# 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 )

## Implementing the model – Finally =D

- We are going to implement a sample application with learned domain driven design techniques.
- Layers :
  - Domain.
  - FrameWork.
  - Web.
  - Tests.

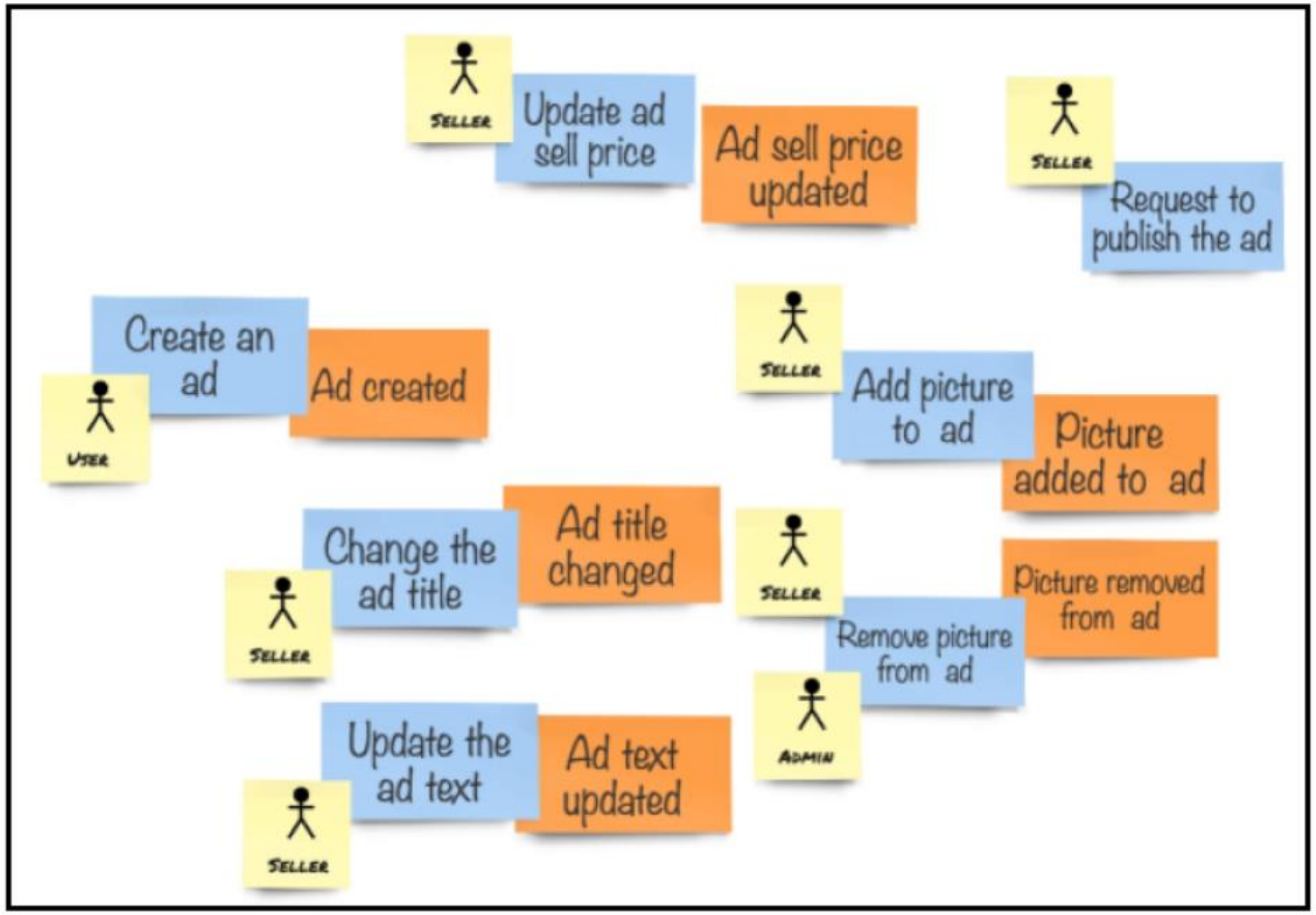
## What is Framework layer

- we will need some components that will allow our domain model to work with things like database, message bus, web server.
- According to DDD these are *Adapters*.
- We will need to have a collection of adapters for all pieces of infrastructure that are being used.
- we definitely will need to build some of the adapters, and although these can be separated in their own libraries, which will connect to our abstractions (hence ports and adapters)

