12/24/2018

Survey on Domain-Driven Design

Boğaziçi University SWE 577

Fall / 2018

Altug, Yusuf Cagri

Contents

Introduction and Motivation

Software development is in our lives for almost 50 years. There is a huge information how to do that. Lots of solutions exist for specific problems, even huge variety of coding languages. It is almost impossible to find a solution for any kind of problems with the today's knowledge of software design. With the increase of the usage of computer technologies, the need of software solutions appeared on the market.

Leading company or not, they all use software solutions for their business. Sometimes these solutions depend their implementations on easy business logic, sometimes not. Weather complex or not the business is, a continuous need of adding new features or changing the present business always exists. When this type of needs occurs, there should be a change on the related software solution. The software seems to be very simple at first becomes very complex and sophisticated at its final stage. At that point, for the developers changing an implementation or adding some new features become complex as the software itself.

Software should be adaptive to changes. It also should be easy to add new things without breaking anything. On the contrary, when complexity of the software increase, software designers can no longer understand the current software. This causes the next implementations to be harder and makes software not adaptive. So, a new approach to make enterprise applications which have sophisticated logic or not is needed. In order to fix such problems described above the concept of domain-driven design (DDD) has come to scene. It is first described by Eric Evans in his book named *Domain-Driven Design: Tackling Complexity in the Heart of Software*.

Description of Domain-Driven Design

Domain-driven design is defined as an approach or framework rather than design pattern or application which helps to solve a particular problem. It is a philosophy which software development process has to have. It makes developers get the proper patterns for building enterprise solution from the business domain.

Domain, Sub Domain and Model

Domain is *a specified sphere of activity or knowledge* (Definition of domain, 2018). With the definition of the domain word, we can simply say that domain in software world refers to a particular area of the business on which is needed to be implemented to software solution. In other words, domain is a specified sphere activity or knowledge which the software solution is built on. For example domain of a banking software includes money and stocks of companies or single users. For different areas, there exist different domains.

We can consider domain as a problem space also. This problem can be related to all company. It can be all the software of the company. When this is the environment which the company has, we can divide this huge domain into smaller ones. These small domains are called subdomains. Suppose we have a transportation company. This company has air transportation, sea transportation and land transportation. Transportation process is the business domain of the company, however there are subdomains regarding sea, air and land parts of the all transportation.

There are some principles that are focused on particularly. First the core domain and domain logic is the center of the Domain Driven Design. Second complex model designs are needed to base a powerful domain. Lastly, collaboration with domain experts has a key role to develop strong application model. It also helps to solve problems related to domain itself very quickly. As seen above, models are placed at the center of the Domain Driven Design. Basically a model is defined as a system which consists of abstraction layers describing ideas and principals of an existing domain. In order to solve the problems of the specified domain we get help from models of the domain. According to Eric Evans, models has three utilities;

"The model is the backbone of a language used by all team members (Evans, 2003)*".* There is a strong relationship between model and implementation. This strong bond causes the product has the analysis properly. It also helps to maintain the product.

"The model is the backbone of a language used by all team members (Evans, 2003)*"*. The language helps domain experts and software developers to communicate easily.

"The model is distilled knowledge (Evans, 2003)*".* This creates some experience on the early stages of the development.

Bounded Context and Ubiquitous Language

There can be multiple models in a project as discussed above. Different models can be applied to solve problems. However, when lots of models are combined, software can be buggy. Communication can become confusing within the team members. In order to solve these kind of challenges, context term comes to scene in domain Driven Design. A context is a certain part of the software or the work of the team. A bounded context is a group of solutions which have to be placed closely as possible in the software. "*A BOUNDED CONTEXT delimits the applicability of a particular model so that team members have a clear and shared understanding of what has to be consistent and how it relates to other CONTEXTS*" (Evans, 2003). Obviously, there is a unified understanding within the boundary context. However, it is not important what exists outside of the boundaries.

