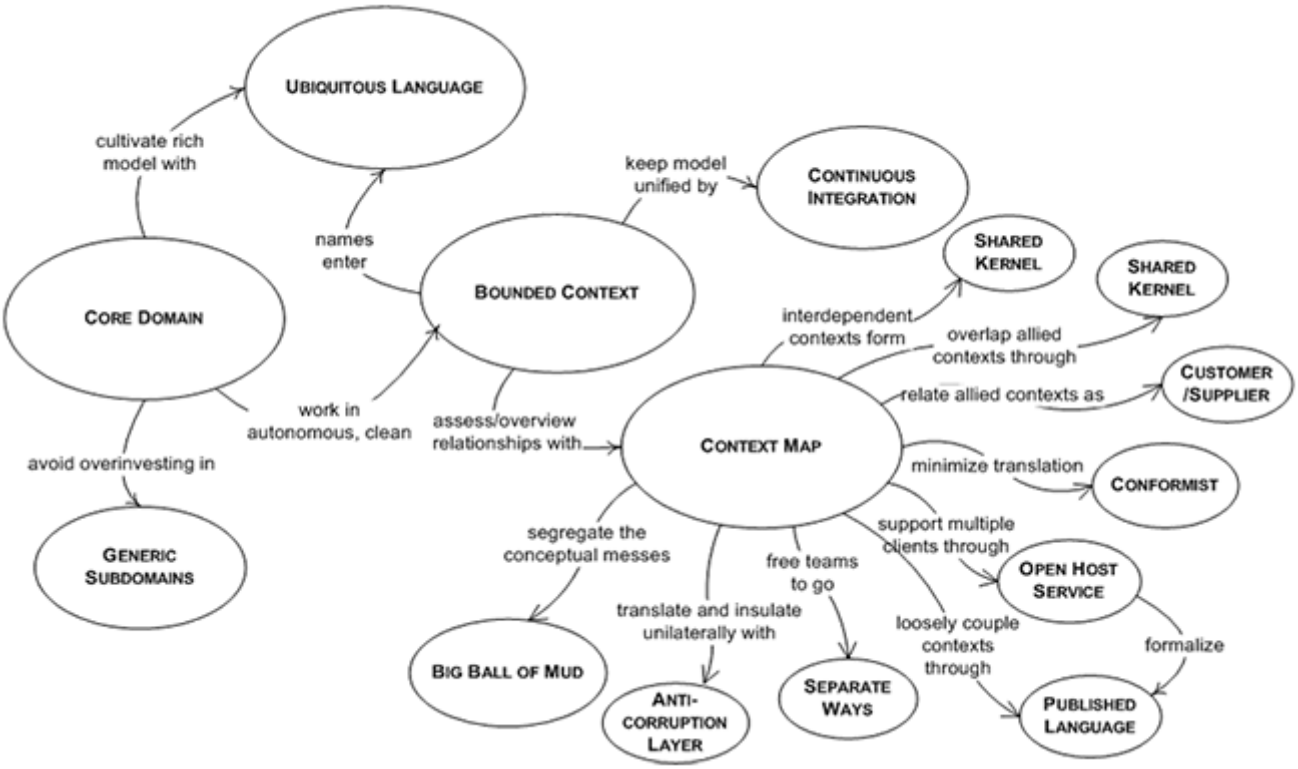
Implications

The decoupling of Domain-Driven Design from the tactical modeling patterns can have positive, far-reaching implications on DDD's accessibility and adoption rates. I want to elaborate on three of them: reducing DDD's complexity, widening its applicability, and the ability to gain a lot of traction by jumping on the Microservices bandwagon.

1. Reduced Complexity

This mind map by Eric Evans depicts the patterns that constitute the Domain-Driven Design methodology:

And this is how it will look if we drop the tactical modeling patterns:



Shabang! Which one do you think will be easier to grasp and explain?

