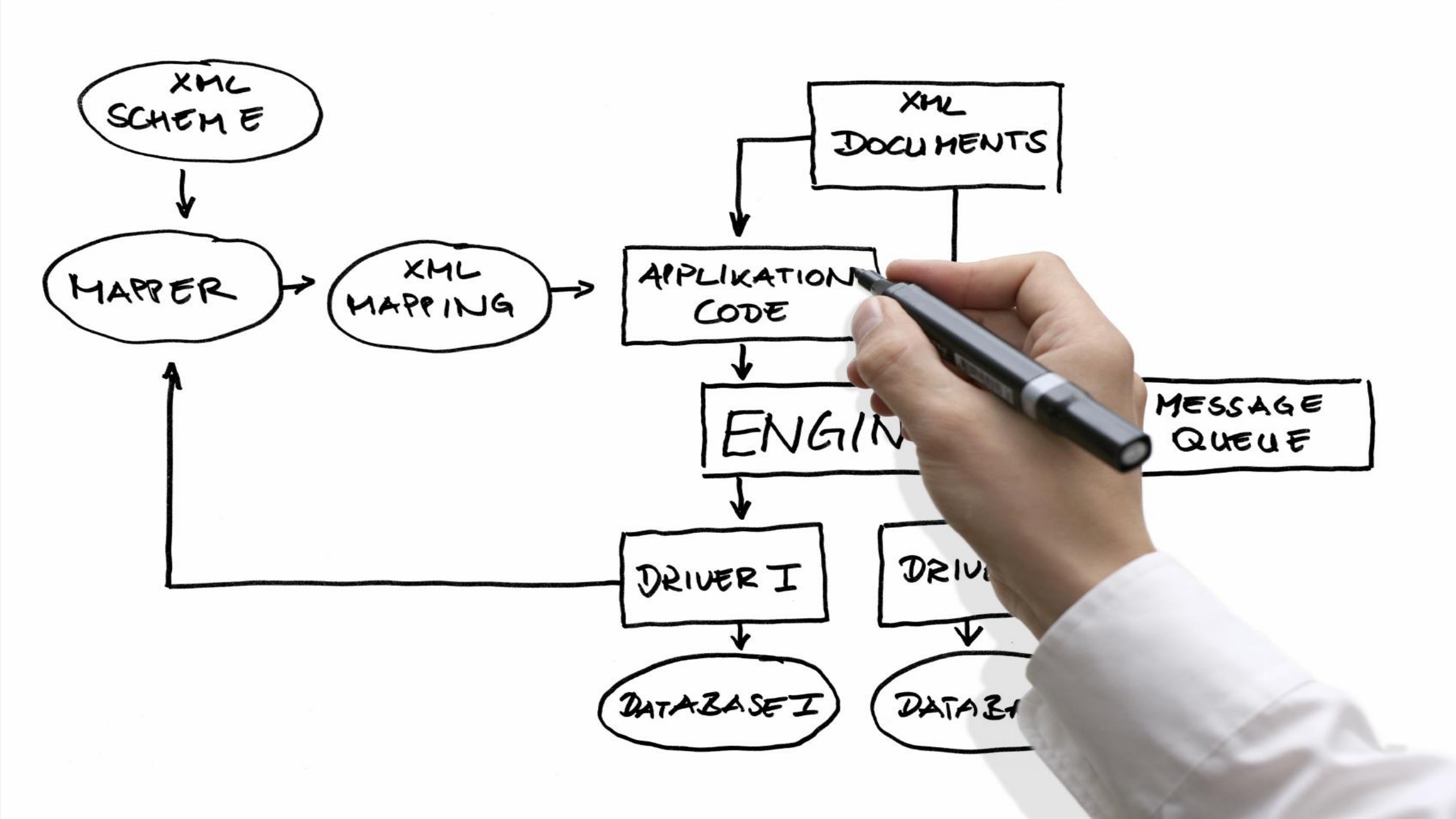
DOMAIN-DRIVEN DESIGN ARCHITECTURE

Pavel Kyurkchiev
PhD. researcher
@pkyurkchiev
https://github.com/pkyurkchiev


XML SCHEME
XML DOCUMENTS
MAPPER
XML MAPPING
APPLIKATION CODE
ENGIN
MESSAGE QUEUE
DRIVER I
DRIV
DATABASE I
DATAB

What is the Domain?

