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)

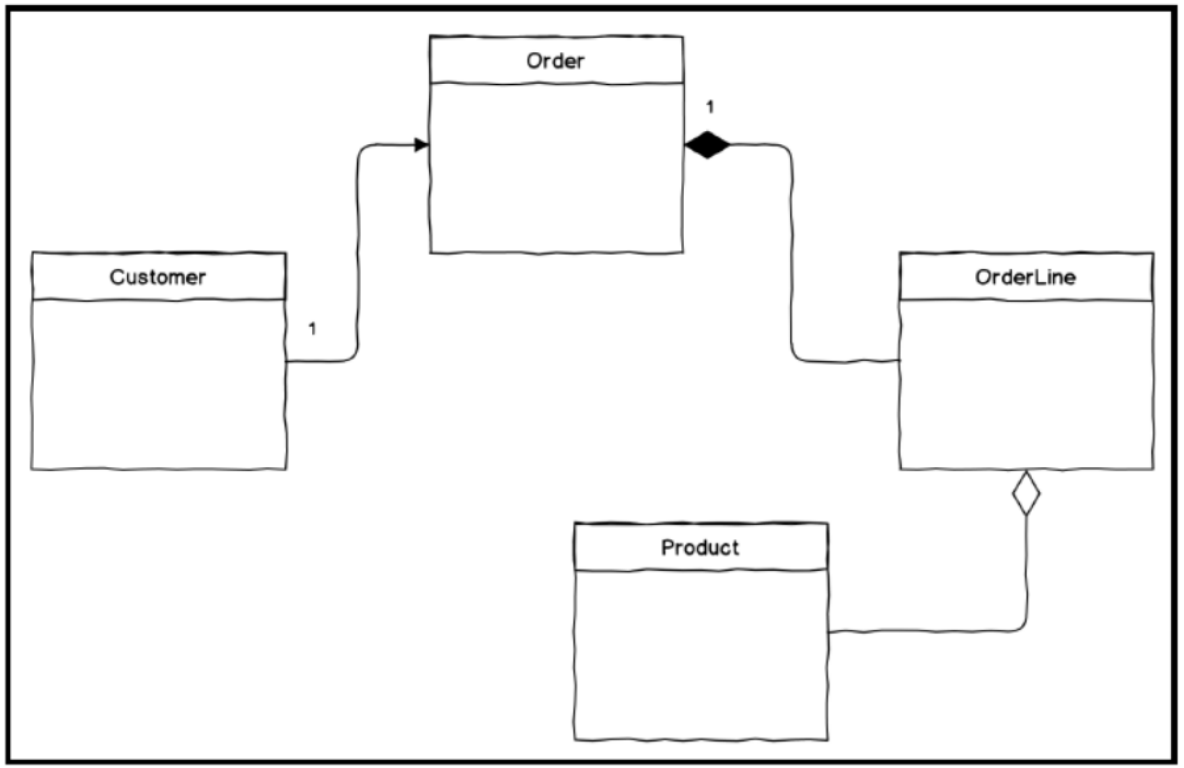
session agenda

- 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)

Aggregate Pattern

- we will concentrate on two aspects of it—**atomicity** and **consistency.**
- An aggregate changes its state as a whole, no matter how the aggregate is persisted.
- If we use an ORM tool and our aggregate spans multiple database tables, all operations on those tables need to be wrapped in a database transaction.
- the consistency aspect requires an aggregate to ensure that the aggregate state is being validated across all operations that are executed on that aggregate.
- **all these validity checks need to be a part of the aggregate code, and therefore they need to be implemented inside the domain model**