## Transferring our models to code

- we will concentrate on writing things inside the domain project, adding some practical classes and interfaces to the framework projects, and writing tests.
- First, we need to identify which building blocks our implementation will be based on.
- These building blocks are often referenced as **Domain-Driven Design** (**DDD**) ***tactical patterns***
- Ubiquitous Language, Bounded Context, and Context Map are the essential parts of DDD, however some tactical patterns are useful and bring clarity and common language for the implementation.

# Transferring our models to code



## Transferring our models to code

- There is a repetition in all those commands.
- All those commands are being executed on something that is called a **Classified Ad**.
- Mainly, we have an entity here. Entities represent unique objects of the same type. In addition to classified ads, we might expect our system to hold information about sellers and buyers, and those might be entities too

## Identity

- we need to use our own identity. The only important rule, really, is that all entities need to be uniquely identified.
- The most frequently used method to get unique identities today is to use unique database keys. It is because most systems out there are data oriented and designed with persistence-first in mind.
- Such a method has
  - Advantage: IDs are usually numeric and incremental, so it is straightforward to dictate such an ID
  - Disadvantage: the database must be present to get such an identity. (which will make our domain model depends on infrastructure)
- we will use one reliable method and identity Type a **globally unique identifier** (**GUID**), more commonly known as a **universally unique identifier** (**UUID**). Such an ID can be generated using the current time and some information about the computer
- When using GUIDs, we can generate identities for objects before touching any infrastructure

## Let’s code ^_^

- Lets try to build our domain model classified ad .

## Let’s code ^_^

- Rules :
  - all entities need to have an ID, and it must be accessible from outside the entity
  - we shall try to encapsulate as much as we can, and keep our internals safe
  - enforced the rule that our entity can only be created giving a valid set of arguments ,and any created entity of the given type will be, by definition, legitimate

## Let’s code ^_^

- Adding Behavior :
  - Remember, we need to design (and implement) behavior-first.
  - The only reason for us to add those private fields to the entity was actually to support the behavior.
  - each action that is performed in the system amends the system state, and those private fields represent just that—state.
  - encapsulation being enforced, we shall not allow manipulating the entity state by changing property values from outside the entity; this will lead us to the dusty land of CRUD.

## Let’s code ^_^

- Ensure correctness:
  - we were checking the entity constructor parameter to be valid to ensure the newly created entity object is also correct. We applied a constraint, which does not allow creating a new entity without specifying the valid parameter value. By doing this, we are guarding our domain model against getting objects that aren't valid.
  - this type of code needs to be a part of the domain model implementation and not outsourced to external layers, like UI or application service layer.
- (since our domain model is the system core, it takes a few hops for data to move from the user interface to domain objects. It is a valid approach to do a preliminary quality check on the data that tries to enter the domain model before it does. It improves the user experience due to much faster feedback).
- However, the ultimate control is always performed inside the domain model itself, since it shall never come to an invalid state

## Let’s code ^_^

- Constraints for input Values:
  - We need to check the input values like owner id , we can do that using constructor.
- Problem with such method:
  - we are adding more parameters to the entity constructor, and the constructor itself grows since we add more checks for these parameters.
  - we can check the validity of such values, even before reaching the entity constructor, using **value objects**.

## Value object

- Value object pattern is not unique to DDD, but it probably became most popular within the DDD community.
- Value objects are expressiveness and strong encapsulation.
- value objects allow declaring entity properties with explicit types that use Ubiquitous Language.
- can explicitly define how they can be created and what operations can be performed within and between them.