On the other hand, communication is obviously one of the problems of the software development world. There is a continuous need of translation between two different languages, business domain and development. Developers understand algorithms, data types, classes, computer terms and etc. Obviously these terms are not the exact translation of business problems. There should be a policy that solve these difficulties. The Ubiquitous language is defined for the domain model. It is used by developer and domain experts to understand each other. Thus, they can work on all the activities of creating software. This language does not have technical terms. It is all about the business domain and the terminologies from the business domain.

It is very important to use ubiquitous language within the bounded context. It is also suggested to divide the teams physically according to bounded contexts. It helps to prevent context switch between team members to understand the models better. Everyone have to learn basic concepts about the domains they are working on. For example; revenue, profit, market, competitive edge and etc.

Also there can be some ambiguities when we do not have bounded contexts. Suppose our software has two concepts as banking and web application. We have a term named account for both part of the software. However account represents a container of persons who keeps their money and other financial things in the bank. Also account means authentication, authorization, user information and credentials to log on to system in the web application perspective. As seen, same terms can means different logical information on the software. However if we divide to bounded contexts we will have two different entities with the same name in these contexts, not knowing each other.

How these multiple bounded contexts communicate with each other? The answer is application programming interfaces as APIs. Each context knows nothing rather except the interfaces.

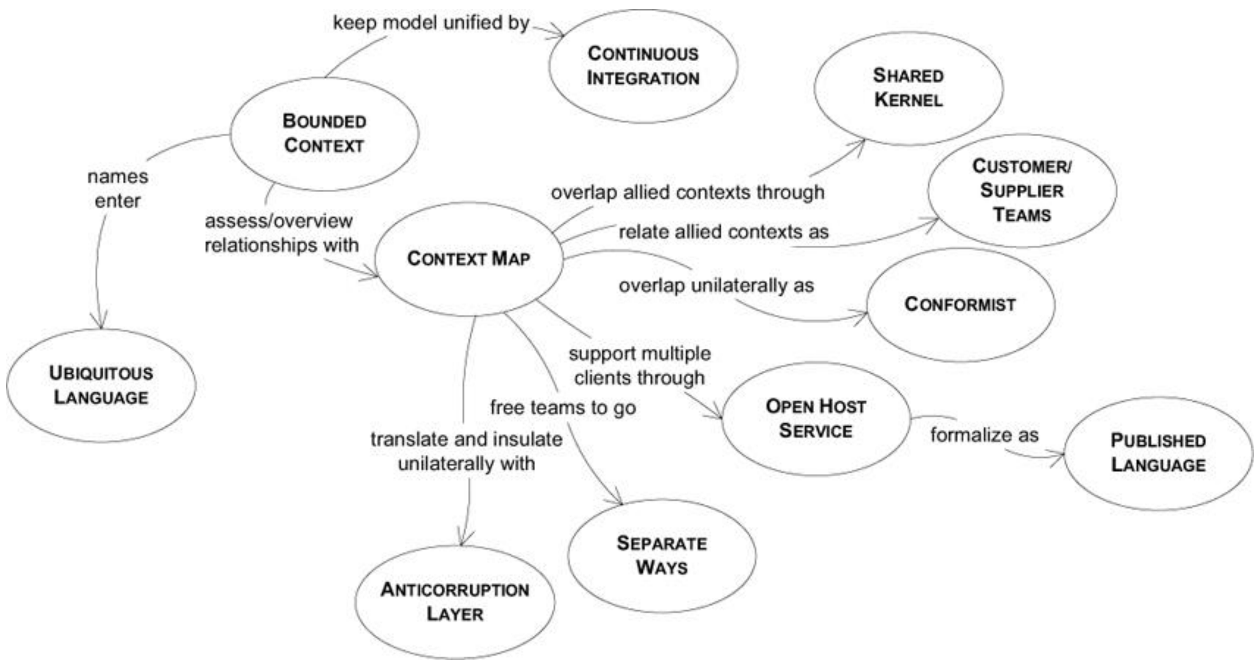

Figure 1. A navigation map for model integrity patterns (Evans, 2003)

Building Blocks

Building Blocks are also core components of Domain Driven Design. They are briefly explained below;

Entities are the identity objects. They are not traditional objects. They do not have only attributes but also some functionalities to provide continuity.

