

**Emerging Trends in IT**

Year 3 (2022), Semester 2

***SCHOOL OF INFOCOMM TECHNOLOGY***

Diploma in Information Technology

**ASSIGNMENT 1 – REPORT**

**Madhumitha Selvam – S10207330J**

**Duration :**  11 Nov 2022 to 9 Dec 2022

**Weightage :** 30% of total coursework

**Individual/Team :** Individual

**Due on Sunday, 18 Dec 2022, 11.59 PM (Week 10)**

**Penalty for late submission**:

10 marks per day (including Sunday and public holiday)

There is a total of 6 pages (including this page) in this hand-out.

|  |
| --- |
| ***WARNING***  ***If a student is found to have submitted work not done by him/her, he/she will not be awarded any marks for this assignment. Disciplinary action will also be taken.***  ***Similar action will be taken for the student who allows other student(s) to copy his/her work.*** |

Table of Contents

[1. Domain Driven Design 3](#_Toc122878517)

[1.1 What is Domain 3](#_Toc122878518)

[1.2 What is Domain Driven Design 3](#_Toc122878519)

[1.2.1 Principles of DDD 3](#_Toc122878520)

[1.2.2 Advantages of Domain-Driven Design 3](#_Toc122878521)

[1.2.3 Disadvantages of Domain-Driven Design 4](#_Toc122878522)

[2. Shopee 4](#_Toc122878523)

[2.1 DDD and Shopee 4](#_Toc122878524)

[2.1.1 How are they connected? 4](#_Toc122878525)

[2.2 Strategic Design 5](#_Toc122878526)

[2.2.1 Designing a Problem Space 5](#_Toc122878527)

[2.2.2 Designing a Solution Space 6](#_Toc122878528)

[2.2.3 Define Bounded Context 6](#_Toc122878529)

[2.3 Evolution 7](#_Toc122878530)

[2.3.1 How to detect new Subdomains 7](#_Toc122878531)

[2.3.2 Evolving and Evolution 8](#_Toc122878532)

[3. Summary 8](#_Toc122878533)

[4. References 9](#_Toc122878534)

# 1. Domain Driven Design

## 1.1 What is Domain

“Domain” in Domain-Driven Design officially refers to a “**sphere of knowledge and activity around which the application logic revolves**”. In other words, the “Domain” is what is commonly referred to as “business logic” in the software world (Grima, 2018).

## 1.2 What is Domain Driven Design

Domain-Driven Design (DDD) is a collection of principles and patterns that is used for designing software for complex business domains. Properly applied it can lead to software abstractions called domain models. These models encapsulate complex business logic, closing the gap between business reality and code (Microsoft, 2009) . It is about designing software based on models of the underlying domain (Fowler, 2014).

DDD provides an avenue to facilitate the development of highly cohesive systems through bounded contexts. Microservices is an implementation approach that encourages you to focus your service boundaries on the business domain boundaries (Vennaro, 2017).

It promotes communication and flexibility on a project that isn't overly technical. Moreover, DDD is not directly about technology, it gives it a clear direction and a flow for the software. Domain Driven Design (DDD) brings business and IT closer and concentrates software development around having authentic insights into the processes to be automated.

### 1.2.1 Principles of DDD

DDD focuses on three main core principles

* Focus of core Domain and Domain logic
* Base complex designs on models of the domain
* Collaboration with domain experts to improve the application model and resolve any emerging domain-related issues.

Evans' *Domain-Driven Design* further defines a few common terms that are useful when describing and discussing DDD practices:

* **Context:**The setting in which a word or statement appears determines its meaning. Statements about a model can only be understood in a context.
* **Model**: A system of abstractions that describes selected aspects of a domain and can be used to solve problems related to that domain.
* **Ubiquitous Language**: A language structured around the domain model and used by all team members to connect all the activities of the team with the software.
* **Bounded Context**: A description of a boundary (typically a subsystem or the work of a specific team) within which a particular model is defined and applicable (Banks, 2022).

### 1.2.2 Advantages of Domain-Driven Design

**Eases Communication**: With an early emphasis on establishing a common and ubiquitous language related to the domain model of the project, teams will often find communication throughout the entire development life cycle to be much easier. Typically, DDD will require less technical jargon when discussing aspects of the application since the ubiquitous language established early on will likely define simpler terms to refer to those more technical aspects.  
**Improves Flexibility**: Since DDD is so heavily based on the concepts of object-oriented analysis and design, nearly everything within the domain model will be based on an object and will, therefore, be modular and encapsulated. This allows for various components, or even the entire system, to be altered and improved on a regular, continuous basis.  
**Emphasizes Domain Over Interface**: Since DDD is the practice of building around the concepts of domain and what the domain experts within the project advise, DDD will often produce applications that are accurately suited for and representative of the domain at hand, as opposed to those applications which emphasize the UI/UX first and foremost. While balance is required, focusing on the domain means that a DDD approach can produce a product that resonates well with the audience associated with that domain (Banks, 2022).