Decoupling of DDD from the Tactical Modeling patterns will prevent many of the misconceptions and difficulties many newcomers experience – for example, reading the first four chapters of the Blue Book and having a feeling that they've got a strong grasp of DDD. And speaking of the Blue Book, many complain that it doesn't provide enough code samples. Well, guess what? Once DDD is decoupled from the Tactical Modeling patterns, it no longer requires any code samples at all. It's a pure system modeling methodology that can be applied in any technology stack and any software paradigm.

2. Wider Applicability

I strongly disagree with the notion that Domain-Driven Design should be applied to complex projects only. This notion is driven by the strong coupling of DDD to the Domain Model pattern. Once we break this coupling, a whole new world of possibilities opens up.

Communication

No matter how simple the business domain is, if team members use different terminology for the same artifacts, sooner or later they will find themselves in the realm of accidental complexity. The Ubiquitous Language pattern prevents this scenario and yields a clear communication medium between all team members.

Business Growth

A domain's complexity increases more often than it decreases. This possibility of increase is highest for the so-called not complex projects. Once this happens, the implementation pattern decision should be rethought and adapted to the new complexity levels.

3. Microservices

Microservices are red hot nowadays. Widening the applicability of DDD to more project types will allow many microservices-based solutions to harness the invaluable DDD tools. The Bounded Context pattern provides a business-driven way of dividing a system into a set of independent services, and the Structure Map is a great way to map the services' topology and interaction patterns between them.

"Are You Nuts?"

That's what you're probably thinking right now. However, I don't think that my proposition – to take the Tactical Patterns out of DDD – is as crazy as it initially sounds. Back at the DDD Europe 2016 conference, Eric Evans himself stated that the Domain Model implementation described in the Blue Book was intended to be *a* way of implementing a *Domain Model*, but many mistook it as *the* way of implementing *Domain-Driven Design*. See, the Tactical Modeling patterns were never intended to be the one-and-only way to do DDD, but many consider them as such. They produce extraneous noise and detract attention away from the most important, and unique-to-DDD material.

Also, you cannot say that the Domain-Driven Design methodology is in its perfect state, and has no reason to change. Unfortunately, its low adoption rates speak for themselves. DDD deserves way more attention than it gets. The Blue Book came out more than a decade ago, and since then the methodology has barely

changed. I believe that it should change. Not because it's bad – on the contrary, because it's great. But it has much, much more potential than it has currently realized.

Final Thoughts

In no way did I intend to denigrate the importance of Tactical Modeling. Quite the opposite: this subject deserves much more attention than it gets. But in its own context. There are many more patterns besides the Domain Model, and more ways to implement them, than can fit in a single DDD book. Moreover, these patterns can be implemented even on non-DDD projects, and a project can follow the DDD principles even if it doesn't have a single aggregate in it.

What do you think?

I'd love to hear your opinion on this in the comments.

All Posts in the "Tackling Complexity" Series

- Tackling Complexity in Microservices (/2018/02/28/microservices/)
- Tackling Complexity in CQRS (/2017/03/20/tackling-complexity-in-cqrs/)
- Tackling Complexity in Domain-Driven Design (/2016/04/05/tackling-complexity-ddd/)

← PREVIOUS POST (HTTPS://VLADIKK.COM/2016/03/05/INTERVIEWING/)

NEXT POST → (HTTPS://VLADIKK.COM/2016/06/29/TEST-SCOPES/)

Featured Comment

Vlad Lyga • 3 years ago

Vladik, just two days ago I had a talk with another DDD practitioner , also an architect in his company, and this is pretty much what we talked about. The difference is you've put it so nicely in this awesome blogpost . Great work! This kind of different thinking is what eventually drives evolution of ideas in our business. Thank you

▲ ▼ • Share ›

An0nym0usC0ward • 3 months ago

https://xkcd.com/927/

The tactical patterns are indeed just one possible ubiquitous language for structuring software. With several distinct ubiquitous languages covering the same domain/bounded context, we'd have babel.

Several years ago, the GoF introduced some design patterns. These became pretty standard naming conventions. That's one of their key values. The same as with the tactical patterns in DDD. You absolutely can do without them, or come up with an alternative. Only, if you do, their value as a tool for efficient and precise communication vanishes.

