

Domain Driven Design and Model Driven Software Development

*Karsten Klein, hybrid labs, January 2007
Version: 1.0*

Introduction

Eric Evans Book 'Domain Driven Design' was first published end of 2003 [Evans2003]. It has been very well received by the object oriented software development community and many of the described concepts have been successfully applied.

During the J2EE era the attention on the domain model has been shifted in favor of technology significantly. The central objective that Evans advocates is to realign focus onto the domain model again, rather than on the utilized technologies the system is build upon.

In [Evans2003] the topic of Model Driven Software Development (MDS) is only touched in a limited, but perhaps the only adequate fashion taking the point of time into consideration. In the past years the concepts provided in the Domain Driven Design (DDD) realm have proven to be very helpful in constructing healthy and sustainable software. In the meantime MDS experienced a renaissance. After code-generation approaches were largely abandoned the Model Driven Architecture (MDA) and MDS initiatives have brought back the idea of generating artifacts from models utilizing Domain Specific Languages and are constantly gaining momentum in industrial environments.

This article is supposed to provide a common ground for elaborating the opportunities of combining and Model Driven Software Development (MDS). Please note that OMG's Model Driven Architecture® (MDA®) [OMG-MDA] is largely excluded from these considerations, since the MDA paradigms are again quite technical and focus on the platform independence, which is explicitly not regarded as a domain concern.

Domain Model and Ubiquitous Language

The domain model is essential. It reflects the understanding of a particular domain using classes, attributes, relationships and collaboration behavior. It can be extended with additional information and constraints. The domain model is the central communication tool, with which is constantly nourished with further insights and requirements. It also becomes the pivot for communication and influences the language we speak about the domain. Evans [Evans2003] introduced the term 'Ubiquitous Language' in this conjunction. The ubiquitous language evolves with the domain model and enables to describe the characteristics of the domain with increased precision. The ubiquitous language is the common ground of the software engineers and the domain experts. Error prone translations are minimized.

Interestingly DDD and MDS have a large intersection here. The focal point of DDD is the domain model and the ubiquitous language supporting it, while MDS projects start with a Domain Specific Language (DSL) and a model facilitating the DSL.

Directly mapping domain model (DDD) to model (MDS) and the ubiquitous language to a DSL is of course a very abstract and adventurous mapping initially, but it could at least be a possible interpretation of domain model and DSL from the MDS point of view.

The main problem here is that both parties - DDD and MDS aficionados - try to be as general as possible. The DDD person does not determine how the domain model is expressed.

Normally the DDD domain model is determined by several artifacts including drawings, documentation, annotations and – most favorable and complete - code. In fact Eric Evans in [SE-8-2006] puts it much more abstract: “A model [...] is a set of related concepts that can be applied to solve some problem [...]” and “The model itself is a set of abstractions, a set of rules of how these abstractions interact [...]”. The manifestation of the model is not determined at all. It can be represented by anything appropriate. In conclusion the same can hold true for the ubiquitous language. Reading [Evans2003] the ubiquitous language appears to be the spoken language, which simply but consequently uses the abstractions and terms defined by the domain model. The manifestation of the ubiquitous language is concluded to be undefined as well.

On the other hand the MDSD community does not determine how model and DSL should look like. The manifestation of both therefore is depending on the utilized MDSD infrastructure and the used concepts. Normally the DSL is represented by an appropriate meta model. The model – granting the target domain as constant - can be simply seen as function of the DSL.

$$\text{model}_{\text{domain A}} = f(\text{DSL}_{\text{domain A}})$$

So what speaks against directly mapping the DDD domain model to the MDSD model and the DDD ubiquitous language to a DSL? The ubiquitous language used for communication would be more concretely defined in the DSL. The DSL is just a (partial) manifestation of the ubiquitous language. The model is the manifestation of the domain using the DSL.

Of course this mapping will never be a hundred percent. In [SE-8-2006] Eric Evans expresses that he sees this approach as an attempt towards domain driven design, which is still cluttered with too many technical artifacts, hiding the important information inherent in the domain model.

However, pursuing this thinking can open new perspectives. In the subsequent chapters a possible mapping of DDD to MDSD is provided. Please regard this as an experimental approach, which tries to raise some questions, encourage discussion and eventually may reveal some opportunities.

