



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)

Ubiquitous Language

- ▶ What is this language and why to use it ?
 - ▶ As a developer we used to think the only language we need to learn is some programming language , so we can translate human language to programming language and that is it !!.
 - ▶ Two people can understand each other only if they can speak the same language
 - ▶ Not only do they need to speak the same language, but this language must be in one **context**.
 - ▶ Lets look deeper into the importance of language and the context in which these concepts are crucial for successful software projects.

Domain Language

- ▶ Every industry has developed a particular language that only people from that industry fully understand.
- ▶ EX: body shop : this term can be any thing (beauty product , outsourcing company , etc) but when we define the domain the term became more clear.
- ▶ The software industry is, unique because it tends to serve a variety of business problems within any other industry.
- ▶ This means that people from the business will come to developers and try to express their problems using their **language**.
- ▶ When this language **isn't** understood correctly, **issues arise**.

Sample application domain

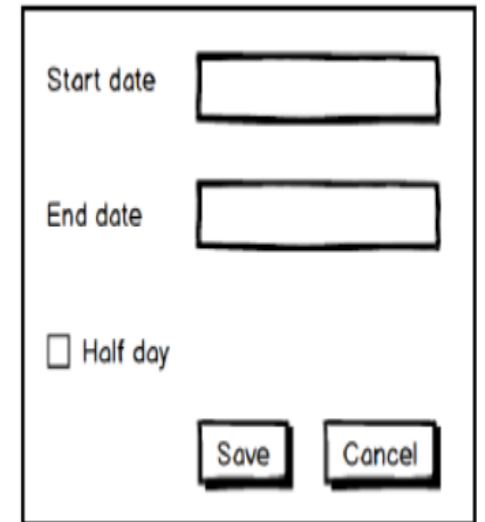
- ▶ Example : selling stuff online for private individuals (OLX)
 - ▶ You can publish a small ad online, and other people might buy things that you no longer need.
 - ▶ We will discuss above example in next few slides
 - ▶ Domain experts state that the ad name is classifiedAd

Making implicit explicit

- ▶ When we start working on a new system, we need to learn a lot.
- ▶ Before we learn about the domain language, we use our understanding, which is often **based on assumptions**.
- ▶ The first and the most obvious risk is that **what you see is all there is (WYSIATI)**
 - ▶ You apply your limited knowledge to past experiences and then get the feeling of understanding.
- ▶ People spend hours arguing about things they thought are different but are the same.
- ▶ People also agree on something that they don't share a common understanding about, and this **never** goes well.
- ▶ To fix this, we need to remove assumptions. We need to make *implicit* things *explicit*.

Making implicit explicit

- ▶ In simple hr-system
- ▶ The **Start date** is ambiguous. It might be the date when sick leave is registered, or it might also be the date when the employee didn't come to work because they got sick.
- ▶ The **End date** is even more ambiguous because it might represent the last day of the sick leave, or the day when the employee came back to work.
- ▶ **The Half day** might apply to both of those fields, but there's no clear indication of what it means.
- ▶ even in such a small form with two fields, one checkbox, and two buttons, many things are **implicit**.
- ▶ If we imagine the **code** behind this form, all those implicit and ambiguous concepts could be found there as well



The image shows a simple web form for sick leave registration. It contains two text input fields labeled 'Start date' and 'End date'. Below these is a checkbox labeled 'Half day'. At the bottom right are two buttons labeled 'Save' and 'Cancel'. The form is enclosed in a black border. The labels 'Start date' and 'End date' are ambiguous as they could refer to different dates in the context of sick leave.

Making implicit explicit

- ▶ In simple hr-system
- ▶ In this form, the fields make much more sense for regular people.
- ▶ Things that were made *implicit* in the first sample are *explicit* here.
- ▶ Everything, from naming individual fields to calls for actions, has a better meaning.
- ▶ So now we can reflect the explicit things on our source code (fields , classes).



First day not being at work

☐ Came to work but had to leave

When the leave is registered

Came back to work at

☐ Came back to work after lunch

Go Back to sample App

- ▶ domain expert, state that they discovered that ads couldn't be published immediately since an ad could contain malicious content. They decided to put in some approval process.
- ▶ So developers will add the new property called Status (enum) to ClassifiedAd class
- ▶ Then developers added update status method which take arguments (status enum)
- ▶ So, after the user clicks **Publish**, the following would happen:
 - ▶ `ad.UpdateStatus(ClassifiedAdStatus.Published);`
- ▶ After the review is complete, the ad would be activated, like so:
 - ▶ `ad.UpdateStatus(ClassifiedAdStatus.Activated);`

Go Back to sample App (Cont.)

