What we will talk about?

- DDD Domain Driven Design
- Ubiquitous Language
- Domains / Subdomains [core | support | generic]
- Domain Expert
- DDD Strategic Tools
- DDD Tactical Tools
- Subdomains Matrix
- Bounded Context
- Context Map
- Agile
- Event Storming
- Rich / Anemic model

What is Domain Driven Design

Domain-Driven Design (DDD) is an approach to software development that emphasizes aligning the codebase closely with the business domain. By focusing on the core domain and modeling real-world business processes, DDD helps create systems that are both scalable and maintainable

Key concepts include:

- **Domain & Ubiquitous Language:** Aligning technical terminology with business language to ensure clarity and consistency between developers and stakeholders.
- **Bounded Contexts:** Breaking down large domains into smaller, manageable contexts to avoid complexity and confusion.
- **Entities, Value Objects, Aggregates:** Structuring core business logic around meaningful models, where entities have identity and value objects represent immutable concepts.
- **Domain Events and Repositories:** Managing changes in the domain and ensuring proper data access through a clean, event-driven architecture.

Why DDD?

- Software should not make sens only for coders but also for the business,DDD empathises making sure business and software talk same language
- Software priorities are aligned with business priorities
- With DDD everybody learns and contributes discovering the business domain
- Knowledge no longer belongs to just developers
- There are no translations between domain experts, meaning no information loss or tedious syncing, everyone talks same language
- The design is the code and the code is the design, the only implemented truth for the common language focused on delivering software continuously through agile discovery processes

Should I start considering DDD as an option?

Depends on the project and business DDD is not a magical solution that fits any business:-

When to not consider DDD as option

- If your application is data-centric and use cases evolve around data manipulation and crud operations [create , read , update ,delete] you don’t need DDD the only thing your company need is a fancy face in front of your database
- If your application has less than 30 use cases, it might be simpler to use framework like symfony or laravel to handle your business logic

When to consider DDD as option

- If you have more than 30 use cases, if you know for sure your system will grow in complexity you should start considering using DDD to fight complexity
- If you know your application is gonna grow and is likely to change often DDD will definitively help in managing the complexity and refactoring your model overtime
- If you do not understand the domain you are working on because it is new and no body invested on a solution before this might mean it is complex enough to start applying DDD you will work closely with domain experts to get the models right

What is the business value of using DDD?

- Useful and meaningful model of its domain
- Domain experts contribute to software design
- Better experience
- Clear boundaries
- Better architecture organization
- Iterative and continuous modeling on agile fashion
- Better tools strategic and tactical

I said DDD use agile fashion but what is agile?

Agile is system that company follows to manage projects and tasks and project versions

The idea that agile separate our project into sprints each sprint can be 2 to 3 weeks long depends on company needs for provide new release

What is sprint?

Sprint is time duration that we set goals and features to do throw this period each day we join meeting talking about what is done what to work on today what at the end of sprint we do something called retrospective which is re check for or steps throw the sprint period to check if we where working right following the sprint goals and to learn what we did wrong and what is right

Example for tools helps to use agile :

Jira - zoho - asana

DDD and crud?

Can we count all crud operation as data manipulation only?

Answer is no maybe the change is happening as a part of a larger business process

Examples :

company A (DDD)

- The manager only will lookup the employees record n the system and select terminate employment contract button
- The system asks for the termination data and reason,the system automatically updates the employee records revokes the user credentials and electronic offic key and finally sends notification to the payroll system

DDD and crud?

Company B (crud)

The manager will do the following:

- Looks up the employees record in the system
- Puts check in the contract terminated checkbox and enter the termination date and reason then click save
- Logs into user management system looks up the users account puts a check in the disabled checkbox and click save
- Logs into the office key management system looks up the user key puts a check in the disabled check box and click save
- Sends an email to the payroll department notifying them that the employee has quit

What is DDD domain and subdomain?

Domain

- **Definition**: The domain is the subject area of the application, the "problem space" you're solving for. It includes the entire business logic that the software is intended to model.
- **Example**: In an e-commerce application, the domain would include concepts like customers, orders, products, payments, and shipping.

Subdomain

- **Definition**: A subdomain is a smaller part or subset of the larger domain. Complex systems often have multiple subdomains, each representing a specific part of the overall business logic. Subdomains help divide a large domain into more manageable sections.