Value objects are different than entities. They are immutable means unchangeable objects having only attributes.

Domain events are discrete events related to model activities of the system. Most of the time, domain experts do care about only for these events.

Entities and value objects within a defined boundary creates an **aggregate**. Entities and value objects do not perform all the actions. Also, external objects do not have direct access to individual entities or value objects placed in the aggregate. External objects can pass instructions to aggregates to work on the items within.

There are some functionalities that are not fit in the entities and value objects. These functionalities are placed in the items called **services**. In most of the applications that Doman Driven Design is not used its implementation, number of services is high. In Doman Driven Design, since entities have functionalities not only services have less functionaries but also there are less services.

Most of the times entities represents the data existing in a collection like data bases or an existing aggregate. **Repositories** are used as interfaces to access these data from the collections.

Creation of objects can be differ from entity to entity, and also challenging. In order to encapsulate to creation of an event or value objects, usage of **factories** are suggested in Domain Driven Design. It is not a new thing, but it is highly recommended.

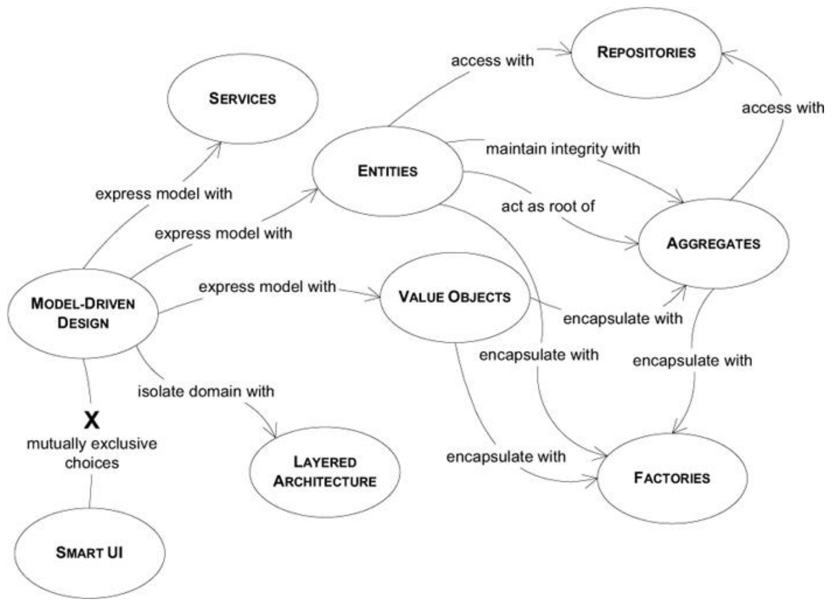

Figure 2. A navigation map of the language of MODEL-DRIVEN DESIGN (Evans, 2003)

Difference between Traditional Models and Domain Driven Design on Architectural Level

Before continue to advantage and disadvantages of Domain Driven Design, let us look into traditional anemic domain model.

Services, entities and data access objects creates different layers with each other. Entities mostly represent the real object from the data bases or any collection. They do not do anything but keeping the attributes of the entity. They have relationships between each other because of the inheritance.

On the other hand, services keep the all business logic within. Most of the time a service class needs another one to complete a business process. It creates dependencies between services. Also this makes service layer thicker.