Evolving a DSL specific to Domain Driven Design

Given the idea of a DDD driven MDSD approach the first obvious thing to do is to construct an appropriate meta model. This will of course not be a one time task, but a continuous effort in order to reflect the evolving knowledge about the domain and the ubiquitous language.

Approaching this in a pragmatic fashion a DSL is considered, which is specific to the problem of doing domain driven design, not yet taking the specificity of the problem domain into account. It is a DSL for the domain ‘domain driven design’ – a DDDSL. The specificity to the problem domain is completely represented by the model.

Put into other words, the DDDSL defines the vocabulary, the model defines the language. For example the terms ‘entity’ and ‘value object’ are part of the vocabulary and the ‘RouteSpecification’ is a term of the model. Both are part of the ubiquitous language. One could also say that the ubiquitous language is an instance of the Domain Driven Design meta model and therefore located on a lower abstraction level as a DSL (which may be a first insight).

Evolving the Domain Model

Taking the previous section as granted the model can be expressed using the DDDSL. While developing and extending the model the ubiquitous language will evolve, too. Moreover the DSL will be extended as required for being able to express the domain model.

Assuming we have a very final idea of a DSL, how is the domain model now expressed in order to represent a valid MDSO model ready for generating code?

Before continuing it is very important to have sacrificed the ‘holy grail of domain driven design’ [Evans2003]. MDSO is explicitly not a discipline of generating everything. You will never try to express object behavior with a MDSO model. You will never try to make your DSL as expressive as a programming language.

Technical Domain Model

The modeling of the technical domain model can be easily done with a UML tool. UML is widely known and since the technical domain model should facilitate a medium for communication with non-software developers it appears to be the best choice.

The UML is extended by a so-called profile containing stereotypes and tags to be used during the modeling. The profile provides the vocabulary to be used in the technical domain model.

It seems that many people are alarmed when hearing that they have to do the design using UML. In fact you don’t have to. It is just an opportunity. Currently activities in the MDSO community go into the same direction. Often UML tools are not very handy, when the DSL is kind of simple. The MDSO community has made a significant effort into being able to easily build custom editors for a DSL [OAW2006].

In the scope of this article a modeling with UML is chosen.

Constraints

Constraints significantly supplement a standard UML model. However apart from low-level constraints it is hard to find standardized mechanisms to express general constraints in your UML model and have them being expressed correctly in code for checking object instances at runtime. There are activities to standardize constraints using the Object Constraint Language (OCL) specified by the OMG [OMG-OCL]. Based on OCL a variety of tools supporting your MDSO approach can be found [Kent-OCL] [EMF-OCL] [Voelter2006].

Some OCL tools support OCL checks only on the model level that means on an abstract level higher than one would naturally expect the constraints when talking about the domain model. When looking at any OCL tooling in the MDSO area, make sure that the level the OCL expressions are written and when they are applied is clear. E.g. the constraints used in [Voelter2006] are expressions on the level of the meta model and are used to check consistency of the model. They have nothing to do with the domain model as such.

Code

The behavior of objects can be best expressed with code. We basically do not put any code in the model, but we nourish the generated code with additional functionality. The initial result from a generator run on the constructed model, can therefore only result in a preliminary anemic object model. The specific behavior has to be added manually by the software developers. There are techniques to accomplish this task and not mixing any generated and

hand-crafted code (see section “*Recipes – Integrating generated and non-generated code*” in [Voelter2006]). Such a separation should be ensured to be able to rerun the generator without eliminating the hand-written lines.

Documentation

Documentation is a continuous effort. You document the model using free-text notes as you design it, you document the code as you write it. Documentation – in the sense of Eric Evans always uses the ubiquitous language.

Further documentation describes the APIs including the services that are exposing your domain model towards a client.

Manifestation of the Domain Model with MDSD

In summary the domain model is an aggregation of everything expressing it. That means it is the holistic composition of the technical domain model (including the constraints), code and documentation. Compared to a classic domain driven design approach nothing has changed in this respect. Only – and this is the added value – the model and the language to describe the model are made much more explicit! They evolve as part of the whole and are always up to date.

Summary

“So far I haven’t seen much in the MDA round that has been useful to me in this way.”

Supposing that Eric Evans included the MDSD approach in this statement (which he probably does taking the context of the interview [SE-8-2006]) it is of central relevance for this article.