Subdomain matrix

Core :-

- Competitive advantage
- Complex
- Continuous development
- Development method [internal – best talents]

Example : Order management for e-commerce

Generic :-

- Not giving competitive advantage
- Complex (complex but not unique or business-critical, so it's common to rely on existing solutions)
- Development method [outsource , bought]

Example : Payment processing

Supporting :-

- Not complex
- Not giving competitive advatage
- Development method [internal , does not required best talents]

Example : Customer support

DDD Phases (strategic)

- Subdomains
- Bounded contexts
- Event storming
- Ubiquitous language
- Relationship between bounded contexts
- Contexts map
- Context map team collaboration types/groups
- Anticorruption layer(ACL)
- Up & down streams(u/d)
- Open host service (OHS)

Bounded context

Definition: A bounded context is a logical boundary within which a specific domain model is applied consistently. Each context has its own rules, meaning, and logic that are isolated from other contexts.

Purpose: To prevent ambiguity and overlap in domain models, ensuring that different parts of the system (like Order Management, Shipping, Billing) maintain clarity and independence.

Key Benefit: Promotes modularity, improves scalability, and avoids conflicts in the system by keeping contexts separate.

Example:

In an **e-commerce platform**, we may have multiple bounded contexts:

Order Management Context:

- Focuses on handling orders, tracking statuses (pending, shipped, delivered), and managing order details.
- The concept of an "Order" here involves status changes, tracking history, and order fulfillment.

Billing Context:

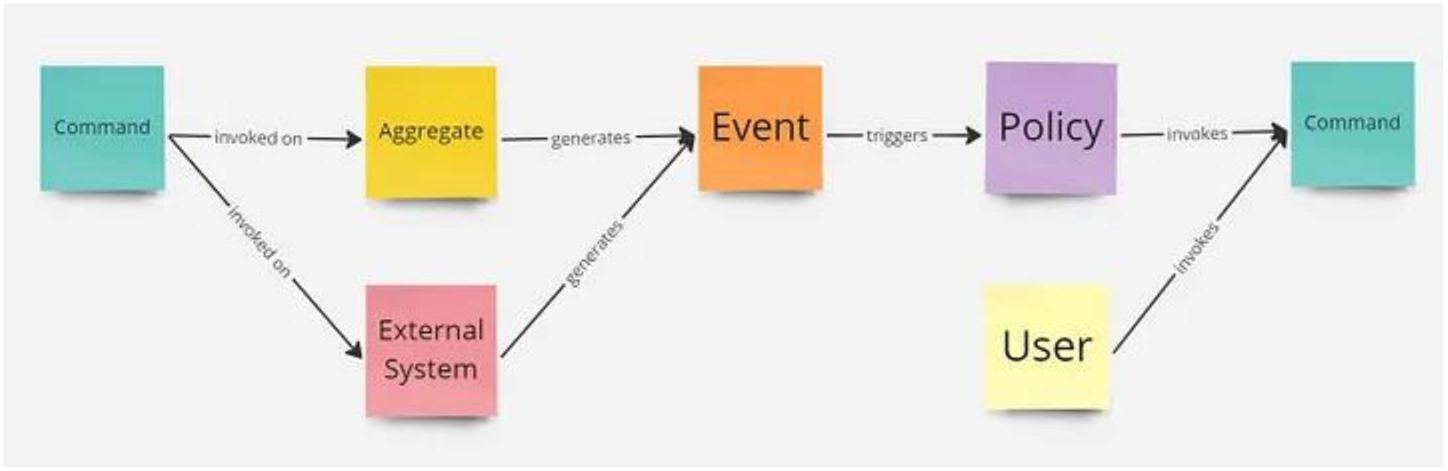
- Handles payment processing, invoices, and transaction history.
- "Order" here is only seen in terms of the amount to be paid, taxes, and payment methods.

Both contexts deal with "Order," but within their own boundaries and perspectives. The **Order** in the Order Management context is more about tracking its lifecycle, while in Billing, it's about financial transactions. The separation avoids confusion and allows each context to evolve independently.

Event storming

Definition: Event Storming is a visual, collaborative method for exploring and understanding a domain by mapping out the events, commands, and entities involved in a business process

Purpose: To facilitate a shared understanding of the domain, uncover hidden complexities, and drive design decisions by engaging stakeholders in the modeling process the whole idea is to bring people with the questions and people who know the answer in the same room and to build a model together



Ubiquitous language

- Ubiquitous Language is a core concept in Domain-Driven Design (DDD), emphasizing the importance of using a consistent, shared language between developers and domain experts. This language is created from the business domain and is used across all parts of the project, from discussions to code. The goal is to reduce misunderstandings and ensure that both technical and non-technical stakeholders are on the same page.
- **Consistency**: Every term and concept used in the code mirrors the real-world business terminology. This consistency simplifies communication and reduces ambiguity.
- **Collaboration**: By adopting a shared language, developers and domain experts can collaborate more effectively, ensuring that the code accurately reflects the business rules.
- **Code Readability**: The language is present not only in conversations but also in the codebase, making the system more readable and maintainable for anyone familiar with the domain.
- **Example**: In an e-commerce domain, instead of using generic terms like data, use business-specific terms like Order, Cart, and Payment. This way, the language is meaningful to both developers and business experts, improving the overall system design.