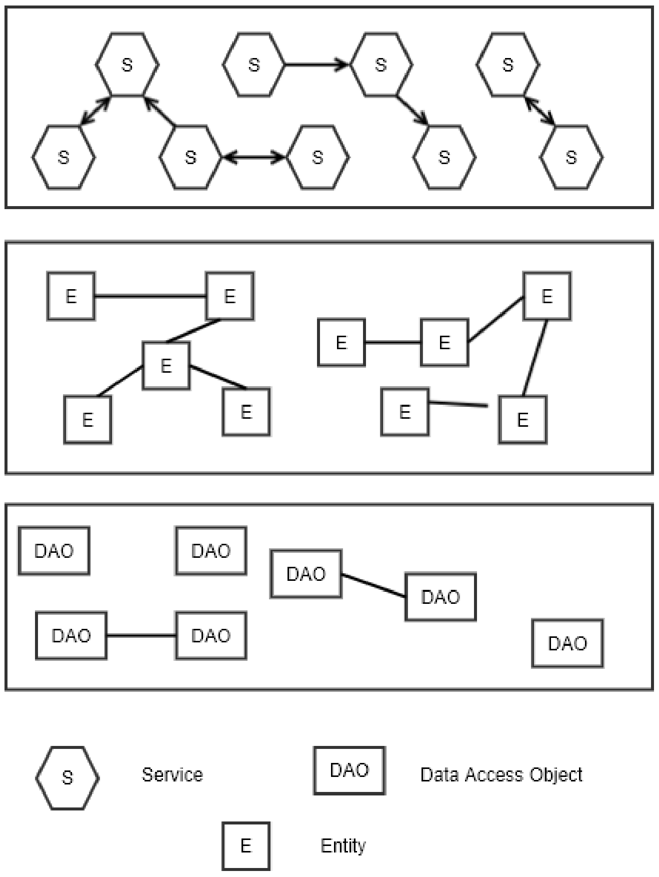

Figure 3 Anemic Domain Model

Data access objects help to retrieve data from the data base. Data access objects also need a relationship with each other. Most of the time one DAO represents one entity. This makes the need of relationship, since entities have with each other.

When we investigate Domain driven design, we see again services and entities on architecture. However they are placed to the layers with a different approach. First we see value objects, aggregates and repositories but no data access objects.