Aggregate example



Aggregate example

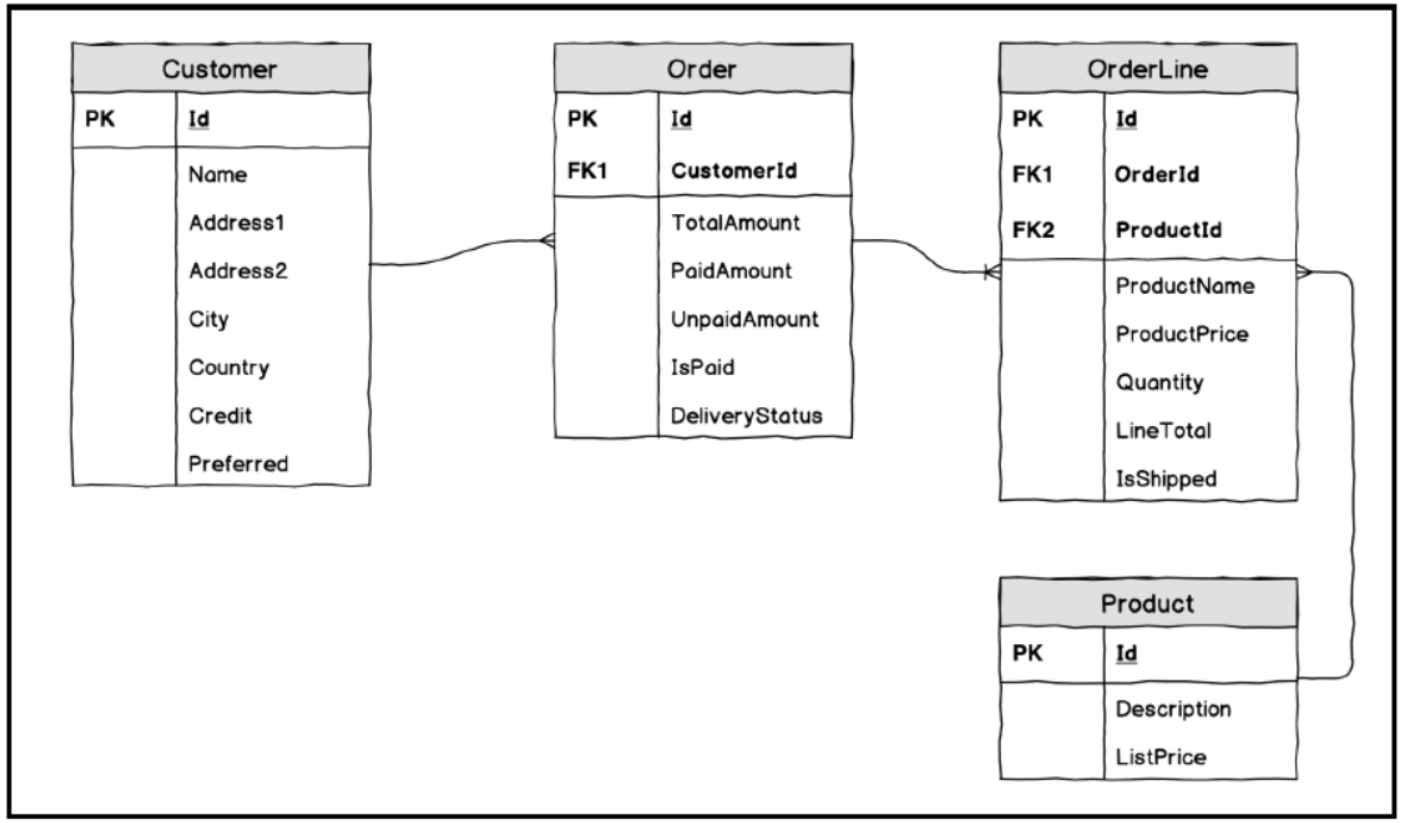
- If we start looking from the data model, we could suspect that Order and OrderLine form some sort of **composition**, since OrderLine records cannot exist without a parent Order.
- If we change the order status because one line of that order is marked as shipped—these changes need to be executed together; otherwise, the order state would become invalid.
- we might get an order with the status *pending* when one order line has already been marked as shipped. So, we would expect such an order to have a *partially delivered* status, and if that isn't the case, then our order state is not valid.

Aggregate example (cont.)

- Since we know that an order line is a child object, we don't really want to expose any operations on order lines directly from outside the domain model.
- itself. In this case, the Order class becomes our aggregate root, and it will have methods that change the state of its lines.
- an aggregate does not guarantee any consistency constraints that are external to the aggregate.
- For the relational data model, it would mean that we cannot have referential integrity between the table that is used to persist our aggregate root (Order) and anything that is outside **the aggregate boundary**.