Relationship between bounded context

- Shared kernel
- Customer supplier
- Conformist
- Anticorruption layer(ACL)
- Open host service (OHS)
- Up/down stream
- Published language

Shared kernel

When two teams share part of their domain model and must collaborate to keep it aligned

Scenario: Suppose you have a system with two bounded contexts: **Customer Management** and **Order Processing**.

Customer Management: This context handles customer details, including customer profiles, addresses, and contact information.

Order Processing: This context handles order creation, processing, and fulfillment.

Shared Kernel: One common component between these contexts is the Address model. Both contexts need to use an Address object but with slightly different requirements.

Shared Kernel Component: Address

- **Fields**: street, city, state, zipCode, country
- **Behavior**: Validation rules, formatting, and basic operations like comparison.

Customer supplier

One context provides a service or data to another context (supplier) while the consuming context depends on it (customer)

Scenario: E-Commerce Platform

Bounded Contexts:

Order Management (Customer)

Inventory Management (Supplier)

Description: In an e-commerce platform, the **Order Management** context is responsible for handling customer orders, including creating, updating, and tracking orders. The **Inventory Management** context is responsible for managing stock levels, updating inventory, and handling stock replenishment.

Relationship:

Order Management (Customer): This context needs to know about the availability of products to complete the order process. It relies on the **Inventory Management** context to provide real-time stock information and ensure that products are available before finalizing the order.

Inventory Management (Supplier): This context provides stock levels and product availability information to the **Order Management** context. It supplies the necessary data for order validation and updates when stock levels change.

conformist

One context depends on another and has no choice but to adapt to the model of the other context

Example Scenario: E-commerce System

Context B: Inventory Management

This context manages product inventory, including product quantities, availability, and stock levels.

The model in this context includes entities such as Product, Stock, and Warehouse.

Context A: Order Processing

This context handles customer orders, including order creation, payment processing, and shipment.

The Order entity in this context needs to know about product availability to ensure that items are in stock before confirming an order.

Conformist Relationship

Inventory Management (Context B) defines the product model, including attributes like product_id, stock_level, and warehouse_location.

Order Processing (Context A) relies on the Inventory Management model to check product availability. When an order is placed, the Order Processing context needs to validate that the products are in stock. Here’s how the conformist relationship works:

Conformance: The Order Processing context conforms to the Inventory Management context's model. It queries the Inventory Management context to get the current stock levels and other product-related information.

Integration: Order Processing does not modify or have control over the Inventory Management model. It simply uses the provided data to make decisions (e.g., to check if a product is available).

Data Exchange: The two contexts exchange data through well-defined APIs or interfaces. For example, Order Processing might call an API exposed by Inventory Management to check stock levels or retrieve product details.

Anticorruption layer

This pattern is used when one context needs to interact with another but prevents its model from being polluted by the other context model the acl translates between models

or when domain interact with external systems or third-party systems. It ensures that the integrity of your domain model is maintained and that your system can interact with external systems in a controlled manner.

Open host service

A bounded context exposes a well defined interface for others to integrate with so we can say **(OHS)** is a pattern used to define a well-defined interface or API that allows other parts of the system or external systems to interact with a particular domain model. It’s a way to expose domain functionality and make it available to other domains or systems.

Up/down stream

Upstream: Refers to components or services that provide data or functionality to other components. They are "upstream" because they are higher in the data flow or dependency chain.

Downstream: Refers to components or services that consume data or functionality provided by upstream components. They are "downstream" because they are lower in the data flow or dependency chain.

Example:

Consider a simplified e-commerce system with a Core Domain focused on **Order Management** and a Supporting Domain for **Customer Notifications**.

Core Domain: Order Management

- This domain handles all the logic related to managing orders, including creating, updating, and tracking orders.
- It may also manage inventory and pricing information.

Supporting Domain: Customer Notifications

- This domain handles sending notifications to customers about their orders, such as order confirmations, shipping updates, and so on.