Listening to the interview one can get the idea that the interviewer is a MDSD-affectionate (not to fond about MDA) and the interviewee, Eric Evans, is very keen on not allowing MDSD/MDA to enter the realm of Domain Driven Design. I personally found the interview a little disappointing. It is professional and respectful, but does not add anything new. The MDSD advocate can not manage to push the DDD evangelist into a direction to jump on the potential opportunities, while the DDD evangelist obviously simply doesn’t want to be pushed that way. He finds DSLs exciting, but does not differentiate MDA, MDSD and Case Tools to a large extend and is revolting about the technological complexity. They do not meet on the topic that would have interested me the most.

In this article I have largely laid out my idea of combining DDD with an MDSD approach. And this is actually the way I’m currently pursuing my project work. There are still a lot of unknowns and shortcomings, but I highly appreciate and advocate the fundamental idea of Domain Driven Design to render the domain model as central element of software development.

The following sections summarize the findings and opportunities in this respect.

Opportunities

Domain Model and Ubiquitous Language are more Explicit

Since the knowledge about a domain is not only in documentation, but is the fundamental part of the DSL and therefore the technical domain model it has become more general and explicit. Designers and developers have to use the language to design the model. It is not personally flavored and cannot be miss-interpreted, since it directly results in generated code.

Domain Models are Normalized

The resulting technical domain models focusing on different domain, but using the same Domain Driven Design DSL are normalized. Domain specificity is inherent to the model not the DSL. The DSL can be reused, which also offers standardization opportunities for the DSL and the underlying generator framework (e.g. standard templates addressing different aspects of the software architecture).

Focus on the Domain Model

The designers and developers can finally focus on the domain model. They design the technical domain model, they add hand-written code supplementing the resulting generated code. All the rest – the layered architecture, the technologies used, etc. is taken care of by the generator provided. The argument that designers and developers are not defining the language they require to talk about the domain ([SE-8-2006]) does not hold true. They base the language on the DSL (which they can influence) and construct their language aligned to the technical model they design.

Technical Domain Model always Up-to-Date

Since the technical domain model in the MDSD arena is part of the source of a module, it is always up-to-date by definition. If a change needs to be done on the model it does not first happen in the code, but it does first happen in the technical domain model. Therefore, the technical domain model – the key artifact for communication purposes – is never outdated.

All Advantages of MDSD apply

In general all the advantages of MDSD apply. E.g. you do not find the developers doing error prone repetitive tasks all day long. Once having identified such a need the generator can be easily extended to address this in a general reusable way as part of the MDSD infrastructure. Moreover the approach provides a high degree of flexibility concerning the software architecture. Ultimately the architecture can evolve, while designers and developers have to make no adaptations.

Conclusion

The article tried to outline that DDD and MDSD can be combined in unison. It provides a very natural approach to achieve the goals of both worlds and to build on the respective advantages. By listing the opportunities in the summary section it tries to open the readers' mind of pursuing the one or the other option in this context.

The article is meant to continuously evolve in the future. Please do not hesitate to provide feedback to contact@hybridlabs.org.

“Perhaps someone will take some of the things that the MDA people have worked out and maybe they create a new modeling language [...]. I think experiments are great and I hope that people continue trying all sorts of things.” Eric Evans [SE-8-2006]

References

- [Evans2003] *“Domain-Driven Design: Tackling Complexity in Software”*, Addison-Wesley 2003
- [Avram2004] *“Domain Driven Design Quickie”*, <http://infoq.com/books/domain-drivendesign-quickly>
- [SE-8-2006] *“Interview with Eric Evans”*, http://www.se-radio.net/index.php?post_id=67317
- [OMG-OCL] *“OCL Specification”*, <http://www.omg.org/docs/ptc/03-10-14.pdf>

- [OMG-MDA] “MDA”, <http://www.omg.org/mda/>
- [Kent-OCL] <http://www.cs.kent.ac.uk/projects/kmf/links.html>
- [EMF-OCL] <http://www.zurich.ibm.com/~wah/doc/emf-ocl/index.html>
- [Voelter2006] <http://www.eclipse.org/articles/Article-FromFrontendToCode-MDSDInPractice/article.html>
- [Efftinge2006] “*The Pragmatic Generator Programmer*“, <http://www.theserverside.com/tt/articles/article.tss?l=PragmaticGen>