

**Emerging Trends in IT**

Year 3 (2022), Semester 2

***SCHOOL OF INFOCOMM TECHNOLOGY***

Diploma in Information Technology

**ASSIGNMENT 1: Domain-Driven Design**

|  |  |
| --- | --- |
| **Name** | **Nur Hakimi Bin Mohd Yasman** |
| **Teacher** | **Mr. Teo Wen Qiang, Wesley** |

# What is Domain-Driven Design?

Domain-Driven Design, commonly abbreviated as DDD, is a software development method that’s purpose is to simplify the process of creating complex applications. This is done through bridging related pieces of software (microservices) into an ever-evolving model. This concept was introduced and popularised by Eric Evans in 2004, when he released his book *Domain-Driven Design: Tackling Complexity in the Heart of Software*.

To get the full picture, there is a need to dive deeper into the initial definition. Domain, in a development context, is the subject area around which the application logic revolves. In other words, it is the sphere of knowledge and activity around what the application is intended to apply. Subsequently, when discussing the domain layer/logic, it refers to discussing the rules and logic governing the elements of an application. This concept builds upon another concept found in Object-Oriented Analysis and Design; The overarching rules that institutes how business objects interact with one another to create and modify data is the domain logic. There are usually multiple domains in an application, with their respective domain logic. Speaking of business objects, these are referred to as entities. Entities can change over time while maintaining the same identity – changing when presented under a different context. On another note, microservices are an architectural approach to software development where software is composed of small independent services, communicating through APIs. Each of these microservices have their own specific bounded contexts, configurations and dependencies. Microservices should not have more than one bounded context, so as to show its independence and ability to stand on its own.

Evan further lists out a couple of common terms that are useful when describing and discussing DDD practices, mentioned in detail by Vaughn Vernon in his book ‘Domain-Driven Design Distilled’. According to Vernon, DDD is essentially modelling a Ubiquitous Language in an explicitly Bounded Context, which is used as a foundation for creating a Domain.

Ubiquitous language is a methodology that refers to the set of words and terms being used by the team when they discuss about the domain they are working on; These terminologies are also reflected in the software models being developed. The entire team develops this language, ranging from the domain experts to the software developers, in order to be able to understand each other perfectly and be on the same page. This prevents the project from facing any serious troubles caused by disrupted language, such as misunderstandings that can arise from the different team members using their own jargon to talk about the domain. It is important that the ubiquitous language developed is strict, exact and tight.

Bounded contexts, on the other hand, are semantic context boundaries that contain the complexities of the application. Models and teams are implemented here, where only certain subdomains are defined and applicable. This way, other subdomains will not make sense when applied, as a function or entity may have different names under different contexts. Within this boundary, each segment of the software model has its definitive meaning and function. In the beginning, a bounded context can be seen as a problem space, where the team can design steps within a project’s constraints while performing high-level strategic analysis. As the project progresses and the model begins to take a clearer shape, the software model will begin to be translated into the project’s source code. Essentially, it will be where the solution is implemented to address the problem identified as the core domain.

The core domain refers to the software model that is seen as the most important above the rest; It is developed to distinguish an organization competitively from the rest, and should address a major line of business that the application is dealing with. This is where it would do project leaders good to choose deliberately what falls under the core domain, and what does not; The application can’t excel and specialise at everything after all. Therein lies DDD’s primary value proposition – project leaders should look to appropriately commit their best resources into the core domain.

Going back to Eric Evans, there are three core principles that Domain-Driven Design is centred around, highlighted in his book.

1. Focus on the core domain and domain logic

Developers should strive to protect the domain knowledge from any outside influence by other domains. The code and model for a domain should align and not stray from their given subdomains. Boundaries should be made explicit, so as to prevent two things – subdomains from hinging on each other’s structures needlessly and blurring the meaning of domain terms. Therefore, a laser focus should be placed onto not crossing the lines.

1. Base complex designs on models of the domain

Reflect the domain model in the code explicitly, by embedding the domain terminology in the code. This is done simply by using naming conventions followed by the respective domain experts, for code pieces like methods, classes, commands and most importantly, domain events.

1. Collaboration with domain experts to improve the application model and resolve any emerging domain-related issues

Everyone working on the project should strive to capture the domain model, using domain terminology, by communicating with domain experts. Get on the same wavelength as the experts in those domains and areas, in order to understand the problem from their point of view. This is how the foundation for a harmonious model is set, and ubiquitous language is established.

DDD has upsides, as well as downsides. The first advantage of DDD is easier communication between the team. Thanks to ubiquitous language, communication between the different teams becomes much more simpler; Less nuanced jargon is used in place for more comprehendible terms. Next, there is much more flexibility in the system, as DDD is object-oriented. With each object (bounded context, subdomain, etc.) being modular and caged, the entire system can be modified and upgraded regularly. Lastly, the domain will be given more importance than UI/UX. Rather than focusing on the interface, the application will tackle the domain and the users that are directly connected to it. Disadvantage-wise, deep domain knowledge is required. There needs to be at least one domain specialist on every team that is able to understand all the facets of the application. Additionally, it contains repetitive practice that follow the use of continuous integration, where some teams may find difficult due to experience with less-flexible work styles like the waterfall model. Lastly, it may not be suited for highly-technical projects, as it can be greatly challenging for business-oriented domain experts. These technically complex projects may cause limitations that are seemingly unsolvable for all team members.

Let’s dive into Shopee as an example. Shopee is an e-commerce platform that provides its users with a secure, quick and simple online shopping experience. It provides its services to already-existing brands such as Sony and Dyson alongside new shops and sellers from virtually anywhere in the world; This is described as having a consumer to consumer and business to business hybrid model. It is in own right a business to business marketplace as well, as it serves customers all around the region. Shopee started their business from a mobile application, rather than from a website, which was a deliberate decision after considering the high average usage of mobile internet in SEA. Now, Shopee has the highest number of downloads with the most active number of users and visitors in SEA. So, how does the Shopee mobile application implement DDD?

From the description of Shopee above, it is clear that the core domain of this application is the business domain, which is e-commerce – the buying and selling of goods using the internet, alongside the transfer of money and data to execute these transactions. This should be the main focus of the application, and what a majority of their resources should be allocated to, since its competency in this aspect is what led it to be one of, if not, the best e-commerce applications in SEA. To figure out what it’s subdomains and bounded contexts are, it would be good to start by listing out the main functionalities the app provides.

Customers should be able to create and login to an account, browse different stores and items, purchase their items and receive them through shipping and delivery. Sellers should be able to create and login to an account, put up listings of their items, and ship their products to their buyers. These are their most basic features, as is with every other e-commerce application. From here, we can outline the different bounded contexts are – account system, e-commerce system, inventory system. Most of the heavy details will be in the e-commerce bounded context, such as the business logic and resources. We can then outline the subdomains, that can fall under multiple bounded contexts. There is the product catalogue subdomain, orders subdomain, invoice subdomain, shipping subdomain, inventory subdomain, customer support subdomain etc. Like mentioned before, some subdomains can fall under multiple contexts. As an example, the inventory subdomain can fall under the inventory system and account system contexts, while the shipping subdomain can fall under the e-commerce and inventory system contexts. As entities are handled by different subdomains and thus move between contexts, they may change their form and value but their identity should remain the same throughout. This entails the entirety of the business domain.