



# Domain Driven Design

THEORY AND HANDS ON ,  
MAHMOUD FADDO

# Course Agenda

- ▶ Why Domain Driven design?
- ▶ Language and context
- ▶ Event storming
- ▶ Designing the model
- ▶ Implementing the model
- ▶ Acting with commands
- ▶ Consistency Boundary
- ▶ Aggregate persistence
- ▶ And much More ..!! (keep you excited )

# second session agenda

- ▶ Why Domain Driven design? (Done)
- ▶ Language and context
- ▶ Event storming
- ▶ Designing the model
- ▶ Implementing the model
- ▶ Acting with commands
- ▶ Consistency Boundary
- ▶ Aggregate persistence
- ▶ CQRS
- ▶ And much More ..!! (keep you excited )

# What is the model

- ▶ models represent some artifact of the real world, but with a narrow purpose.
- ▶ How much space the building will occupy and how high the whole complex will be?
- ▶ Models do not intend to replicate real life. Instead, they represent some particular aspects of real life at a certain level of detail

# What is the model EX:

- ▶ A map of a city's transport lines is a great example:
  - ▶ In any generic map, you can see that it shows the transport links, all stations, and changes. It also indicates some essential geographical aspects, such as relative distance to the sea or on which side of the river the stations are located.
  - ▶ At the same time, such map does not show the distance between stations and has no direct resemblance to real geographical locations. To find exactly where stations are located in the city, or how much time you would need to spend traveling from one station to another

# Models Considerations

- ▶ We will model our solution
- ▶ Models will be our domain models.
- ▶ Martin Fowler Definition : *An object model of the domain that incorporates both behavior and data.*
  - ▶ This definition is quite short and concise, but for all your career you will find domain models as something that we can consider as data models or entity models. Such models visualize entities, data fields with types and relations between entities
  - ▶ such models only express the system state and know nothing of how this state changes
  - ▶ Those models are **anemic-models**

# Why Anemic Models is popular?

- ▶ guidelines to visualize domain models in UML suggest that things that are envisioned are **conceptual classes**.
- ▶ UML models became the only domain models, where domain behavior was considered insignificant.

# What is the problem with anemic models?

- ▶ Anemic models are anti-pattern:
  - ▶ tend to be implemented entirely by database operations
  - ▶ The association between an anemic domain model and its state in the database is so tight that they become siblings and cannot be distinguished from one another.



# When to use anemic model

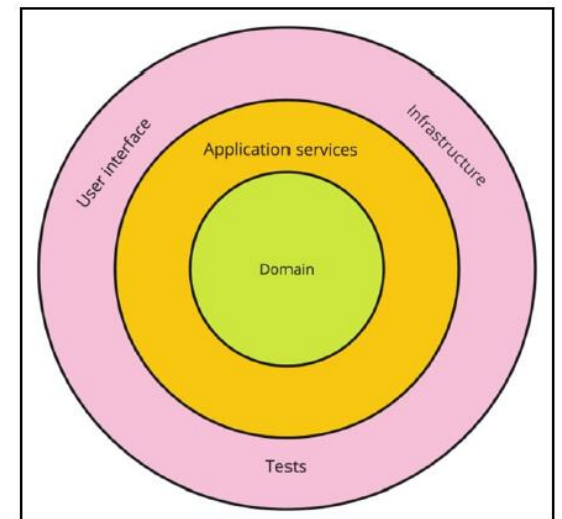
- ▶ where the application has little or no behavior, and such a model serves perfectly fine as a persistence model
- ▶ In this case those models are called data models not domain models.

# Design considerations

- ▶ things such as communication protocols, **user input validation**, and persistence implementation are not seen as part of the domain model. These are technical and infrastructure concerns.
- ▶ **A good rule of thumb here :** the whole domain model should be testable without involving any infrastructure. Primarily, in your domain model tests, you should not use test harnesses and mocks.

# Architecture

- ▶ onion architecture, the hexagonal architecture, and clean architecture principles The center of any application is the **Domain**
- ▶ **Application services** and **Infrastructure** are kept outside and form layers around this core of the system.
- ▶ Unlike a layered architecture, which has dependencies going down from the UI layer to the data layer.



# CQRS

## ▶ CQS: command-query separation

- ▶ states that object methods are separated into two categories
- ▶ Commands: which mutate the system (most often the object) state and return void.
- ▶ Queries : which return part of the system state and do not change the state of the system.