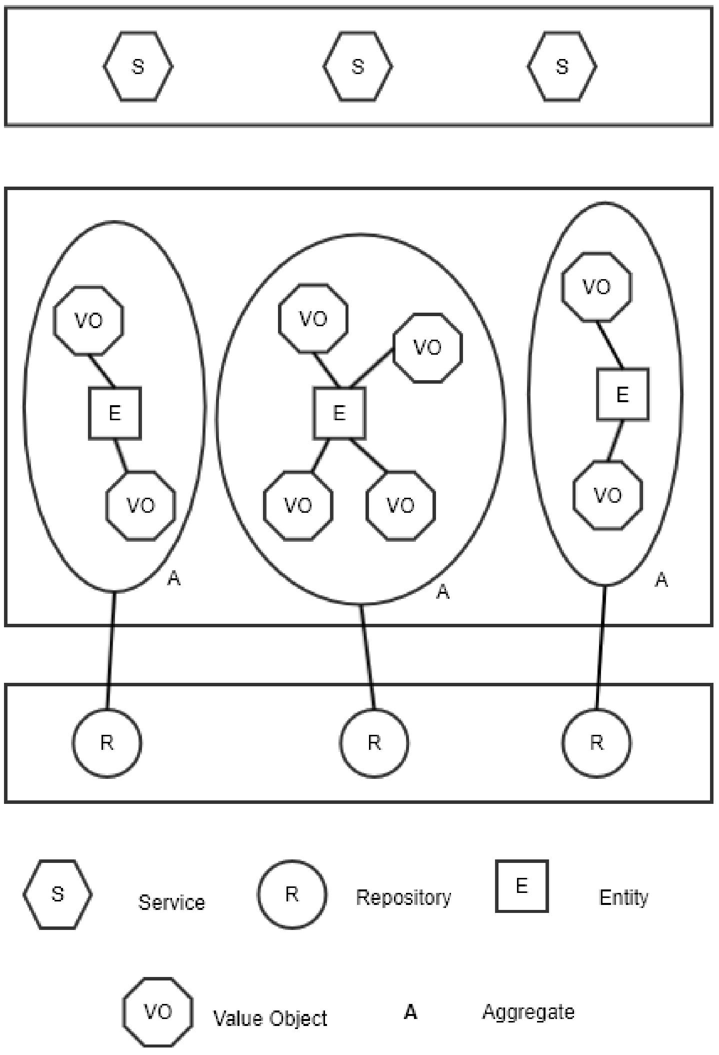

Figure 4 Domain Driven Design

In domain driven design we have less entities. Instead of nonfunctional entities we have value objects. As seen there are relationship between value objects and entities. They stay together in a group called aggregate. This aggregate represents a solution rather than a problem in the architecture. This grouping causes a thicker layer than anemic domain model has.

Entities have identity. Besides, some functionalities are taken from the services to entities. They can do some functions like basic comparisons, etc. This makes that existing services are focused on the business logic only. It also makes less services presents on software. Since basic functionalities taken from the services to entities and existence of less services, this layer is much thinner than anemic domain model has.

Every aggregate is related to a repository as seen from figure 4. This makes a thinner layer on repositories like the services have. All this architecture model makes easier to focus on a specific domain or subdomain for the developers. On domain perspective functionalities are more understandable and less buggy.

Advantages and Disadvantages of Domain-Driven Design

Advantages

Architecture

As seen at anemic model architecture, there is a thicker and bulky service layer. The reason is that entities are nothing but classes containing only attributes and relations with other entities. The other functionalities present in utilities, controllers, services and helpers which are placed in service layer. This causes less understandable and more buggy code. On the other hand, with domain driven design we have more functional entities within aggregates. This makes service layer more understandable. It is easier to for the developers to handle buggy conditions without harming other parties of the software. Encapsulation of entities and value objects makes our API simpler. Simple communication between aggregates, services and repositories reduce risk of wrong implementation of the process.

Usage of OOP

Object oriented programming is at the center of domain driven design. Almost everything in real life or domain can be represented with classes. System can be improved for future by adding new classes or changing them. Obviously usage of the OOP makes the enterprise application to be modular. It increases flexibility of the application

Focus on Smaller Chunks

Domains, subdomains and aggregate concepts are a perfect example of divide and conquer methodology. These concepts divides the complex problems into smaller ones which are easier to solve. Undoubtedly, small implementations are easier to develop and test. Also dividing the problem into smaller chunks makes domains more understandable. On the other hand, it is easier to use the some software principles and patterns which are accepted by community for the smaller problems or solutions.

Communication

For successful implementation, communication plays a critical role on domain driven design. Although there exists more back and forth communication between domain experts and developers, ubiquitous language makes easier to communicate than the other approaches. The business teams and developer teams can use this language to communicate about the business domain requirements, entities, and processes. The usage of common terms between developers and domain experts can create fast and reliable development with easier communication. Risk of misunderstandings is reduced with the usage of same language. Also coordination between team members gets better, since everybody uses the same terms through development process.

Disadvantages

Domain Experts

Domain Driven Design requires domain experts which is not easy to find always. Since they have valuable knowledge, it is expensive to hire them. Limited budget can cause to hire people who are not capable. This can cause communication leaks and misunderstandings between teams and wrong implementation of software. Sometimes there are lots of outside integration. This requires more than one domain expert.

Communication

When technical complexity increases on the project, it is getting challenging to find common language between all parties of the project. It makes harder to continue for domain experts and developers. *Domains that seem less technically daunting can be deceiving: we don't realize how much we don't know. This ignorance leads us to make false assumptions.* (Evans, 2003)

Effort

Obviously, domain driven design has high learning curve. Epically for the developers, it is hard to get used to the new way of thinking. Even if developers' learning curve is completed, there is still a time consuming process at the communication level. Developers have to spend lots of time with experts. With the complexity of the system, deciding domain and subdomains takes time.