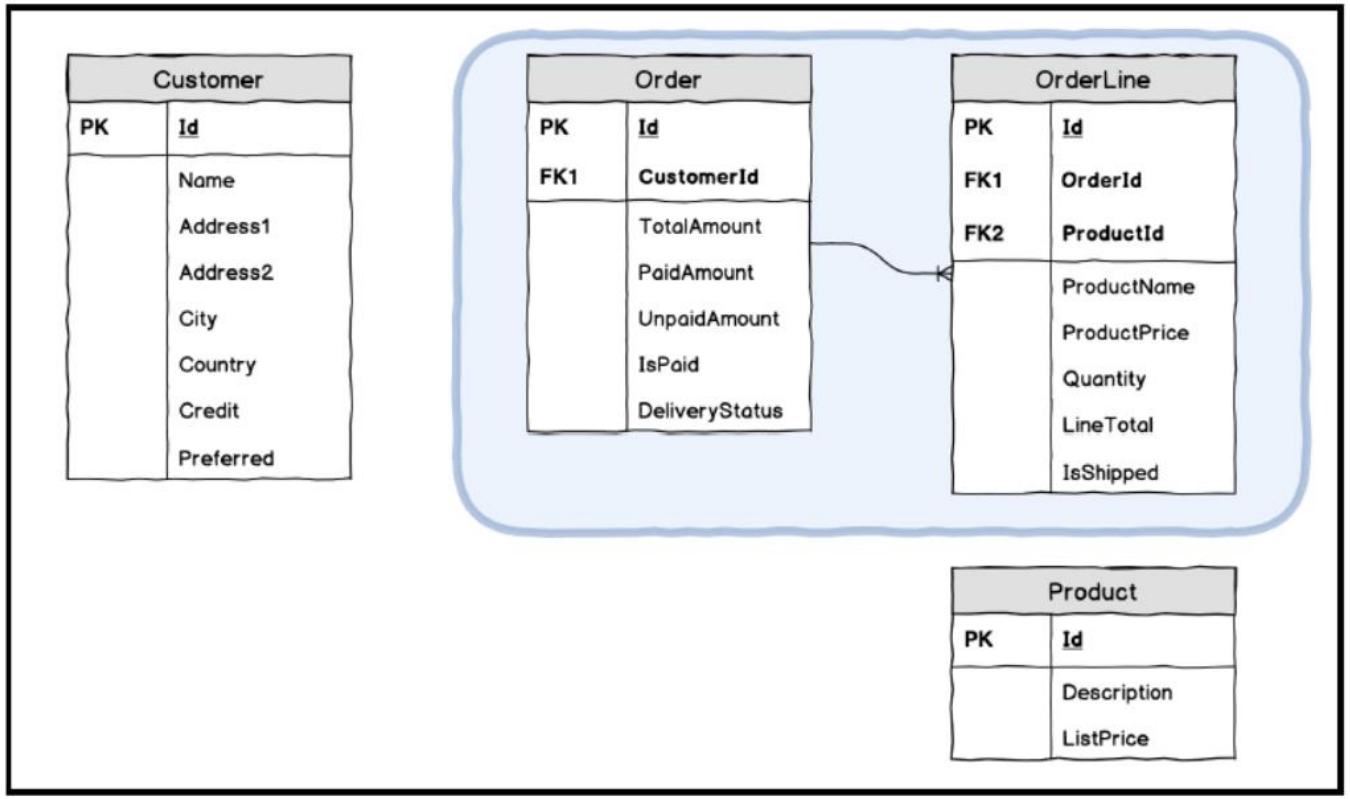
Aggregate example (cont.)

- Traditional Data Model:



Aggregate example (cont.)

- Enhanced with aggregate Data Model:



Aggregate example (cont.)

- Discuss the aggregate Data Model:
  - Note that relations between Order and Customer tables and OrderLine and Product table are now gone, but we kept the reference fields—CustomerId and ProductId.
  - We still need to know whether the customer has placed the order and what products we are selling.
  - However, as regards the normal operation of our system, we don't need object references in ORM, and a lack of referential integrity that some developers might perceive as a negative side-effect of isolating the aggregate, in fact, gives us a new degree of freedom.

Aggregate example (cont.)

- Discuss the aggregate Data Model:
  - Since we do not have object relations between our Customer and Order objects, and we have decided that our aggregate is wrapping all about the order handling, but not Customer, we cannot complete one transaction on these two distinct objects.
  - This might sound like an impossible task, and often, such dilemmas lead to workaround and shortcuts, and then the aggregate pattern is seen as something that gets in the way and needs to be ignored in one or two specific cases.

Aggregate example (cont.)