## ▶ CQRS: Command-query responsibility segregation (2007 - 2010)

- ▶ takes this principle outside of an object. It is the same principle but applied to the system level.
- ▶ Separating commands and queries on the system level means that any state transition for The system can be expressed by a **command**, and such a command should be handled efficiently, optimized to perform the state transition. Queries, on the other hand, return data derived from the system state, which means that queries can be executed differently and can be optimized for reading the state

# CQRS (cont.)

- ▶ CQRS makes commands and queries first-class domain objects. Domain events should always be seen as first-class domain objects, but in more advanced CQRS implementation models, domain events play the crucial role of keeping the whole system consistent, and therefore the role of domain events becomes even more critical.
- ▶ we should not only include classes with properties and methods for our domain model, but things like commands, queries, and domain events belong to the model just as much, and

# Design level – Event storming

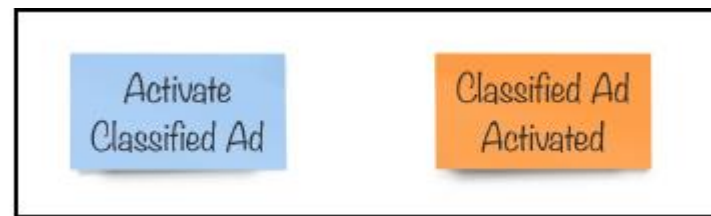
- ▶ Events visualization are not enough ,
  - ▶ We need to know what actions can be performed in the system, and by whom.

# Extended notations

- ▶ we cannot model things such as classes, fields, methods, or functions.
- ▶ we need to use more universal concepts.
- ▶ Commands express the user's intent.
- ▶ Queries represent what users want to see on their screens to take decisions and execute other commands.

# Commands

- ▶ Commands and events are not bound to any language or technology. They also describe the system's behavior very well, using Ubiquitous Language and expressing the intent of our users.
- ▶ Commands express the intent of the users that interact with the system, so applying commands to our system would naturally produce state transitions and result in events being emitted inside our domain model.



- ▶ Note that we have no arrows connecting sticky notes. The flow is determined solely by placing them together in chronological order



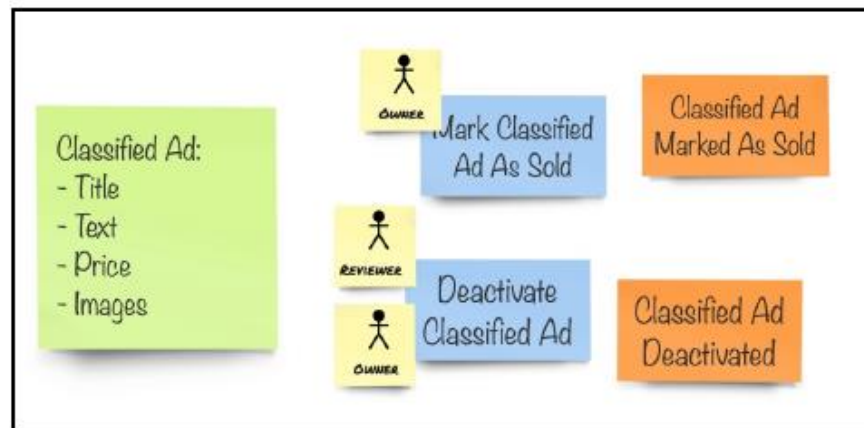
# Read Models (Queries)

- ▶ The read model is something that our users look at before asking the system to do something. It could be any screen in our application, such as a form, a dashboard, or a report.
- ▶ we could assume that our read models are screens in the system, so we can identify what information we need to combine and show to the user.



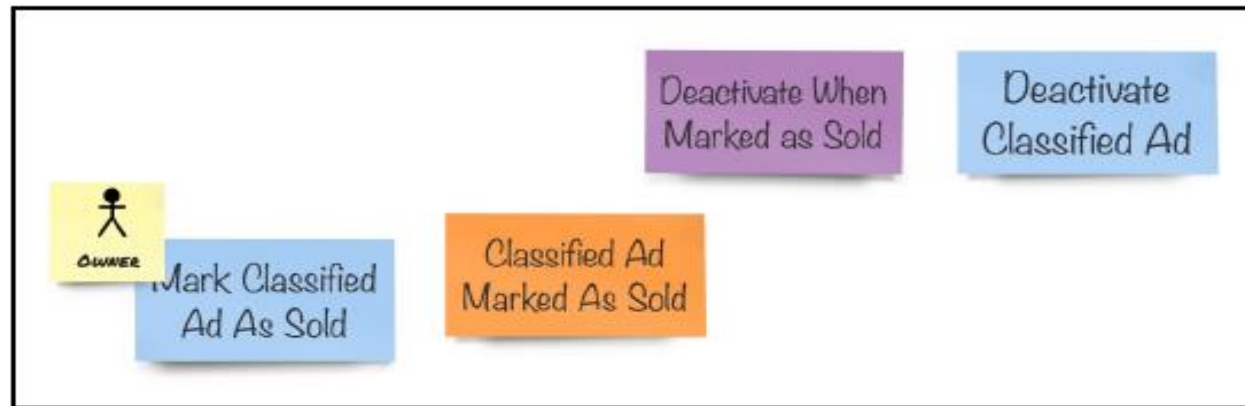
# Users

- ▶ commands in our system are executed by people who use the system—by users. When designing the model, we often need to understand who is running which command.
- ▶ when you are modeling an existing system in which you know specific people and their particular duties, and using their names directly in the model will bring clarity and understanding to everyone involved.



