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)

Consistency Boundary

- In monolithic systems, everything seems to be fully consistent. To achieve consistency, a lot of logic is **outsourced** to the database engine and becomes implicit, hard to figure out at a glance, and hard to test.
- Database transactions are frequently used to ensure that multiple state mutations are executed at once.
- If the data becomes inconsistent, that usually means failure, and that requires an extensive investigation to fix the issue.

Aggregate ??!

- Aggregate is the heart of Domain Driven Design.
- **Domain-Driven Design** (**DDD**) means avoiding complex graphs of entities. Instead, developers need to find a minimal logical set of entities that belong together and therefore need to be updated together to ensure consistency. Such a group of entities is called an **aggregate**

Domain model consistency

- When it comes to modeling, we often hear that data models need to be at the center of any system.
- *If you want a good system, you need a good data model.*
- One of great engineer once said : "*I participated in a large project where we started with defining the data model, and, after eighteen months, the project was shut down because the model wasn't complete*."
- If we create a data model first and then try to create our code around it, it is very hard to understand why some rules are being enforced, why those columns in that table are mandatory, and why one table has a many-to-many relationship with another table. These relationships are also *hard to test*, and even if we have tests, we can only run them if we have a properly configured database with a pre-populated set of data, so our tests are also becoming database-oriented.

Domain model consistency (Cont.)

- DDD advocates a different approach.
  - the domain model is essentially detached from the persistence.
  - and it is primarily designed to serve specific business rules.
- So the type of consistency we need to think about in DDD :
- We need to *encapsulate just enough information in our classes so we can ensure that our model keeps being consistent after any state transition.*
- We want to ensure *that our objects cannot violate the rules that are defined by the business, and these rules need to be explicitly defined in code*
- *Important hint:*
- The relational database consistency is outsourced to the database engine.

Transaction boundaries

Do not forget that commands are *Intents :*

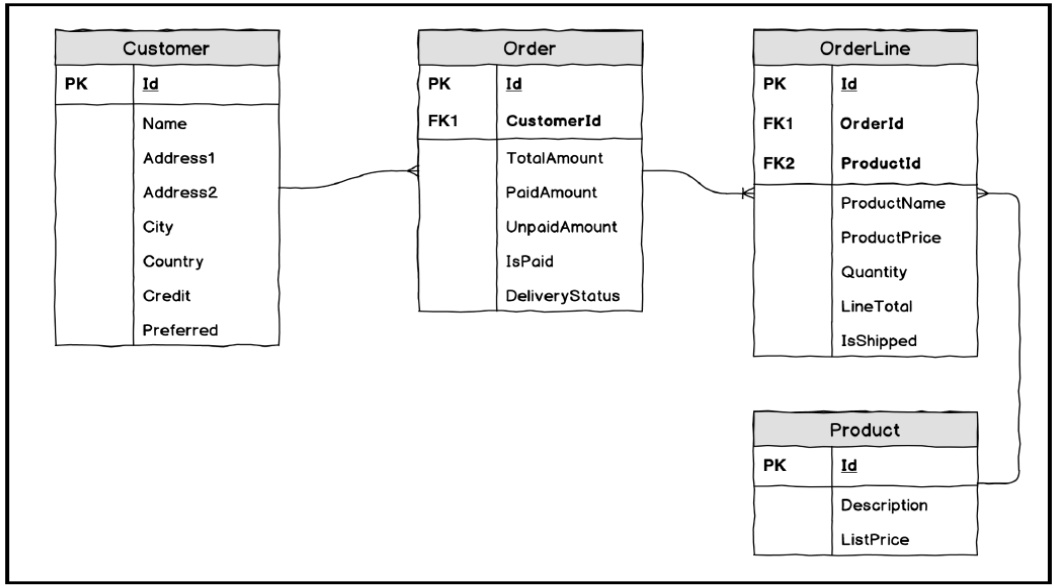
- It could be the case that command comes from another system or even from a timer, but it still expresses some intent.

- Before the command is handled, our domain model finds itself in a valid state. When the command is handled, the domain model should also be in a valid state. This can be a new state if the command handling resulted in an operation being executed, or the same state as before, if the command handling has failed.
- Let's look at the code for handling commands that we created in the previous lecture.

What is the problem with traditional way?!