### 1.2.3 Disadvantages of Domain-Driven Design

**Requires Robust Domain Expertise**: Even with the most technically proficient minds working on development, it's all for naught if there isn't at least one domain expert on the team that knows the exact ins and outs of the subject area. In some cases, domain-driven design may require integrating one or more outside team members who can act as domain experts throughout the development life cycle.  
**Encourages Iterative Practices**: While many consider this an advantage, it cannot be denied that DDD practices strongly rely on constant iteration and continuous integration to build a malleable project that can adjust itself when necessary. Some organizations may have trouble with these practices, particularly if their past experience is primarily tied to less-flexible development models, such as the waterfall method.  
**Ill-Suited for Highly Technical Projects**: DDD is not very well-suited for applications with marginal domain complexity but has a great deal of technical complexity. Since DDD so heavily emphasizes the need for (and importance of) domain experts to generate the proper ubiquitous language and then domain model on which the project is based, a project that is incredibly technically complex may be challenging for domain experts to grasp, causing problems down the line, perhaps when all members of the team did not fully understand technical requirements or limitations (Banks, 2022).

# 2. Shopee

Shopee is a Singaporean multinational technology company that specialises in e-commerce. In this section we will be looking at how Shopee and Domain Driven Design are related. Find out any bottlenecks across the various steps in the process or missing links.

## 2.1 DDD and Shopee

The heart of high-performing software is about more than just efficient code and robust integrations. Understanding the environment in which the code has to do its work, 'the domain', plays a crucial role in its success. When starting to develop a software, many think about the data structures first to design the DB tables and connected classes and modules.

However, DDD offers to reflect knowledge about the business domain to software design by using a set of techniques and design patterns.

### 2.1.1 How are they connected?

For Shopee, **User** (customer) places an **order**. And the **product owner** ships the product based on the **order invoice** of the user from a **bigger list of Invoices**. The application will be designed based on data structures.

On the surface, we can see the same Order data structure used in two applications, User and ProductOwner.OrderInvoice, payment method, and price of the order are presented to both the customer and product owner. Thus, the whole solution could be designed as a set of connected tables with the Order Invoice table in the core.

Once setting the above logic, the appropriate programming code units: modules, classes, and interfaces are added.

Since the data to be presented to both types of users has much in common, the code could partially have generic interfaces and shared behaviour. Thus, Shopee will be design as a single, all-encompassing model.

It could be dangerous because while the complexity grows with the number of entities and their relationships, each specific concept of Passenger and Driver will require functionality with more specific responsibilities (Chatuev, 2020).

For the user, the Order Invoice could have attributes such as Delivery Address and Price. The delivery address is the user’s point of contact and shipment destination and the end of the total order. Price includes all the fees to the product owner, the delivery cost, and the application fee, and is processed as a money transfer from the Passenger’s account to the application’s account.

For the product owner, the price shown to the user is not presented to the Product owner, but some margin is.

We can see that data of single Order needs lots of pre-processing before being presented to different parties and has different meanings depending on the context. Without careful design, the system has chances to grow into a Big Ball of Mud — a monolithic application consisting of tightly coupled modules with lowly cohessed elements (Chatuev, 2020).

## 2.2 Strategic Design

Strategic design is a set of principles and patterns for maintaining model integrity, distilling business domain models, and working with multiple models. Strategic design is very useful in breaking down large and complex business problems into multiple parts with clear boundaries and specific responsibilities to build a high-level software design topology.

Steps for a Strategic Design:

1. Problem Space
2. Solution Space
3. Define Bounded Context

### 2.2.1 Designing a Problem Space

Business Domain of Shopee is the problem we are trying to solve with a software effort. Thus, the domain is the problem space that is being addressed.

A domain can be split into Subdomains, e.g., Order, Identity Management.

To build a business domain, first, modularize the problem area by dividing it into subdomains. Dividing a business domain into subdomains is not easy. The biggest challenge in splitting is related to handling boundaries. How to identify the boundaries of each context, areas of overlap, and how to handle these areas.

Diagram

Description automatically generatedThe next step after we have done designing a Problem Space, is to design a Solution Space. In order to design a solution space, we need to match each subdomain to an appropriate Bounded Context.