## Value object in our example

- we were taking the ownerId parameter in the entity constructor, and checking it to have a nondefault GUID.
- What we want here is a user ID, since we know that the ad owner is one of our users, because people need to be registered in the system before creating classified ads. It means that we can embrace the type system and make implicit more explicit by using a new type called UserId, instead of using Guid.

## our example is not enough

- value objects aren't just wrapper types around primitive types.
- A classical example of a value object is money.
  - entities are considered equal if their identities are the same. Value objects are different since their equality is establishing by value, hence the pattern name.
  - If we take two €5 banknotes, they represent two different entities, since they are in fact two distinctly different objects, and even have unique numbers printed on them. But for payment, both are entirely identical, since they have the same value of €5.

## Let’s play with Money value object

- What do you think the output of below :
- var firstAmount = new Money(5);
- var secondAmount = new Money(5);
- firstAmount == secondAmount secondAmount

## Let’s play with Money value object

- Offcourse it will fails and that is the most common problem with value object.
- So let’s try to implement Iequitable inteface

## Generic value object

- classes, are reference types.
- It means that those value objects aren't completely following the immutability principle
- will try to do as much as we can to ensure that these objects cannot be changed freely
- We will enforce immutability by creating value abstract class to be used for all value objects

## value object with new C# 9 feature

- We can do the need with one line of code.  =B

# Let’s expand money to meet our domain

- In our domain, classified ads need to have a price, and the price cannot be negative, as our domain expert explained.
- Ensure immutability
  - we must ensure that there are no methods that our value objects expose, which allow changing field values inside these objects
  - If we want to do some operation on a value object instance, it needs to produce a new instance of the same type, but with a new value

## Finally we can make our IDs

- Go to code

## Advanced ways to instantiate value objects and entities

- we have three methods in the entity that expressed its basic behaviour—SetTitle(string), UpdateText(string), and UpdatePrice(double).
- When we work with title we need to make sure some validation before going deep to constructor so we need to make the constructor private and make factories to our oject.
- Factory functions can execute some logic to construct valid instances
- such logic could be different per factory. It is why we would expect to have multiple factory methods in one value object class,
- Factories also help to make implicit things more explicit by using proper naming

## Title value object factory

- Let's see how we can use factories to handle different use cases. Imagine that we get a requirement for ad title to support Markdown partially. In fact, we only need to support italic and bold. We do need to validate the existing factory argument since any string is a valid Markdown string anyway. But, if we can get input from some online editor that can only produce pure HTML, we can do a conversion in a new factory function.

## Money value object factory

- Let’s restrict our money value object with some rules. Go to code.
- Now, let's imagine that our application needs to support different currencies. I mean that currency information would also need to be included in this value object.

## Domain service

- As we our domain model should not depend on external dependencies. we use pattern called **domain service**.
- Because we do not want to depend on anything on the outside of our domain model, we should not put any implementation details inside the domain model. It means that the only thing we are going to have inside the domain project is the domain service interface

## Money after currency lookup

- We give the value object a dependency on the currency lookup domain service. Since we are using the interface, our domain model still has no external dependencies.
- We do not use two as the maximum number of decimal places. Instead, we get this number from the currency lookup, so our value object becomes more flexible.
- For our public methods, we need a simplified constructor, since these methods control that both operands have the same (valid) currency. Because we only trust our internals to use this constructor, it needs to be private. Both Add and Subtract methods use this constructor.