We will use the e-commerce domain as an example. (open the code)

- So, this code uses the **unit of work** pattern, and DbContext of the entity framework wrapped in the using block implements this pattern perfectly because it accumulates all changes in the database elements and commits those changes all at once when we call context.SaveChangesAsync().



What is the problem with traditional way?!

- Of course, we could expect a lot more tables in the overall model. It could include things such as Product, Supplier, and Shipment.
- These tables all have relations to other tables, and those relations are all one-to-many (or zero-to-many). Our Entity Framework model uses an object reference between Order and Customer.
- We can access the Customer object that is associated with a particular order just by using the order.Customer property and modify any of the Customer properties as we wish.
- It changes properties for both the order and the customer in one logical operation. This operation needs to either complete entirely, or fail

What is the problem with traditional way?!

- Such behavior is typically associated with database transactions.
- A transaction is characterized by four principles, known as **ACID**:
  - Atomicity
  - Consistency
  - Isolation
  - Durability

What is the problem with traditional way?!

- In the preceding code, we can see that the transaction is wrapping the whole operation for paying an order using customer credit.
- This is correct, and what we are dealing with here is a **transactional boundary.**
- For that particular method, TakeOnCustomerCredit, the transactional boundary would include ***two tables—Customer and Order***
- another operation on the same model : (open the code)
- In this unit of work, we have records in Order and OrderLine tables changed in one transaction.

What is the problem with traditional way?!

- *traditional* layered architecture, with no real domain model in place
  - transactional boundaries are being decided by any piece of code that performs changes in the database.
  - The model itself does not enforce any kind of boundary.
  - Two methods, which could even be located in one controller class, **operate on two different transactional boundaries**
  - although the Order table that both methods change will be a part of both transactions.

What is the problem with traditional way?!

- What will happen if processing the remaining order payment on a customer's credit by calling the TakeOnCustomerCredit method could happen in parallel, with one order line being marked as shipped by the ShipOrderLine method?
- Because of the *consistency* part of ACID, one of these methods will fail. It would be very weird for users of this system to know that credit and payment processing is somehow related to shipment and delivery.

Why this conflict happened?!

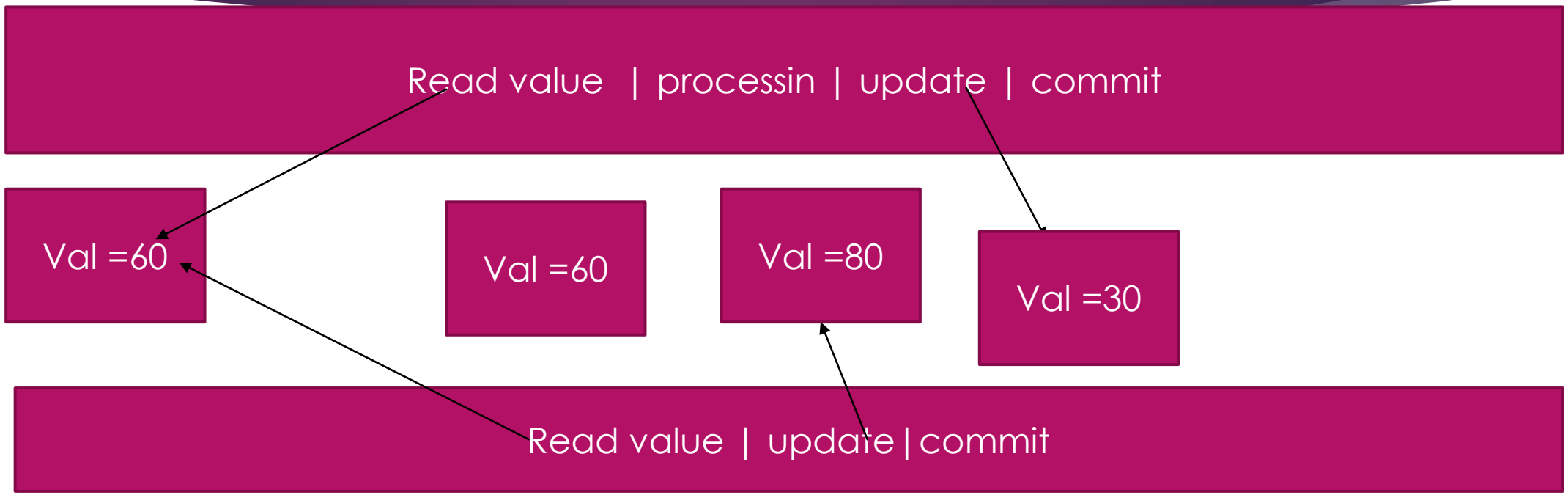
- we see a system with one global data model that closely represents the model of the real world, covering all aspects of the domain that the system implements. It naturally results in large object graphs in code that reflect such a holistic data model.