Figure . Shopee App Problem Space

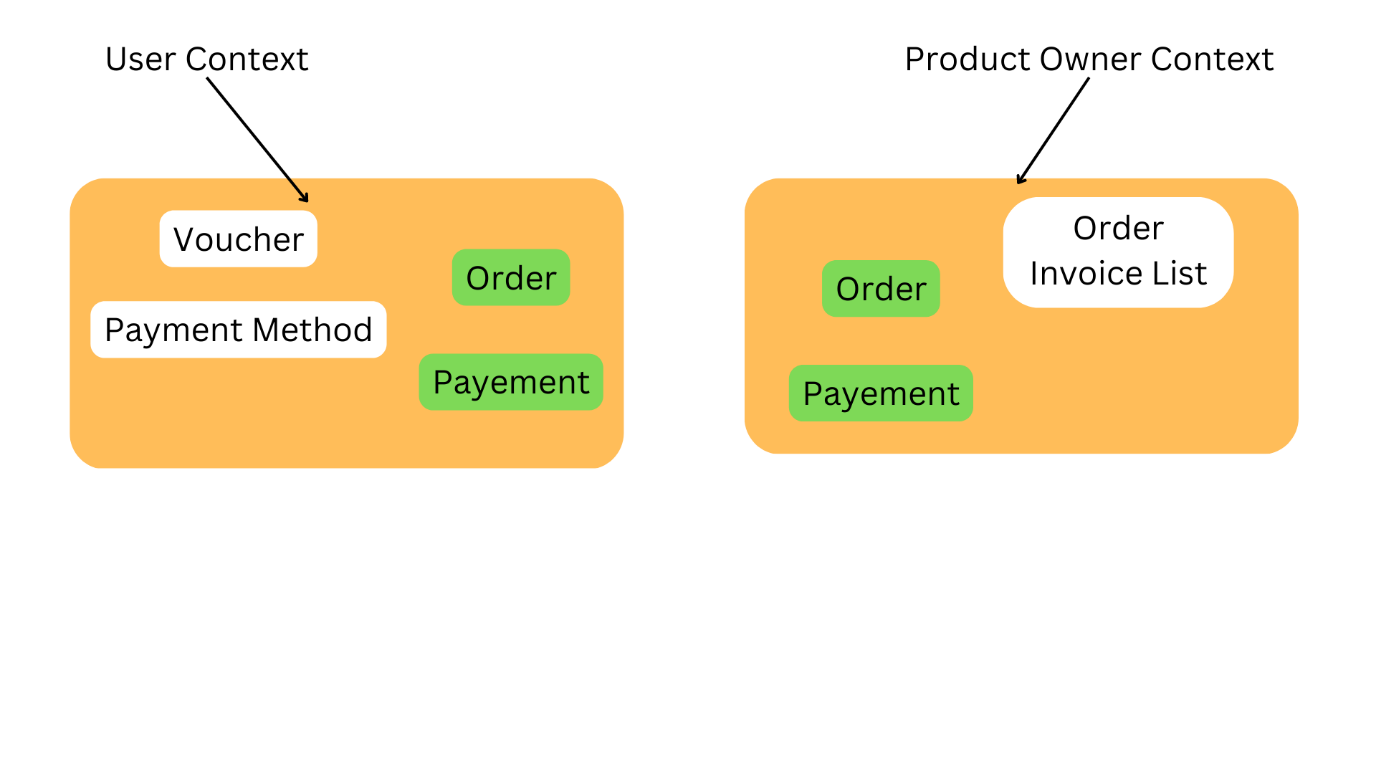
### 2.2.2 Designing a Solution Space

In DDD, a Subdomain in the problem space is ideally mapped 1-to-1 to a Bounded Context in the solution space (Chatuev, 2020).

To analyse and solve the overlapping part of the model is Shared Kernel Pattern. Shared kernel is used when several Bounded Context. Shared kernel is used when several bounded contexts can share one or many common entities. Using Shared kernel in our case would mean a direct matching of subdomains and concepts from *Figure1* into Bounded Contexts and Entities with shared Order Invoice and Payment.

The second option is a distinct Bounded Context holding specific entities. From the perspective of microservices architecture, the second option is usually more preferred and sustainable in the long-term, as any code unit tends toward a single responsibility principle.

The below picture shows Shopee’s Problem space is abstracted with distinct Entities living inside their own ‘Split Bounded Context’.



### 2.2.3 Define Bounded Context

This is where the DDD concept of *bounded contexts* comes into play. A bounded context is simply the boundary within a domain where a particular domain model applies. Looking at the previous diagram, we can group functionality according to whether various functions will share a single domain model.