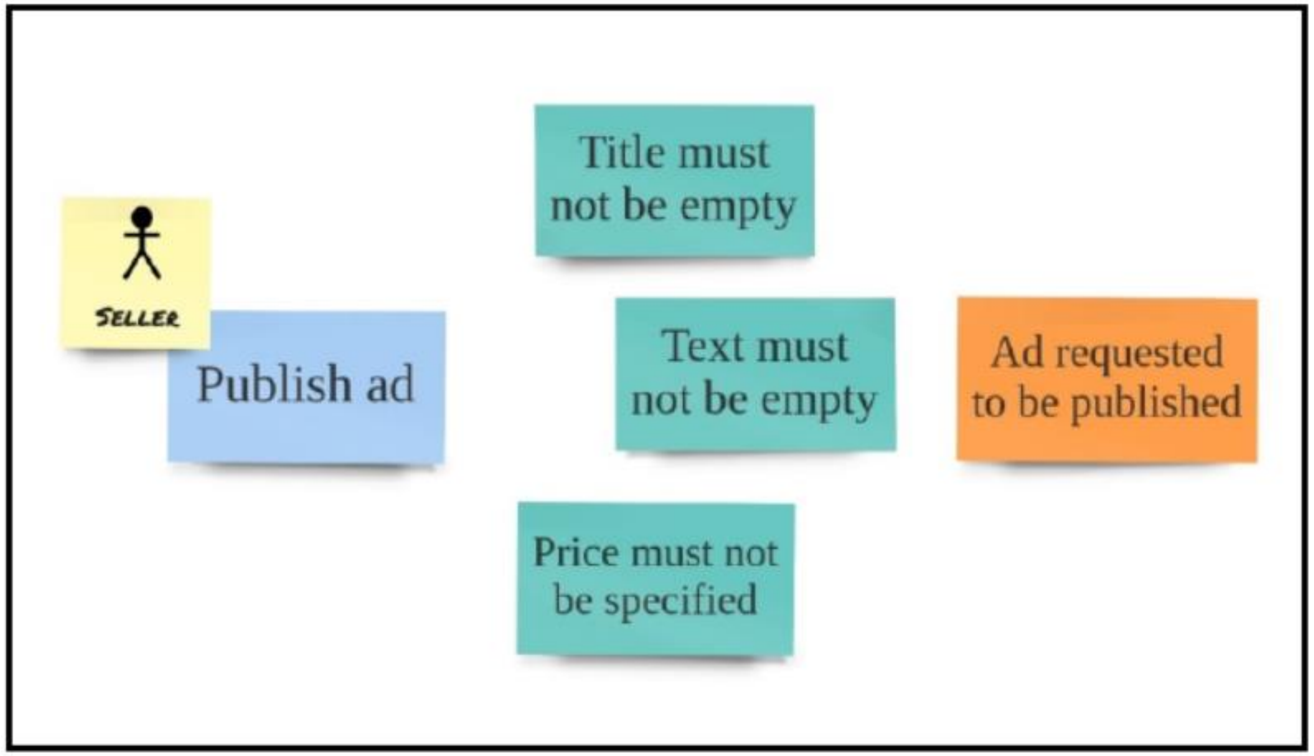
## Domain service and testability

- Our domain model still testable.? Go to code

## Entity Invariant

- We have gone through using value objects to protect invalid values from being even used as parameters for entity constructors and methods. This technique allows moving a lot of checks to value objects, provides nice encapsulation, and enables type safety.
- However , when we create a new entity or execute some behavior using entity methods, we need to do some more checks. Since we can be quite sure that all parameters already contain valid individual values, we need to ensure that a given combination of parameters, current entity state, and execute behavior, is not going to bring the entity to some invalid state.

## Entity Invariant  example



## Where to check

- method of checking constraints before executing the operation, in the operation method itself, has one disadvantage. If we now change the price to zero, it will go through, because UpdatePrice method is not checking the price value.

- To combine rules in one place, we can use techniques of contract programming
- When we execute the operation without doing any additional checks, we will need to do a combined test (post-condition control).

