
# Wrong and Useless Models built by using Domain-Driven Design

Vadzim Prudnikau

URL: wall.sli.do
CODE: #geecon

HUMAN MADE
NO AI USED

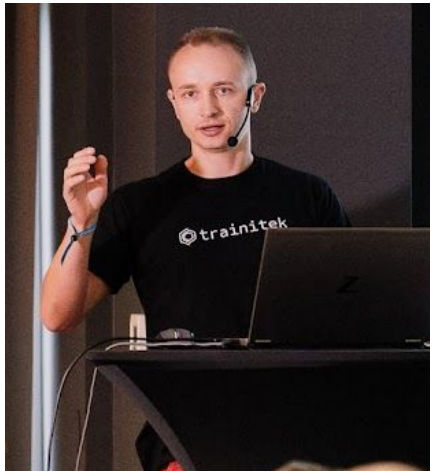
# What you won’t learn today

- How to become a domain modelling expert in 40 minutes :)

trainitek

# Shortly about me

I run training in Architecture, DDD, EventStorming, etc.




**Co-owner and instructor at** trainitek

**Hands-on architect at** APOTEK 1

**Vadzim Prudnikau**

vadim-prudnikov

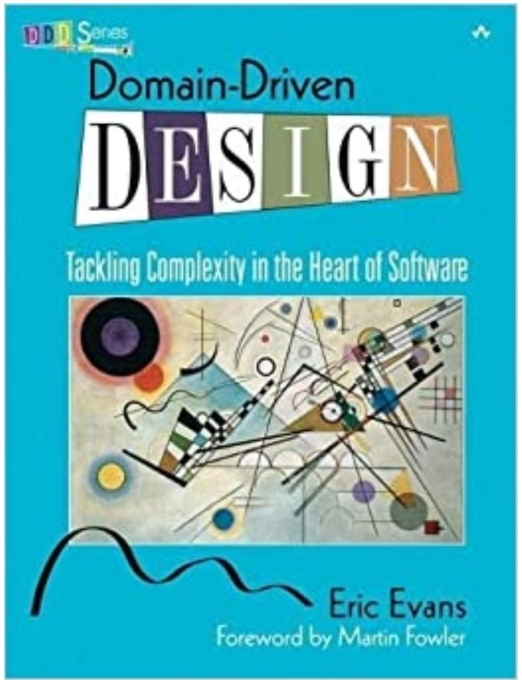
# What will we address today?

- Domains and Models
- A quick intro to Tactical DDD
- Important tips for designing models and writing code
- Model persistence
- Conclusions

trainitek

# Super-fast intro to Domain-Driven Design

- DDD sets focus on solving real business problems (and what doesn't? ;))
- DDD makes communication between people, teams, departments more effective (UL)
- DDD helps distill the core of complex domains
- DDD gives tools and patterns to manage dependencies inside complex systems
- DDD provides some tools on the code level (Tactical DDD)




"**Domain-Driven Design**: Tackling Complexity in the Heart of Software", Eric Evans 2003

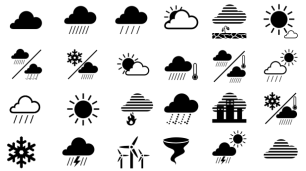
# Domains and Models

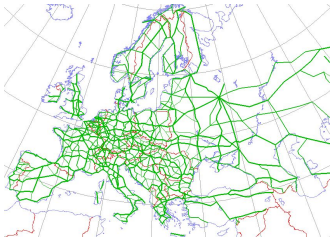
# What is a Domain?

- A **Domain** is a sphere of knowledge, representing the main business of a company
- Domain may refer to an area, sector, or process within the company
- Domain provides the context for the **Problem Space**

# Examples of domains

- Meteorology, atmospheric science, and data related to weather patterns
- Warehouse management, inventory levels, and supply chain processes
- Transportation systems, road networks, and traffic patterns
- Banking, financial transactions, and patterns of fraud





trainitek

# Domains, Problems, and Models

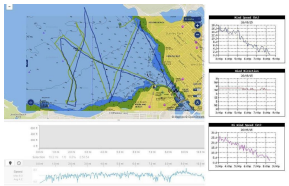
| Example | Domain | Problem | Possible Models - Possible solutions |
|---|---|---|---|
| **Fraud Detection in Banking Transactions** | Banking, financial transactions, patterns of fraud | Detect and prevent fraudulent transactions in real-time | Supervised machine learning, unsupervised machine learning, rule-based systems |
| **Smart Traffic Control System** | Transportation systems, road networks, traffic patterns | Optimize traffic flow and reduce congestion at peak hours | Traffic simulation models, machine learning, real-time data analysis |
| **Inventory Management in a Warehouse** | Warehouse management, inventory levels, supply chain processes | Maintain optimal inventory levels to minimize costs and avoid stockouts | Economic order quantity (EOQ) model, just-in-time (JIT) inventory management, safety stock models |

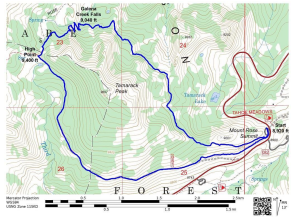
trainitek

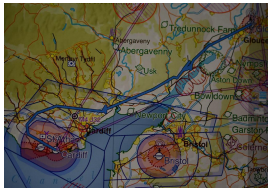
# What model to use?