Upstream/Downstream Example:

Upstream: In this case, the **Order Management** domain is upstream for the **Customer Notifications** domain. The **Order Management** domain provides data and events related to orders, such as "Order Created" or "Order Shipped."

Downstream: The **Customer Notifications** domain is downstream because it consumes the events and data provided by the **Order Management** domain to send appropriate notifications to customers.

Published language

Definition: Published language refers to the specific subset of the ubiquitous language that is formally documented and communicated to external systems, APIs, or stakeholders.

Purpose: It ensures that external interactions and integrations use the same terms and definitions as those agreed upon within the team, providing a clear and consistent interface for communication.

Scope: It is particularly important in contexts where the system interacts with other systems or external stakeholders. It is reflected in API documentation, data contracts, and integration specifications.

DDD Phases (tactical)

- Value object
- Entities
- Aggregates
- Domain events
- Domain services
- Repositories
- Factories
- [Rich | Anemic] model
- CQRS

Value object

Value object are objects that represent a concept idea or characteristics (صفات) in the domain model they do not have an identity and are defined by their attributes

Characteristics :-

- Equality
- Immutability
- Side effect free

Equality means that if instance from value object class comparison with another value object instance that have same data it should be equal as example

Address class which is our value object include state , city , zip code

If I made two instance from address class and in both class set the same value for object one and object two it should be equals to each others

Immutability means that once object is created the values and data that object contains can not be changed any operation that seems to "modify" a value object actually returns a new instance with the modified values rather than changing the original

Side effect free means that value object does not cause any changes to the system state or trigger any side effects (such as modifying a database, sending a message, or interacting with other objects).

Entities

Entity is an object that has a distinct identity that remains constant throughout its life time even if it's value changed entities are used to model concepts in the domain that have a unique identity and are distinguishable (مميز) from other objects

Characteristics :-

- unique identifier
- Mutable
- Entities encapsulate behavior that operates on their own state and may collaborate with other objects in the domain

Mutable means that the entity data can be changed

Aggregates

Aggregate is a cluster of related domain objects (like entities and value objects) that are treated as a single unit for data changes. The **Aggregate Root** is the main entity in this group, responsible for enforcing the integrity of the entire aggregate

Characteristics :-

- The aggregates is created retrieved and stored as a whole
- The aggregate is always in a consistent state
- The aggregate root whose ID is used to identify the aggregate it self

Example :

Order which is our **aggregate root** includes entities like customer and product and value object like payment method

Domain events

Domain Events are significant occurrences within the domain that reflect a change in the state of the business. These events are essential because they convey that something meaningful has happened, allowing other parts of the system to react or adapt to these changes without direct coupling between components

- **Capturing Meaningful Business Changes**: Domain events should represent something that is important in the business context, such as "Order Placed" or "Payment Completed".
- **Immutable**: Once a domain event occurs, it cannot be changed. It is a record of something that has already happened.
- **Raised by Aggregate Roots**: Aggregates are responsible for ensuring business rules are enforced, and when certain actions or changes occur, the aggregate root raises a domain event.
- **Cross-Context Communication**: Domain events allow different bounded contexts to communicate in a decoupled way. They let one part of the system react when something important happens in another part.

Domain services

Domain Services represent operations or logic that don't naturally belong to any particular Entity or Value Object, yet they are still essential to the domain. These services encapsulate business rules and domain-specific operations that involve multiple entities, aggregates, or value objects

Characteristics :-

- Stateless
- Focus on domain logic
- Expressive

Stateless means they don't have any internal state. They operate on objects or aggregates passed to them

Focus on domain logic means they encapsulate domain logic that doesn't fit inside entities or value objects

Expressive means their names and methods clearly represent domain concepts and behaviors

When to Use a Domain Service:

- When an operation spans multiple entities or aggregates.
- When the logic doesn't fit well into any entity or value object.
- When you're dealing with pure domain logic that isn't tied to infrastructure concerns (like sending emails or writing to a database).

Example:

Let’s consider an e-commerce system. Suppose we have entities like Order and Payment. In this case, the business logic for processing a payment might span both of these entities, but it doesn’t naturally belong to either one. This is a good fit for a **Domain Service**.