Bounded contexts are not necessarily isolated from one another. In this diagram, the solid lines connecting the bounded contexts represent places where two bounded contexts interact. For example, Shipping depends on User Accounts to get information about customers (Martinekuan, 2022).

## 2.3 Evolution

With time Shopee have grown from a small start up to what it is now currently. As they are upgrading, different subdomains and concepts inside them will also be updating. Existing concepts, like Order Invoice or Order Invoice List could also expand into a new Back-Office subdomain. Even the whole product could extend to a B2B market. As a result, the problem space will become more complex and require further iterations of investigation and matching to bounded contexts. So, with further iterations of business evolution, we need to loop over the problem space to find new subdomain and apply Strategic DDD again.

### 2.3.1 How to detect new Subdomains

If the team starts using new terms to express an existing concept or an existing term is used to express a new meaning, then it is a red flag for the team and the subdomains’ boundaries will be overlapping.

A growing business domain is expressed orally and shared with an extended vocabulary of domain-specific terms, called the ubiquitous language. UL is documented and spoken by all employees in the subject area involved in the software development process. In other words, it is called ubiquitous because it is used ubiquitously by engineers and domain experts.

Example in Shopee Context using Ordering Concept:

* User makes an Order with the attributes: User Name, Delivery Location, Delivery Type, Price, Payment Method, etc.
* The order placement triggers the creation of an Offer which I added to the Offers list monitored by the Product Owner and Back-Office. Offer has Price for Product Owners, Total Price, Special Requests, User personal details like Phone, and Name.
* The product Owner accepts the order and starts their shipment process and managers the whole A-B Delivery process. The delivery has Arrival Date, Final Price and many other.

Reflecting on the evolving behaviour and attributes, the Ordering concept is starting to have different meanings under User, Product Owner, and newly appeared Back-Office subdomains. So, the Order is naturally transforming and could be split into Order, Deliver, and Offer.

Now, the iterated problem space looks like this

Diagram, venn diagram

Description automatically generated

### 2.3.2 Evolving and Evolution

Shopee started as a consumer-to-consumer (C2C) marketplace, but it pivoted into a C2C and business-to-consumer (B2C) hybrid model. On each iteration of the domain discovery, it’s crucial to make the **UL** reach. So, the new meanings of concepts and new subdomains could be detected and reflected in future system design.

# 3. Summary

Strategic DDD patterns are used to design business domain model abstractions that include behaviour and data. Collecting and using the ubiquitous language can detect changes in current business subdomains and the emergence of new subdomains. By structuring bounded contexts and their relationships, you can build high-level topologies for modular software design. A modular structure helps you set up microservices with clear responsibilities, which improves system maintainability and developability.

# 4. References

Banks, F. (2022, December 10). *Domain-Driven Design: What is it and how do you use it?* Retrieved from Airbrake Blog: https://blog.airbrake.io/blog/software-design/domain-driven-design

Chatuev, M. (2020, December 10). *Strategic Domain-Driven Design. In a set of articles I decided to… | by Magomed Chatuev | Medium*. Retrieved from Medium: https://medium.com/@chatuev/ddd-for-microservices-4778a363c071

Evans, E. (2003). Domain-Driven Design. In E. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software* (p. 446). Addison-Wesley Proffessional.

Fowler, M. (2014, December 10). *Bounded Context*. Retrieved from martinFowler.com: https://martinfowler.com/bliki/BoundedContext.html

Grima, L. (2018, December 10). *An introduction to Domain-Driven Design | by Laurent Grima | inato | Medium*. Retrieved from Medium: https://medium.com/inato/an-introduction-to-domain-driven-design-386754392465#:~:text=%E2%80%9CDomain%E2%80%9D%20in%20Domain%2DDriven,logic%E2%80%9D%20in%20the%20software%20world.

Martinekuan. (2022, December 10). *Domain analysis for microservices - Azure Architecture Center | Microsoft Learn*. Retrieved from Microsoft Learn: Build skills that open doors in your career: https://learn.microsoft.com/en-us/azure/architecture/microservices/model/domain-analysis

*Microsoft*. (2009, December 10). Retrieved from Best Practice - An Introduction To Domain-Driven Design: https://learn.microsoft.com/en-us/archive/msdn-magazine/2009/february/best-practice-an-introduction-to-domain-driven-design

Vennaro, N. (2017, December 10). *From domain-driven design to microservices | InfoWorld*. Retrieved from InfoWorld: https://www.infoworld.com/article/3243578/from-domain-driven-design-to-microservices.html#:~:text=DDD%20provides%20an%20avenue%20to,on%20the%20business%20domain%20boundaries.