- Discuss scenario to enhance the model:
  - Checks whether the customer has enough credit to cover what remains unpaid in relation to an order
  - Decreases the customer's credit amount for the unpaid amount of the order
  - Increases the order's paid amount
  - Decreases the order's unpaid amount
  - Sets the order status to *paid* if the unpaid amount is zero (for that code, it will always be true)
  - Increases the customer's total spent amount by the payment amount
  - Upgrades the customer to *preferred* status if this customer has spent over a certain threshold

Aggregate example (cont.)

we need to check what does not belong here:

- updating the total spent amount and upgrading the customer?
- these two actions have no relation to the order processing. Imagine one of these actions failing. Such a failure should have no effect on the order processing.

Aggregate example (cont.)

we need to check what we have to know and do in order to complete the operation:

- we must ensure that the remaining credit limit is higher or equal to the order
- To do so we need to have the information about the available credit limit for the ordering customer.
- But these details are now out of our aggregate scope, so what can we do about it to ensure that an order cannot violate the consistency rule?

Aggregate example (cont.)

- Here comes another aspect of emerging aggregate boundaries, where we need to evaluate the speed of change in objects that our system work with.
- The Customer object now contains some information that forms a customer profile—name, address, and so on. At the same time, it contains some financial details that potentially change for every order that we process.
- Our conclusion should be that the customer profile information changes very rarely, while the remaining customer credit limit changes quite often.

Aggregate example (cont.)

- This means that we are dealing with two different aspects of customer details:

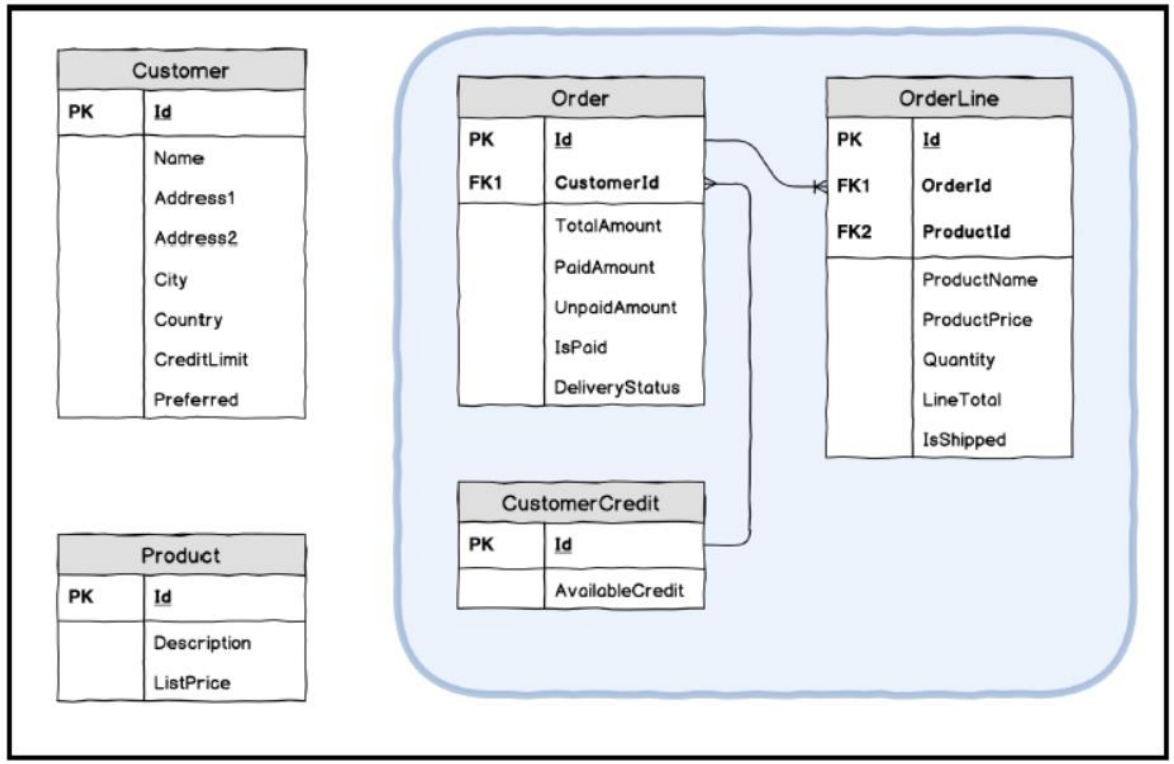
Customer profile	Customer running credit
Name, address, total credit limit	Available credit limit
Changes now and then	Changes for each credit order
No rules for order processing	Required for the consistency of order processing