▲ ▼ • Reply • Share ›

vladikk Mod ➚ An0nym0usC0ward • 3 months ago

The tactical patterns are a way of modeling a business domain in code. There are other ways - event sourcing, dci, and other patterns, each having its pros cons. If I am not mistaken, Eric proposes other ways of modeling domains in the last chapter of the DDD book. I am not for inventing another one, but for using the right tool for the job. Unfortunately, many associate DDD with tactical patterns only, use it as a silver bullet, and expect it to work everywhere.

▲ ▼ • Reply • Share ›

Alexey Zimarev • 2 years ago

I think this is the whole theme for DDD Reboot. Delay tactical patterns until the very last moment and concentrate on ubiquitous language. The essence of DDD is - "develop Ubiquitous Language within Bounded Context". These are two the most important concepts. Searching for core domain and making a context map, basically those parts of strategic design that are described from Chapter 11 of the Blue Book. The Scott Millet and Nick Tune book takes it in the right order and I think they manage to explain it quite well.

▲ ▼ • Reply • Share ›

Emilien Pecoul • 3 years ago

Really cool post, can't agree more, and actually wrote something with lots of shared ideas http://ouarzy.azurewebsites...

I also experienced how easier DDD can be explained when you remove the tactical part.

When do we start the #NoTacticalPattern move guys?

• Reply • Share ›

vladikk Mod ➚ Emilien Pecoul • 3 years ago

Thanks a lot, Emilien!
Wow, indeed, our posts have lots of shared ideas! I guess it means that the #NoTacticalPatterns movement has already started.

• Reply • Share ›

ToJans • 3 years ago

Hey Vladik,
Good article; for me the biggest value of the "blue book" is that it allows me to talk about scope, relationships, dependencies and components etc using a common language shared by everyone involved in the process... I assume that's the same thing you would like to point out in this post?

• Reply • Share ›

vladikk Mod ➚ ToJans • 3 years ago

Thanks, Tom! Yes, I think the Strategic Design is the essential part of the book, and has wider applicability than the Tactical Modeling patterns.

• Reply • Share ›

Vlad Lyga • 3 years ago

Featured by Vladikk

Vladik, just two days ago I had a talk with another DDD practitioner , also an architect in his company, and this is pretty much what we talked about. The difference is you've put it so nicely in this awesome blogpost . Great work! This kind of different thinking is what eventually drives evolution of ideas in our business. Thank you

• Reply • Share ›

Alex Badyan • 3 years ago

Brilliant. I think this does a lot to simplify DDD and promotion of it. Not sure it will cause more people to adopt it just like that because we do like practical solutions and what you did turns DDD into a somewhat amorphic thing. I do think this makes talking about DDD much easier.

• Reply • Share ›

Vlad Lyga ➚ Alex Badyan • 3 years ago

I personally think that this actually makes DDD more practical , in a sense that DDD is instantly becomes useful in a much wide range of applications/scenarios . The 'DDD' thinking. I would also very strongly advise to look into Data Context and Interactions by J. Coplien, his book "Lean Architecture" is a great read.

1 • Reply • Share ›

ALSO ON **VLADIKK**

SQS Exactly-Once Processing is a Hoax - Vladikk

5 comments • 2 years ago

vladikk — Thanks for the clarification! In my opinion, that defies the purpose of having a distributed queue, but that's a whole other

Tackling Complexity in CQRS

35 comments • 2 years ago

jbogard — Complexity trap #4: projections should be asynchronous.I can count on 1 finger the number of times, within a single

TDD: What Went Wrong …Or Did It? - Vladikk

6 comments • 3 years ago

DDDEU 2016 Impressions

7 comments • 3 years ago

Weronika — I'll check out both! Thanks for

(mailto:vladik@khononov.com) (https://www.facebook.com/vladikk)
(https://github.com/vladikk) (https://twitter.com/vladikk)
(https://linkedin.com/in/vladikkhononov)
(https://stackoverflow.com/users/1736215/vladikk)
(https://www.instagram.com/vladikkk) (https://telegram.me/vladik_kh) ()