- ▶ We miss the intent !!
 - ▶ instead of saying that we want to *publish* the announcement, we *update the status*. Instead of *activating* the ad, we *update the status* again!
- ▶ Even if it all seems to work after some more behavior is added to the system, a complex code start to appear
- ▶ The number of flow control operators is growing, and most of the behavior is now being **driven by status updates**
- ▶ The intent of this update operation is dissolved, and each call needs to be carefully controlled for side effects. The risk of damaging an existing behavior when adding new features is now genuine.

Go Back to sample App (Cont.)

```
public void UpdateStatus(ClassifiedAdStatus newStatus)
{
    if (newStatus == ClassifiedAdStatus.Published
        && (string.IsNullOrEmpty(_title)
            || _price == 0 || string.IsNullOrEmpty(_text)))
        throw new DomainException(
            "Ad can't be activated because some mandatory fields are
            empty");
    if (newStatus == ClassifiedAdStatus.Activated
        && _status == ClassifiedAdStatus.ViolationReported)
        throw new DomainException("Reported ads can't be activated");
    if (newStatus == ClassifiedAdStatus.Deactivated
        && _status != ClassifiedAdStatus.ViolationReported)
        throw new DomainException("Only a reported ad can be deactivated");
    _status = newStatus;
    DomainEvents.Publish(new AdStatusUpdated(newStatus));
}
```

Go Back to sample App (Cont.)

[event handling example]

```
public void Handle(ClassifiedAdStatusUpdated @event)
{
    // controlling the ad visibility based on it's reported status
    if (_status == ClassifiedAdStatus.ViolationReported
        && @event.Status == ClassifiedAdStatus.MaliciousContentDetected)
        CommandDispatcher.Send(
            new UpdateAdVisibility(@event.Id, false));
}
```

Go Back to sample App (Cont.)

Discussions with domain experts also lost some meaning. Instead of using phrases such

as *if the malicious content is detected, we hide the ad and inform our moderation group*, it becomes

and then we query all ads with the status equals MaliciousContentDetected and use the notification service to deliver a message to all users that have moderation rights.

The meaning of the language gets lost behind technical gibberish, mixed with generalized words such as

status and **message**.

Go Back to sample App (Cont.)

- ▶ Refactor using domain language :

```
public class ClassifiedAd
{
    private ClassifiedAdStatus _status;
    public void Publish()
    {
        _status = ClassifiedAdStatus.Published;
        DomainEvents.Publish(new ClassifiedAdPublished(_id));
    }
}
```

Go Back to sample App (Cont.) event handling

- ▶ Refactor using domain language :

```
public void Handle(ClassifiedAdPublished @event) =>
    CommandDispatcher.Send(new ShowClassifiedAd(@event.Id));
public void ReportViolation(User reportedBy, string reason)
{
    _violationReports.Add(reportedBy, reason);
    DomainEvents.Publish(new ViolationReported(reportedBy, reason));
}
public void Handle(ViolationReported @even) =>
    CommandDispatcher.Send(new InformModerators(@event.Id, @event.Reason));
public void Handle(MaliciousAdDetected @event) =>
    CommandDispatcher.Send(new InformModerators(@event.Id, @event.Reason));
```


Build Domain language

- ▶ the domain language can't be built by making a glossary with nouns.
- ▶ A misconception about collecting a lot of nouns in an extensive list and calling it a domain language definitely exists. But this isn't a happy path, and usually, it leads to something called an **anemic model**.
- ▶ Classes in anemic models only have properties, and properties are always named by nouns
- ▶ But a no less important part of every domain is the **behavior**
- ▶ Nouns express what the domain operates with, but verbs describe what's being done.
- ▶ Without verbs, our domain tends to be a set of magic actions when properties change values without any particular reason.