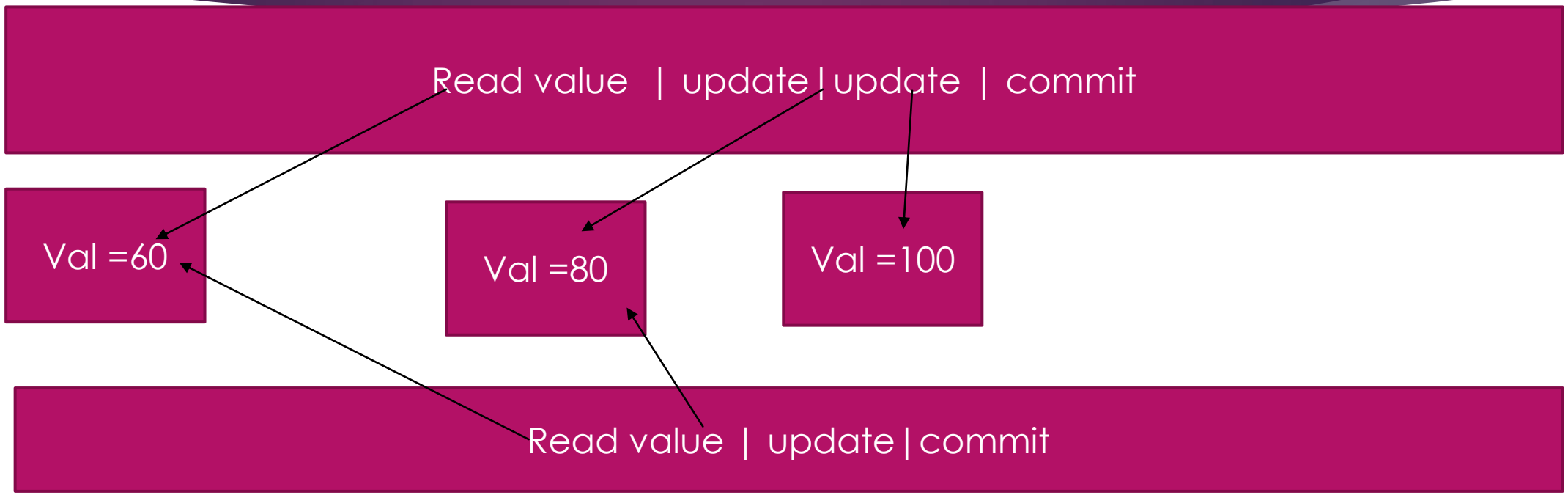
How to resolve it?