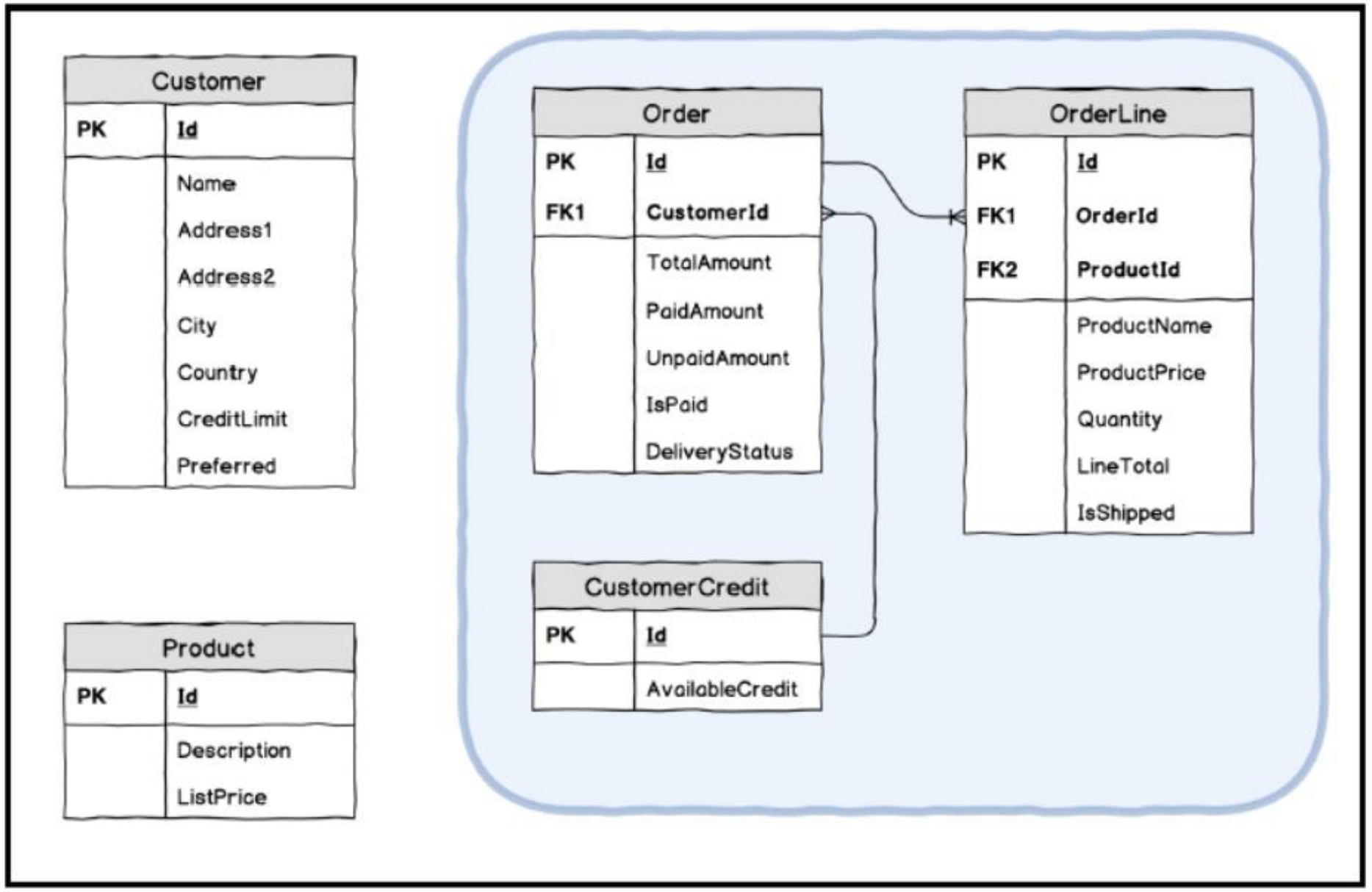
Aggregate example (cont.)

- We finally come to the conclusion that our *holistic* Customer object is not suitable for these different use cases.
- The solution for our model would be to move the information that is required to ensure order processing consistency and that the business rules need to be moved closer to the order processing logic.
- We can do this by splitting our Customer entity into two, each of which is responsible for its own set of use cases

Aggregate example (cont.)