- **Domain:** Cartography, Geospatial Data, Transportation Networks
- **Problem:** Route Planning and Navigation: Assist users in efficiently moving from one place to another (A -> B)
- **Possible Models:**
  - Google Maps, OpenStreetMap (in cities or between cities)
  - Marine Navigation Systems (e.g., Navionics, iNavX)
  - Mountain Navigation Systems (e.g., Gaia GPS, AllTrails)
  - Air Navigation Systems (e.g., ForeFlight, SkyDemon)
  - Classic physical Maps

Google Maps







# Models

- A **Model** is a **simplified** representation of a domain, focusing on essential elements to **address specific problems**

- **All models are "wrong"** as they cannot capture every aspect of reality, yet some are useful for problem-solving

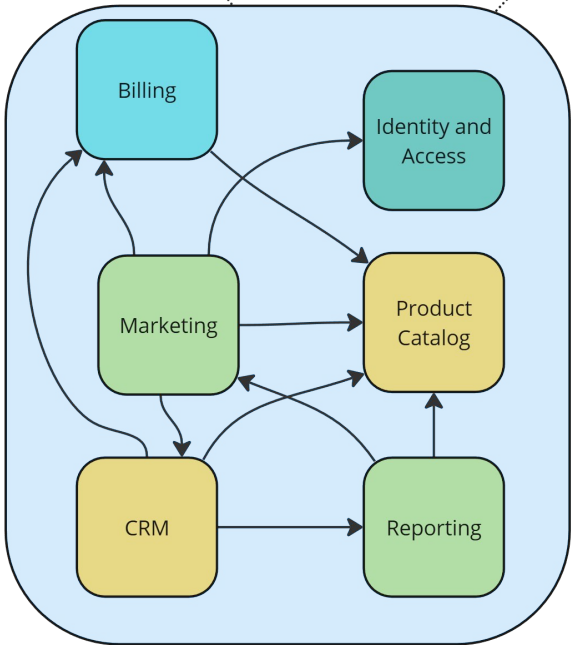
- **Models live in the solution space**, representing potential approaches to tackle challenges within a problem context

trainitek

# Risky models

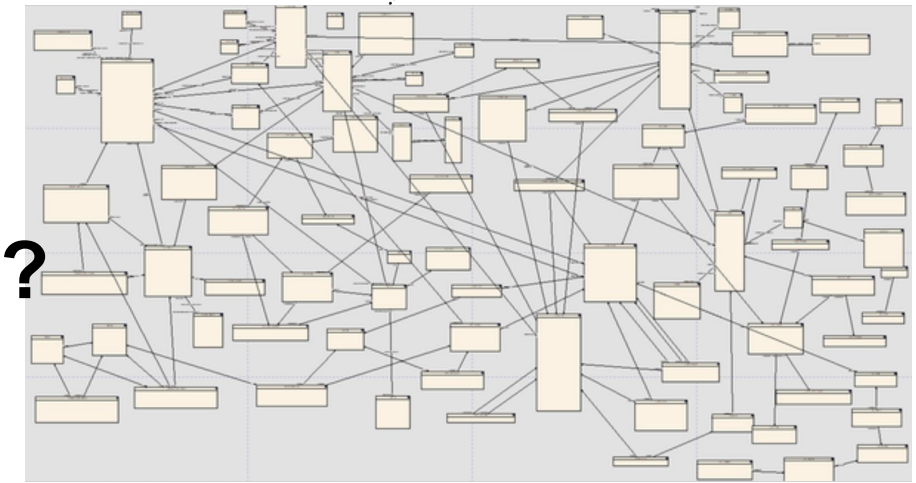
Usually a Monolith or a Distributed Monolith

System with different capabilities




= ???