The common dictionary definition of domain is:
“A sphere of knowledge or activity.”

From a software engineer point of view:

Domain refers to the subject area on which the application is intended to apply “sphere of knowledge and activity around which the application logic revolves.”

Definition of DDD:

Domain-driven design (DDD) is an approach to software development for complex needs by connecting the implementation to an evolving model.

Base principles:

- Focus on the core domain and domain logic.
- Base complex designs on models of the domain.
- Constantly collaborate with domain experts, in order to improve the application model and resolve any emerging domain-related issues.

Common terms

Model

- A system of abstractions that describes selected aspects of a domain and can be used to solve problems related to that domain.

Context

- The setting in which a word or statement appears that determines its meaning. Statements about a model can only be understood in a context.

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.

Building blocks:

- Entity
- Value Object
- Domain Event
- Aggregate
- Service
- Repositories
- Factories

Entity

- An object that is identified by its consistent thread of continuity, as opposed to traditional objects, which are defined by their attributes.

Car

Value Object

- An immutable (unchangeable) object that has attributes, but no distinct identity.

Tires

Domain Event

- An object that is used to record a discrete event related to model activity within the system. While all events within the system could be tracked, a domain event is only created for event types which the domain experts care about.

History of motor oil

Aggregate

- A cluster of entities and value objects with defined boundaries around the group. Rather than allowing every single entity or value object to perform all actions on its own, the collective aggregate of items is assigned a singular aggregate root item. Now, external objects no longer have direct access to every individual entity or value object within the aggregate, but instead only have access to the single aggregate root item, and use that to pass along instructions to the group as a whole.

Machine

Service

- Essentially, a service is an operation or form of business logic that doesn’t naturally fit within the realm of objects. In other words, if some functionality must exist, but it cannot be related to an entity or value object, it’s probably a service.