Enhanced Design :




Customer
PK
Id
Name
Address1
Address2
City
Country
CreditLimit
Preferred
Order
PK
Id
FK1
CustomerId
TotalAmount
PaidAmount
UnpaidAmount
IsPaid
DeliveryStatus
OrderLine
PK
Id
FK1
OrderId
FK2
ProductId
ProductName
ProductPrice
Quantity
LineTotal
IsShipped
CustomerCredit
PK
Id
AvailableCredit
Product
PK
Id
Description
ListPrice

Aggregate example (cont.)

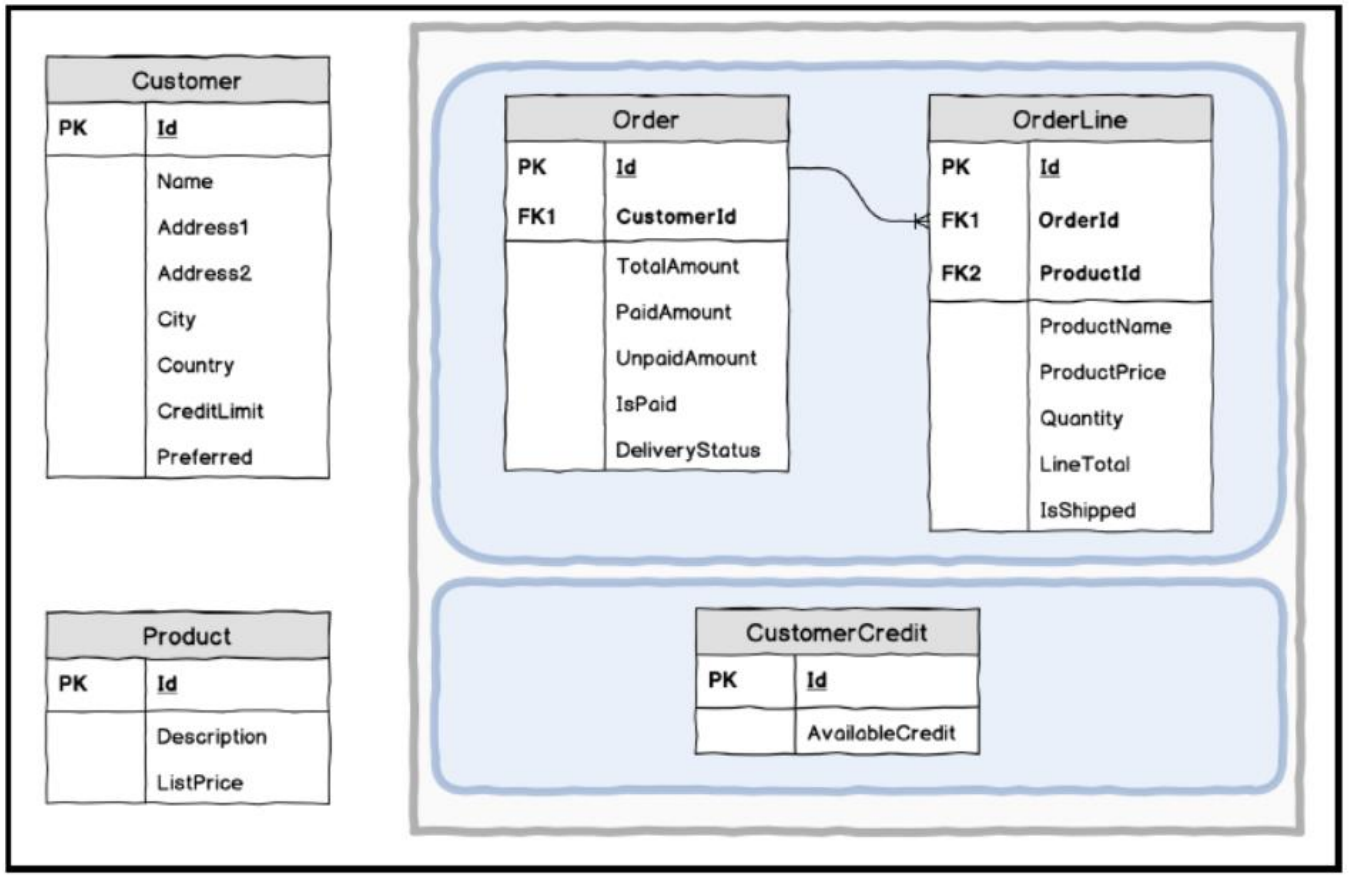
- Our new model looks better, but it has one issue—the aggregation has now shifted to the CustomerCredit entity.
- it looks weird to process all orders by calling methods on the CustomerCredit entity.
- if we remove the CustomerCredit object from the system, we also need to remove all of its orders. This is definitely not what we need.
- In this case, we clearly see the downside of having a larger aggregate with the dubious responsibility of the supportive entity

Aggregate example (cont.)