Repositories

repositories play a crucial role in managing the persistence of aggregates. A repository is responsible for retrieving and storing aggregates in a way that abstracts the underlying persistence mechanism (like a database or an external service), allowing the domain model to remain clean and focused on business logic

Characteristics :-

- Aggregate Management
- Persistence Abstraction
- Collection-like Interface

Aggregate Management means repositories are used to retrieve and save aggregates, not individual entities or value objects. This ensures that the aggregate's consistency is maintained

Persistence Abstraction means repositories hide the details of data storage (e.g., whether it's in SQL, NoSQL, or another storage medium). This decouples the domain model from the infrastructure layer

Collection-like Interface means a repository often behaves like a collection (e.g., with methods such as add, remove, find, etc.), allowing developers to interact with aggregates in an intuitive way

Repository Responsibilities:

- **Retrieving Aggregates**: Repositories retrieve complete aggregates from the underlying storage based on certain criteria, typically by their unique identity (e.g., findById).
- **Saving Aggregates**: When an aggregate is modified, the repository is responsible for persisting those changes back to the storage.
- **Maintaining Aggregate Boundaries**: Since aggregates enforce consistency rules within their boundaries, the repository ensures that only valid aggregates are retrieved and stored.

Factories

factories are responsible for creating complex aggregates or entities. The primary role of a factory is to encapsulate the creation logic of objects that are complex, require dependencies, or have multiple invariants to be satisfied at the time of creation. Factories abstract away the intricacies of instantiating these objects, ensuring that the system remains focused on business logic rather than object creation details

Characteristics :-

- Encapsulation of Complex Creation Logic
- Decoupling of Construction from Use
- Creation of Aggregate Roots
- Ensures Valid States

Encapsulation of Complex Creation Logic means factories ensure that all rules and invariants are satisfied when an object is created. This is particularly useful for aggregates, which may involve the creation of multiple entities or value objects at once

Decoupling of Construction from Use means by using factories, the domain model is not burdened with the details of how objects are instantiated. This promotes separation of concerns

Creation of Aggregate Roots means factories are often used to create aggregate roots. Since aggregate roots govern the lifecycle of aggregates, ensuring that all components of the aggregate are created correctly is crucial

Ensures Valid States means factories help maintain the integrity of the domain by ensuring that objects are created in a valid state

Example :

Consider a scenario where you're working with a **Payment** aggregate. This aggregate might consist of various entities like Payment, Transaction, and Customer Details. Creating a Payment aggregate manually would require setting up several dependent entities, ensuring they all follow the business rules.

Rich model

A **rich model** (also known as **behavioral domain model**) aligns with the key principles of DDD. It emphasizes embedding behavior and data within the same domain object. In a rich model, domain objects not only hold data but also contain the business logic required to manipulate that data

Characteristics :-

- Encapsulation
- Business logic resides inside the domain model
- Better maintainability
- Reduced duplication

Encapsulation means data and behavior are grouped together within domain objects (entities, value objects, aggregates)

Business logic resides inside the domain model means domain objects are responsible for ensuring their own invariants, rules, and constraints

Better maintainability means changes to business rules often involve modifying the domain object itself rather than separate service layers

Reduced duplication means the same domain logic is not scattered across multiple services but is centralized within the domain objects

Anemic model

An **anemic model** is often considered an anti-pattern in DDD. It separates the data (properties) from the behavior (business logic). Domain objects in an anemic model tend to have only getters and setters for their attributes, while the business logic resides outside the domain objects, typically in service layers

Characteristics :-

- Separation of data and behavior
- Violation of encapsulation
- Simpler but less expressive

Separation of data and behavior means domain objects mainly hold data, and business logic is handled by external services

Violation of encapsulation means the domain objects themselves don't enforce rules or invariants; external services are responsible for maintaining consistency

Simpler but less expressive means while easier to initially understand, anemic models often lead to complex service layers with scattered business logic

Example :

```
class Order {
    public $status;
    public $totalAmount;
    public $items = [];
}
```

CQRS (command query responsibility segregation)

CQRS is a design pattern that separates the responsibility of handling commands (actions that change the state of an application) from queries (actions that retrieve data without changing state). It aligns well with DDD principles, especially in complex systems where domain models are rich and perform both read and write operations

In DDD, separating these concerns can help ensure that the command-side focuses on business logic, domain validation, and consistency, while the query-side can optimize for performance and scalability, often allowing for different models or even storage mechanisms for each side.

Command Side: Handles business logic and ensures that any state changes are valid and consistent with the domain rules. It writes data.

Query Side: Handles data retrieval, often optimized for performance and simplicity, focusing on reading data and presenting it.

By implementing CQRS, you can

- Scale read and write sides independently.
- Use different storage models or databases for each side.
- Simplify read models without affecting the complexity of write models.
- Enable eventual consistency when paired with event sourcing