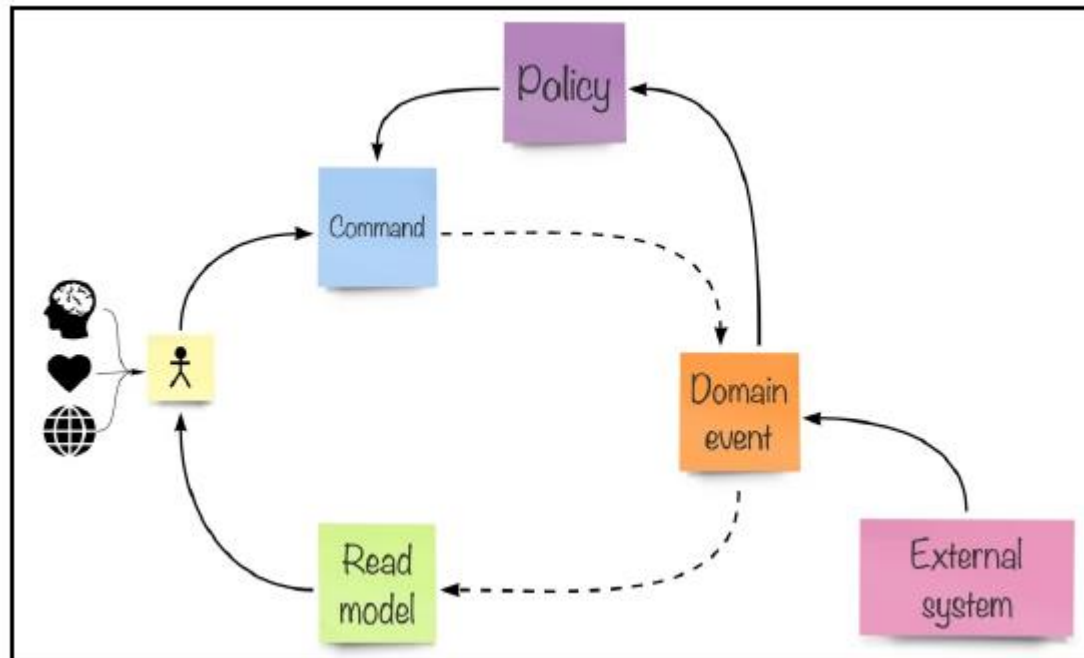
# Polices

- Policies subscribe to domain events, and when a policy receives some domain events it is interested in, it will check the event content and potentially send another command to the system to complement the work. There might be numerous policies reacting to the same event type, doing all kinds of post-processing in an asynchronous fashion, while the user gets control back after the original command has been executed.

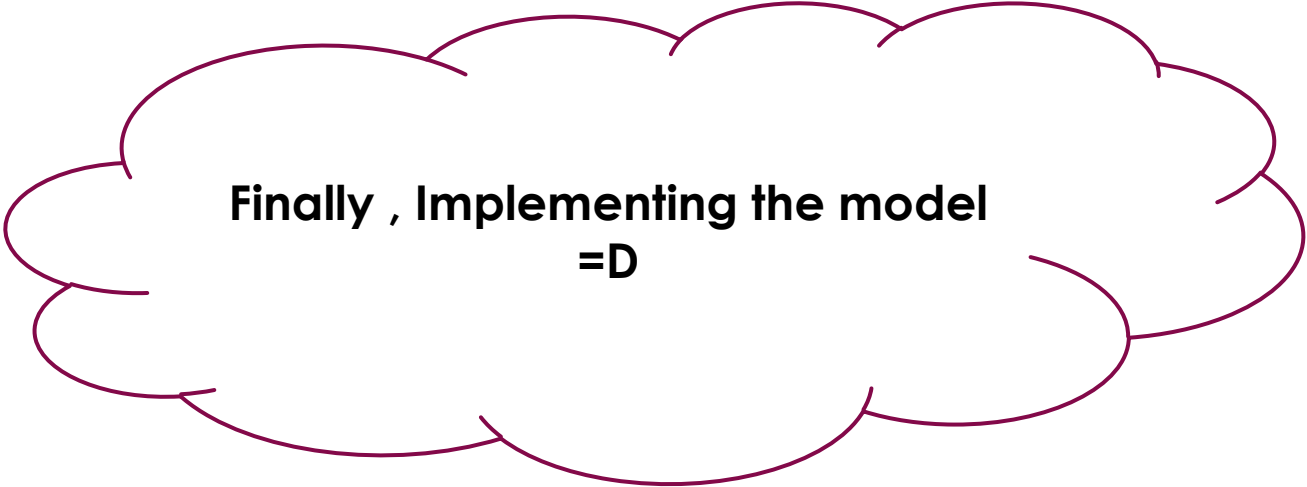


- *when the owner of a classified ad marks it as sold, the system should also deactivate this ad.*

# All together now



# What is next !! Keep excited



**Finally , Implementing the model  
=D**

# Hands on

- ▶ Your turn:
  - ▶ For who interested in increase his skills on event storming Kindly you are requested to make an event storming Design level modeling for Matrix payment cycle.
  - ▶ The Event storming should be done by all of you together.
  - ▶ Use this app to do the Event storming workshop :
    - ▶ <https://miro.com/>
  - ▶ Invite me to workshop after you done all events storming.



# Thank you

SESSION 4