- The order has its own invariants—a set of unbreakable rules that guarantee the consistency of each order.
  - The sum of PaidAmount and UnpaidAmount should be equal to TotalAmount.
  - DeliveryStatus of an order can only be set to Delivered if, for all order lines of the IsShipped property, this is set to true.
  - The TotalAmount of an order must be equal to the sum of the LineTotal of all the order lines.
  - For each order line, LineTotal must equal the ProductPrice multiplied by Quantity.

Aggregate example (cont.)

- Our final move



Aggregate example (cont.)

- In this model, we have two aggregates inside an isolated part of the system.
- Now, the question arises of how can we enforce our constraints if orders have no information about the available credit for the ordering customer.

Aggregate example (cont.) Domain Service

- We will use the power of domain services to perform this check.
- Inside our domain project, we can define an interface for such a domain service:

```
public interface ICustomerCreditService
{
Task<bool> EnsureEnoughCredit(int customerId, decimal amount);
}
```

- Notice that we have the EnsureEnoughCredit method that returns a Boolean value, instead of returning the available credit limit itself. By doing this, we enforce the utilization of the ubiquitous language and shift the credit limit check logic to the domain service.

Aggregate example (cont.) application Service

- we can use our application service to handle the TakeOnCustomerCredit command
- Go to code.
- This approach moves some domain logic to the application service, and that might not be desirable in some cases.
- To solve this, we could use the double dispatch pattern and let the Order aggregate decide on the constraint.
- Go to code.

Aggregate example (cont.) application Service

- Dependency problem:
- It might look as if we are creating a dependency between our domain model and the infrastructure since it is clear that the domain service needs to fetch the CustomerCredit entity to get the data.
- However, our Order aggregate root only gets the interface dependency.
  - the interface itself is defined inside the domain project =D.

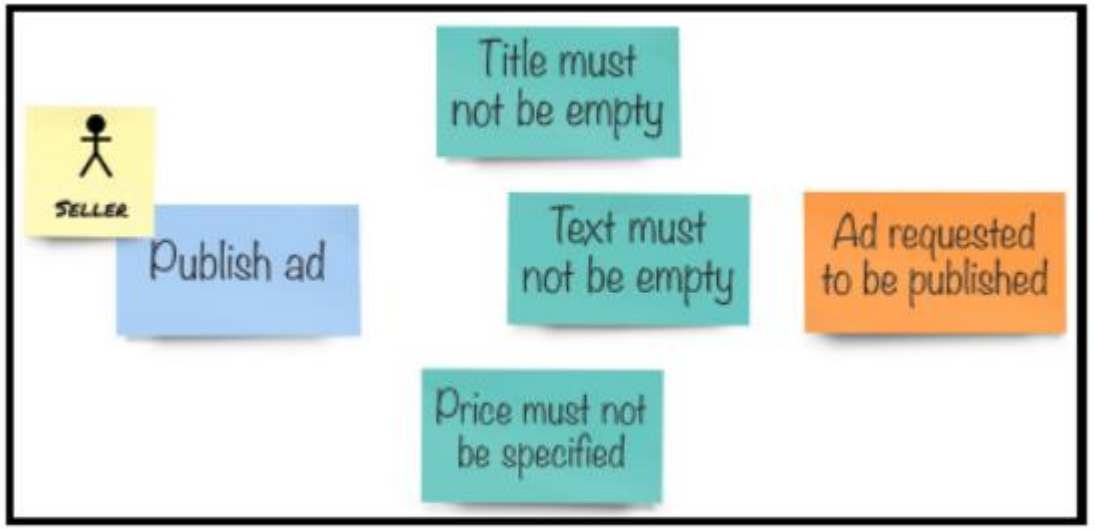
Protecting invariant

- we went through using value objects to protect invalid values from even being used as parameters for entity constructors and methods.
- This technique allows us to move a lot of checks to value objects, provides nice encapsulation, and enables type safety.
- Then, when we create a new entity or execute some behavior using entity methods.
- So what is the invariant?
- we need to execute further 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, the current entity state, and the executed behavior are not going to bring the entity to an invalid state

Protecting invariant

- Protecting the internal state from being invalid.
- bringing the model into an inconsistent state, is one of the most important characteristics of aggregates.
- we need to ensure that we control the aggregate state when calling any command method on the aggregate.