Complexity of Domain

When complexity increases, it makes sense to divide problems into smaller domains and subdomains to find proper solutions. Domain Driven Design is the right choice when the software is huge and complex. However, communication with domain experts, deciding domains and subdomains, making architecture according to domains, all of these are over engineering for the smaller software.

Conclusion

Domain Driven Design is helpful to understand clients. It focuses on the problems of the clients and related solutions. It is a very powerful approach to make development on complex domains. For the practitioners, it makes easier to understand the idea behind the business domains that clients have. Especilly with the growth of cloud technologies and the micro service approach, we see that enterprise companies build their solutions by using Domain Driven Design. It work fine. Obviously there is a successful history for this approach.

On the other hand, does your company have enough budget, complex domain and time? If the answer is yes for all of them, definitely you should really consider to use it. Even if you have time and money, complexity plays key role. For example, if you have a basic CRUD operations on your software, it is over engineering to use domain driven design. However, if your application grows enough, it can be considered to use it. At this point, by refactoring the code to convert from the traditional model to domain driven design you can gain lots of benefits.

The last but not least, it is obvious that domain driven design has lots of powerful advantages. However, it is not a magical key that can open every door. Is it really necessary to use it? Do we really need this? Before start to using this approach, perhaps considering the need is the best way.

References

Application Design: Data-driven vs Domain-driven. (2018). Retrieved from https://passwork.me/info/blog/applicationdesign

Artemiuk, Z. (2017, April 18). *Refactoring from anemic model to DDD*. Retrieved from https://blog.pragmatists.com/refactoring-from-anemic-model-to-ddd-880d3dd3d45f

Berdychowski, K. (2017, Janurary 3). *Domain-Driven Design vs. anemic model. How do they differ?* Retrieved from https://blog.pragmatists.com/domain-driven-design-vs-anemic-model-how-do-they-differ-ffdee9371a86

Bogatinov, A. (2018, February 23). *DOMAIN DRIVEN DESIGN VS. OBJECT ORIENTED PROGRAMMING*. Retrieved from https://haselt.com/blog/domain-driven-design-vs-object-oriented-programming

Brandolini, A. (2009, November 25). *Strategic Domain Driven Design with Context Mapping*. Retrieved from https://www.infoq.com/articles/ddd-contextmapping

Definition of domain. (2018). Retrieved from https://en.oxforddictionaries.com/definition/domain

Ebertz, J. (2017, December 14). *A Brief Intro to Domain Driven Design*. Retrieved from https://austinstartups.com/a-brief-intro-to-domain-driven-design-326cf1669bbc

Evans, E. (2003). *Domain-Driven Design: Tackling Complexity in the Heart of Software.* Addison-Wesley Professional.

Ghanbari, E. (2013, August 16). *Domain Driven Design VS Model Driven Architecture*. Retrieved from https://ehsanghanbari.com/blog/post/35/domain-driven-design-vs-model-driven-architecture

Jain, A. (2016, October 20). *Domain Driven Design Advantages and Disadvantages*. Retrieved from http://www.abhijainsblog.com/2016/10/domain-driven-design.html

Martin, R. C. (2008). Clean Code. In R. C. Martin, *Clean Code.* Prentice Hal.

Sarna, A. (2018, May 1). *Is Shifting to Domain Driven Design worth your Efforts?* Retrieved from https://blog.knoldus.com/is-shifting-to-domain-driven-design-worth-your-efforts/

Vernom, V. (2012, November 9). *Getting Started with Domain-Driven Design*. Retrieved from http://www.informit.com/articles/article.aspx?p=1944876

Vernon, V. (2012, November 9). *The Challenges of Applying DDD*. Retrieved from http://www.informit.com/articles/article.aspx?p=1944876&seqNum=5