## Enusre testability

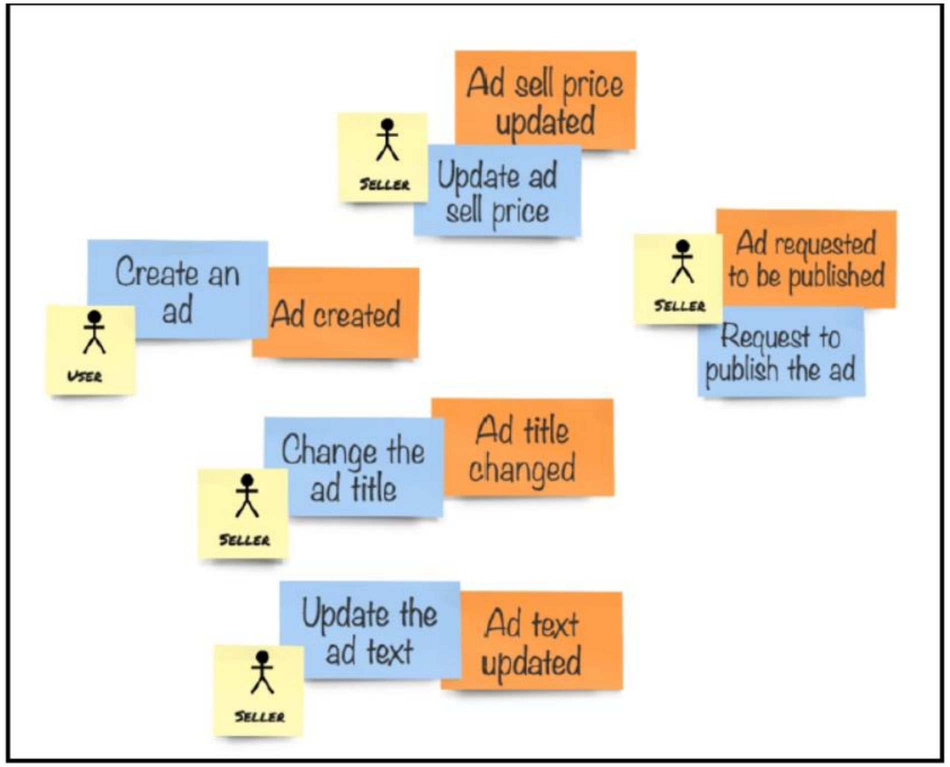
- we converted all private fields to public read-only properties. We need public properties to write tests, although we don't necessarily need to expose the internal entity state

## Domain Events in code

- making domain events first-class citizens in the domain model has excellent benefits:
  - Allowing one part of the system to inform other parts of the system about its state changes.
  - Persisting domain events to get a full history of state changes inside the domain model (Event sourcing)

# Domain Events as object

- Each of those operations changes the state of our entity, and by doing this, raises an imaginary domain event



## Domain Events as object

- Classes that represent events need to clearly describe events (what happened) and contain the necessary information that explains how the system state has changed.
- Usually, events are reactions to executions of commands. Therefore, data in events typically represent data in commands, and maybe some other details from the entity from which the event has been raised.

## Domain Events as object important rules

- We do not use value objects in events.
- **<u>We use noly permitive type.</u>**
- Of course, not using value objects in events mean that some more **complex value objects need** to be flattered.
- each event has the Id property because it makes no sense to raise an event without knowing which entity it comes from.
- the most critical thing about domain events is to represent things that happened, and these things cannot be changed, because we have no time machine or TARDIS to erase or fix the past. Therefore, events should be as simple as possible so we can always load past events, and this should never fail.

## Raising events- Domain Events

- we need to raise events from our methods. To do this, we need some event list inside the entity so we can keep events that are being created

## Raising events -positive

- domain events can be used for integration between different parts of the system. If we publish events to some message bus, and other components in our system subscribe to those messages, they can execute reactive behavior, and make some changes in their domain models, or execute some particular actions, like sending emails, text messages, or real-time notifications.

## Events change state

- events represent the fact of state change.
- It means that an entity state cannot be changed without some interaction with a domain event.
- Go to code

## When method VS value object factory

- entity properties that are still of value object types, use constructors for value objects instead of factory functions. It is because factory functions apply constraints and perform checks while constructing valid value objects. However, domain events represent something that already happened, so there is no point in checking these past facts for validity.
- If they were valid at the time, they should be just let through. Even if the logic in value object has changed, this should never have any effects on applying events with historical data.

## What is next !! Keep excited

**Using our domain model by application service**

## Hands on

- Your turn:
  - ***Pull the repo and try to understand the code base.***

# Thank you

SESSION 5