Protecting invariant ex



Protecting invariant ex

- we need to ensure that before an ad can be put to the review queue, it must have a non-empty title, text, and price.
- We cannot put these checks combined with the value object, since before the ad is sent to review, it can have an empty title and text, and it can have no price.
- Only when a given command is being executed do we need to check whether these constraints are satisfied.
- It is what we can call an invariant for this entity.
- We already did that in EnsureValidState function go to mehod.

Enforcing the rule

- how we execute operations on the entity?
  - Call the entity method for operation (the CQS command).
  - The method emits an event.
  - An event is then applied to the entity state to perform the state transition.
- The need to protect its state only applies to the aggregate root entity because it must ensure that the whole aggregate state is correct, and not just its own state validity.
- Therefore, we can make a new base class for this special type of entity.
  - Go to code.

Enforcing the rule

- Discuss the code:
- we have renamed the _events collection to _changes to make the naming more explicit.
- We also added a call to EnsureValidState to the Apply method. This means that whenever we execute an operation that is supposed to change the aggregate root entity state, we apply a new event and the state changes in the When method.
- Before adding the new event to the list of changes, we check whether the new state is valid and that no invariants are broken. If the new state violates the invariants, we throw an exception.

Entities inside an aggregate

- Lets remember :
- when an aggregate is removed, the aggregate root and all its children are also removed from the system.
- None of those child objects should be referenced, accessed, or manipulated outside the aggregate boundary.
- boundary. All operations on an aggregate need to be performed by calling methods on the aggregate root.
- accessing any child objects inside the aggregate needs to go via the aggregate root as well.

Entities inside an aggregate ex

- we need pictures to be added to ads.
- An ad can have multiple pictures.
- *Go to code :*
- we do not expect to hold the image itself inside the entity as a byte array. The physical image itself is not a concern for our domain.
- Within the domain model, we assume that all images are stored somewhere and we just need to have an image location (a URL to an external resource) to be connected to the classified ad.

Entities inside an aggregate ex

- Discuss the value objects:
- we use the power of value objects and ensure the validity of input values inside it, so we don't need to spread this logic everywhere.
- We still need an internal constructor that will allow us to create this object without validating values because we need to be able to unconditionally retrieve the existing object from the database, and we cannot rely on the idea that validation rules will not change over time.
- *Update when on the code*
- *Updating the constructor*

Entities inside an aggregate ex

- This all seems fine, except it isn't =D.
- Our aggregate root performs the logic that belongs to the Picture entity itself.
- The entity needs to be responsible for updating its own state.
- Notice that our Picture class implements the When method from the base class, but it is completely empty. We need to find a way to empower our entities to handle their own events.
- In addition, entities can have their own methods, so the aggregate root doesn't contain the logic that belongs to entities, and instead calls entity methods.
- When we add methods to an entity class, it will produce events to change the entity state.
- But those events can be of interest for the aggregate root as well, so we need to have some code that will traverse events from the entity level to the aggregate root level

Entities inside an aggregate ex

- we need events that are raised on the entity level to be added to the list of changes for the whole aggregate, and this list is maintained by the aggregate root
- All those things require us to change the base classes, and that is what we are going to do now
- *Go to code :*

Entities inside an aggregate ex

- Discuss the code :
- added a constructor to this class that will accept an applier delegate
- Since we always instantiate entities from the aggregate root, we will pass the Apply method of the root to all entities.
- Then, an entity will use double dispatch to inform the aggregate root of events that the entity will be producing
- By doing this, we'll make sure that the aggregate root can also handle events from its child entities.
- We use private methods to implement the new interface, so these methods will not be exposed when we use a class that is inherited from the AggregateRoot or Entity base classes, and this is exactly what we want.

Entities inside an aggregate ex

- Refactor picture entity:
  - change the event handling in the When method
- demonstrate how can we add some logic to the Picture entity:
  - resized and we get new sizes. Out page, cannot be smaller than 800 x 600 pixels
  - *Code*
- We can define the picture size rule directly in each check inside the EnsureValidState method, but it will be quite verbose and not very clear from the language perspective. Instead, let's create a new extension method for the Picture entity.
  - We use an extension method instead of putting this logic inside the entity itself because it is not really a rule for the entity.

What is next !! Keep excited

Hands on

- Your turn:
  - ***Pull the code and understand the code.***

Thank you

SESSION 8