Build Domain language

- ▶ our preceding code clearly expresses the **domain behavior** by introducing verbs as part of the domain language
- ▶ Verbs should be :
 - ▶ precise,
 - ▶ show the intent
 - ▶ describe actions
- ▶ We use them in **imperative** style for activities.
- ▶ publish domain events when describing the history

Link Domain language with previous code

- ▶ In the preceding example, we not only improve our code and brought a better understanding of what it does and what concepts are being present there, but we also discovered some new terms and concepts that our domain model would benefit from. We can start using this terminology when talking to domain experts and see whether they understand it

Result

- ▶ By making implicit things explicit, not only do we discover missing concepts in our code, but we also put them into our domain model. This part is essential because the language is used across the whole range of models—business and mental models, conceptual and visual models, and domain models in diagrams and code. This pattern of using the same concept and, in general, the same language across multiple levels of models in the system is called **Ubiquitous Language**.

Language and context

- ▶ It's important to realize that Ubiquitous Language is only valid within a context. A different context is defined by a different language. There's a misconception that Ubiquitous Language is called *ubiquitous* because it's the single language for the whole business, organization, or domain.
- ▶ It's ubiquitous not horizontally, but vertically. Each context might have its own language, but all layers in this context share one Ubiquitous Language—meetings with the business, models, code, tests, UI concepts, data structures, and so on.

E-Commerce Example :

- ▶ Let's have a look at the classic example of the term **Product** being used in different contexts of the e-comer
- ▶ Although we're operating in the same domain, clearly the term **Product** has a somewhat different meaning in each of the identified contexts:
- ▶ **Sales:** For the salespeople, product means the sales price and maybe the margin. It's where the company earns money, and other properties of the product aren't that important.
- ▶ **Purchasing:** If we buy a product to resell, we're mostly interested in the purchase price, how many items of a given product the supplier has in stock and how fast they can be delivered.
- ▶ **Inventory:** We're mainly interested in how many items we have in stock. If a specific item is out of stock, this context can keep the estimated data when it will be back in stock. Here, we probably also define some internal properties of the product, such as the item number.
- ▶ **Warehouse:** It needs to manage the space that's required to store products, so people in this context need to know when product batches arrive when shipments to customers take place, how products are packaged, and where they are stored.

E-Commerce Example :

- ▶ although we have a popular term, such as **Product**, different departments in the same domain or organization really have very little shared interest and have more indepth knowledge in their subset of properties of something that otherwise could be considered the same object.
- ▶ even for the same domain, there's a different context where language changes and sometimes changes significantly. What happens if we keep the same meaning of words across contexts? Well, things become much less explicit. The degree of ambiguity increases with each new context that we fail to identify and separate. It leads to unclear models and, as a result, to obscure code, where we need to clarify what we mean exactly when we use this and that word.

Context Switching:

- ▶ Mixing different contexts in one working environment also leads to something called **context switching**.
- ▶ many software systems create so-called God classes, such as Customer or Product, which contain all possible properties for all possible views of a physical object. (Lead us to Context Switching)

Context Switching downsides:

- ▶ not all properties need to have value during a specific life cycle of such an object in the system since we have one class for everything, we must assign empty values to all of these properties. Such an approach leads to a high degree of confusion since we hardly understand why these properties are vacant.
- ▶ Inevitably : such classes attract a lot of dependencies , the whole complex system has one large SQL database where tables have many cross-references. The model becomes tangled and very hard to maintain. Sometimes, it gets worse because referential dependencies are sometimes plain wrong. For example, it isn't correct to show an updated product description on some order from the past. The order should contain a snapshot of the purchased product as it was at the moment of purchase

Context Switching avoiding

- ▶ We've identified enough reasons for us to be cautious when it comes to forgetting about context when seeking language.
- ▶ Ubiquitous Language is always
 - ▶ unambiguous,
 - ▶ explicit,
 - ▶ context-specific
- ▶ As soon as you sense or observe that the meanings of words start to change between different parts of the system, this should trigger an alarm in your head that you are probably **crossing context boundaries**.

Context Switching avoiding

- ▶ We've identified enough reasons for us to be cautious when it comes to forgetting about context when seeking language.
- ▶ Ubiquitous Language is always
 - ▶ unambiguous,
 - ▶ explicit,
 - ▶ context-specific
- ▶ As soon as you sense or observe that the meanings of words start to change between different parts of the system, this should trigger an alarm in your head that you are probably **crossing context boundaries**.

Modeling

- ▶ Always remember that good discovering of Ubiquitous will come up with good modeling , good modeling => good software code.
- ▶ Always avoid context switching.
- ▶ Do not ever cross context boundary.

What is next !! Keep excited



Event Storming >3 >3



Thank you

SESSION 2