Create a car

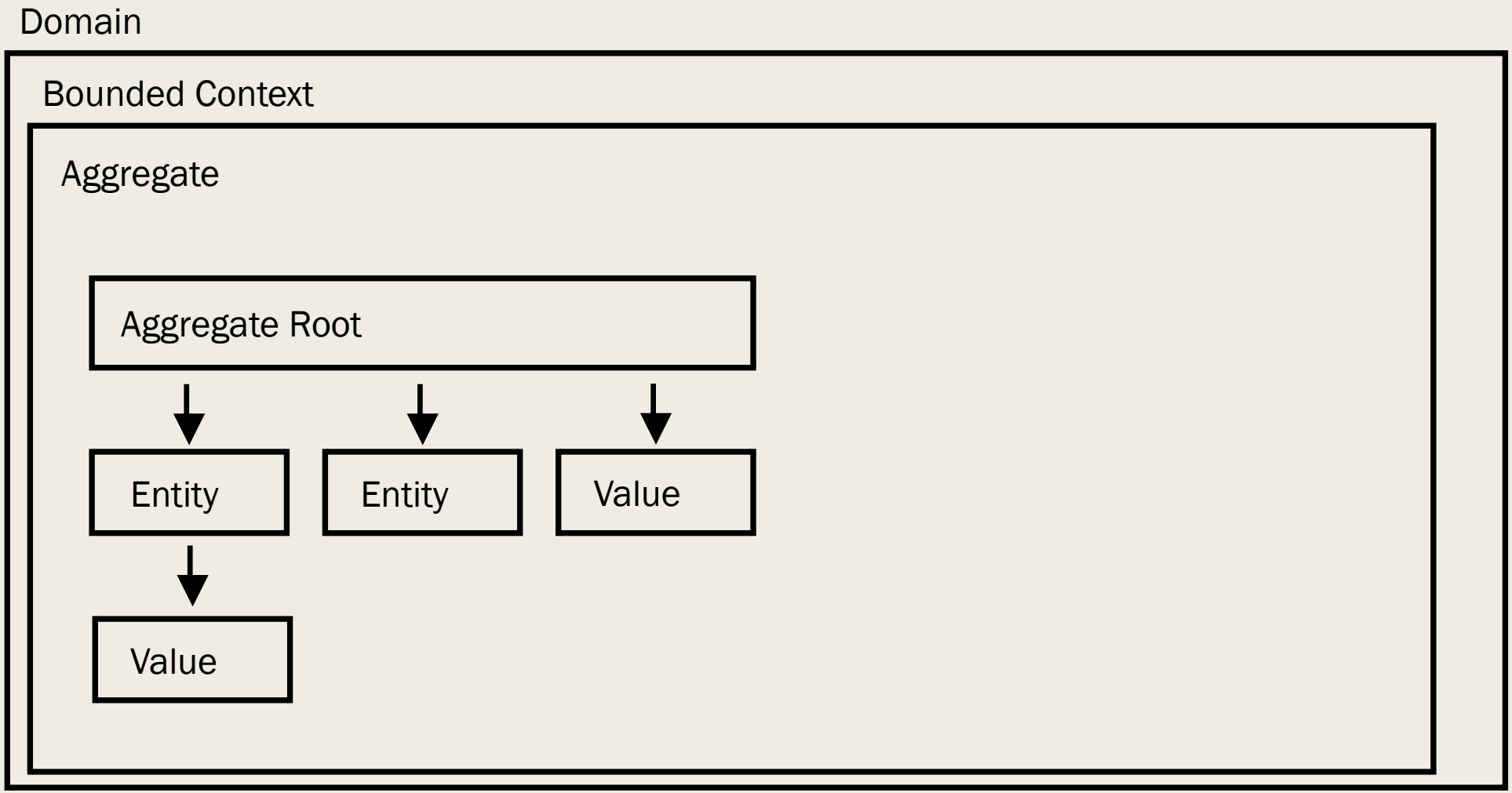
Repositories

- Not be confused with common version control repositories, the DDD meaning of a repository is a service that uses a global interface to provide access to all entities and value objects that are within a particular aggregate collection. Methods should be defined to allow for creation, modification, and deletion of objects within the aggregate. However, by using this repository service to make data queries, the goal is to remove such data query capabilities from within the business logic of object models.

Factories

- DDD suggests the use of a factory, which encapsulates the logic of creating complex objects and aggregates, ensuring that the client has no knowledge of the inner-workings of object manipulation.

Domain


Domain
Bounded Context
Aggregate
Aggregate Root
Entity
Entity
Value
Value

Advantages of DDD:

- Eases Communication
- Improves Flexibility
- Emphasizes Domain Over Interface

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 around the concepts of object-oriented analysis and design, nearly everything within the domain model will be based on an object and will, therefore, be quite modular and encapsulated. This allows for various components, or even the entire system as a whole, 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 an obvious balance is required, the focus on domain means that a DDD approach can produce a product that resonates well with the audience associated with that domain.

Disadvantages of DDD:

- Requires Robust Domain Expertise
- Encourages Iterative Practices
- Ill-Suited for Highly Technical Projects

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 on which the application is intended to apply. In some cases, domain-driven design may require the integration of one or more outside team members who can act as domain experts throughout the development life cycle.

Encourages Iterative Practices

- While many would consider this an advantage, it cannot be denied that DDD practices strongly rely on constant iteration and continuous integration in order 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 largely tied to less-flexible development models, such as the waterfall model or the like.

Ill-Suited for Highly Technical Projects

- While DDD is great for applications where there is a great deal of domain complexity (where business logic is rather complex and convoluted), DDD is not very well-suited for applications that have marginal domain complexity, but conversely have a great deal of technical complexity. Project that is incredibly technically complex may be challenging for domain experts to grasp, causing problems down the line, perhaps when technical requirements or limitations were not fully understood by all members of the team.

QUESTIONS ?