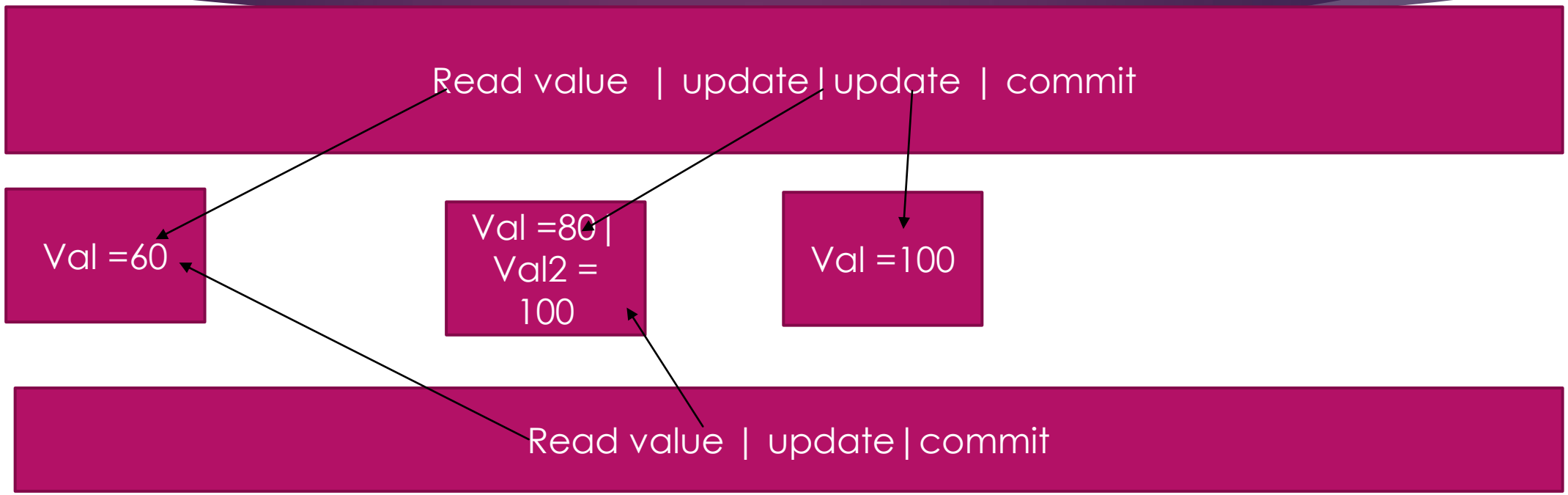
- build our model in a way that we can define transactional boundaries such that different use cases will not conflict with one another when our software needs to execute operations on the same real-world objects that, however, can be represented by different objects in the software model, or even belong to different models.

Let's talk some DataBase

- Concurrency problems
  - **Lost update and dirty write**
  - **Dirty read**
  - **Non-repeatable read or fuzzy read**
  - **Phantom reads**


Read value | processin | update | commit
Val =60
Val =60
Val =80
Val =30
Read value | update | commit

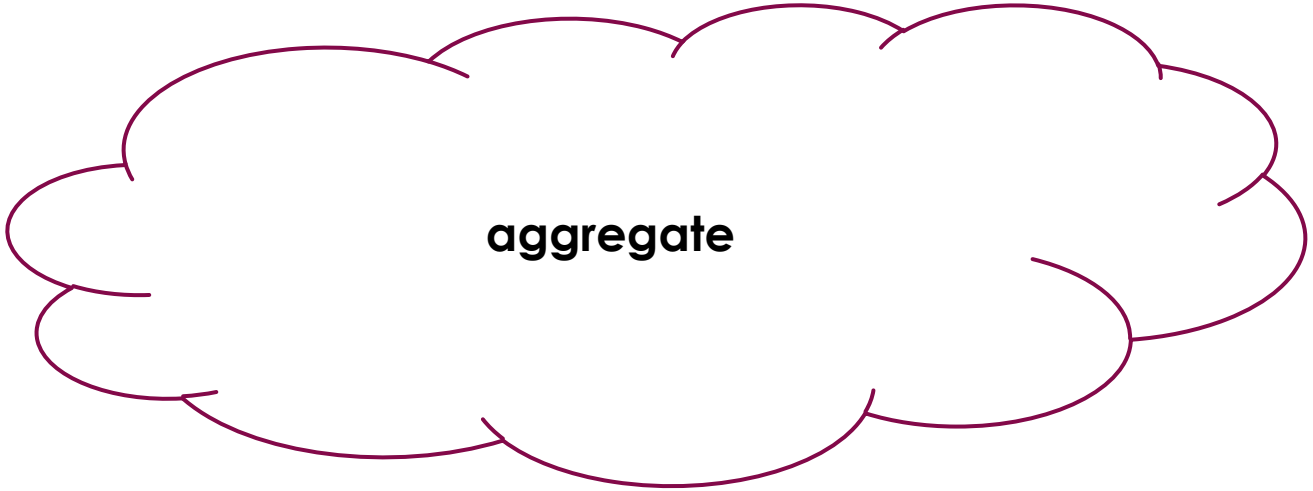

Read value | update | update | commit
Val =60
Val =80
Val =100
Read value | update | commit


Read value | update | update | commit
Val =60
Val =80 | Val2 = 100
Val =100
Read value | update | commit

Select * from table | processing 1 on x rows | select * from table | processing 2 on x+y | commit

Insert into Table y rows
| commit

What is next !! Keep excited



Hands on

- Your turn:
  - ***Do nothing =D.***

Thank you

SESSION 7