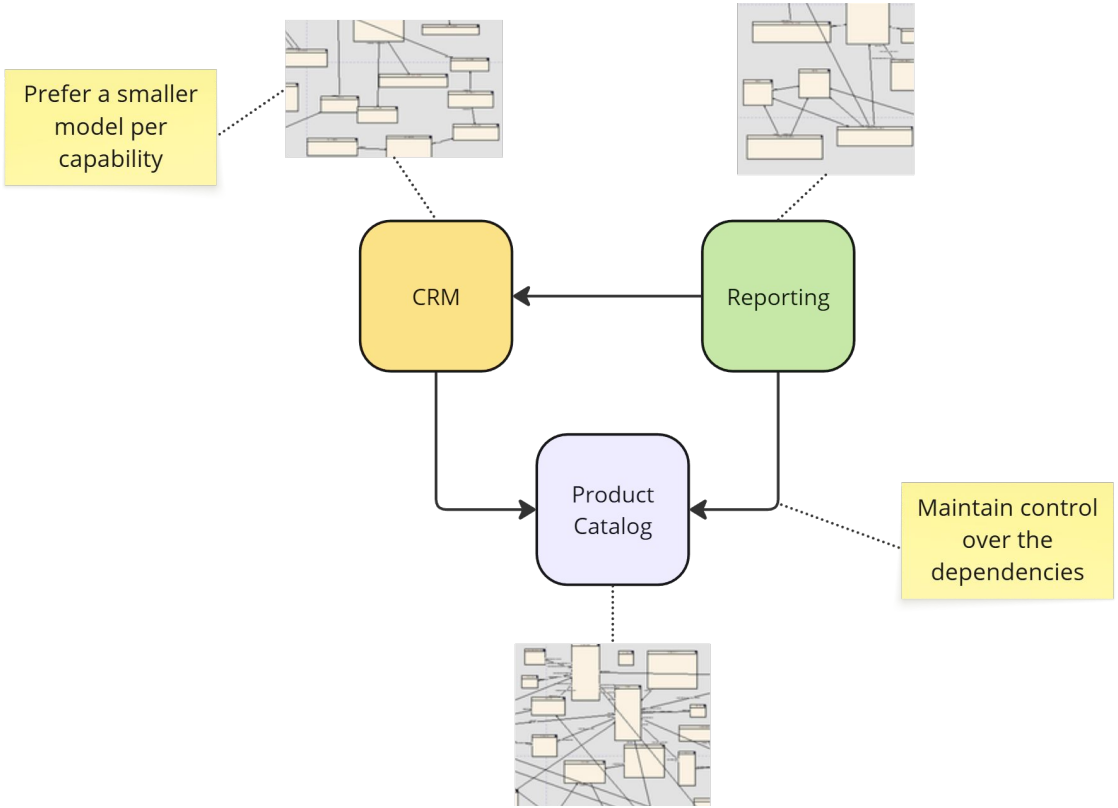
This is what it looks like when the technical model covers all the capabilities in one go



Source: https://hackolade.com/img/edm.png

trainitek

# Better models



# Should we always build models ourselves?

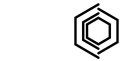
- No. Sometimes, it’s easier to buy off-the-shelf.

# A golden path of applying DDD

- Strategic DDD (Ubiquitous Language, Bounded Contexts, Context Maps)
- Tactical DDD
- Not other way around! :)

# Is there any good starting point?

- Some tools, like EventStorming or Event Modeling, can help you achieve the goal.




# A quick intro to the essentials of Tactical DDD

# Value Objects

- They are defined by the **values of their properties**
- Can contain business logic
- Can be unit tested
- Should be immutable:
  - Final / Sealed class,
  - With constructor with all required fields,
  - No setters.

# Value Objects: an example

```java
public final class Price {

    private BigDecimal amount;
    private Currency currency;

    private Price(@NonNull BigDecimal amount, @NonNull Currency currency) {  1 usage
        checkArgument( expression: amount.compareTo(BigDecimal.ZERO) >= 0,
                "Amount can't be negative but it was (%s)".formatted(amount));
        this.amount = amount.setScale( newScale: 2, RoundingMode.HALF_UP);
        this.currency = currency;
    }

    public static Price of(@NonNull BigDecimal amount, @NonNull Currency currency) {
        return new Price(amount, currency);
    }
```

trainitek

# Value Objects: an example

```
class PriceSpec extends Specification {

    def "Price can be created with the properly scaled amount"() {
        expect:
        new Price(input as BigDecimal, NOK).getAmount() == expectedAmount
        Price.of(input as BigDecimal, NOK).getAmount() == expectedAmount

        where:
        input || expectedAmount
        0     || 0.00
        0.0   || 0.00
        0.00  || 0.00
        1.78  || 1.78
        1.155 || 1.16
        2.789 || 2.79
        2.784 || 2.78
    }

    def "Price cannot be created with negative amount"() {
        when:
        Price.of(input, NOK)
```

# Entities

- Have a life cycle
- Contain business logic
- Have identity not depending on their properties
- Are mutable
- Can be unit tested

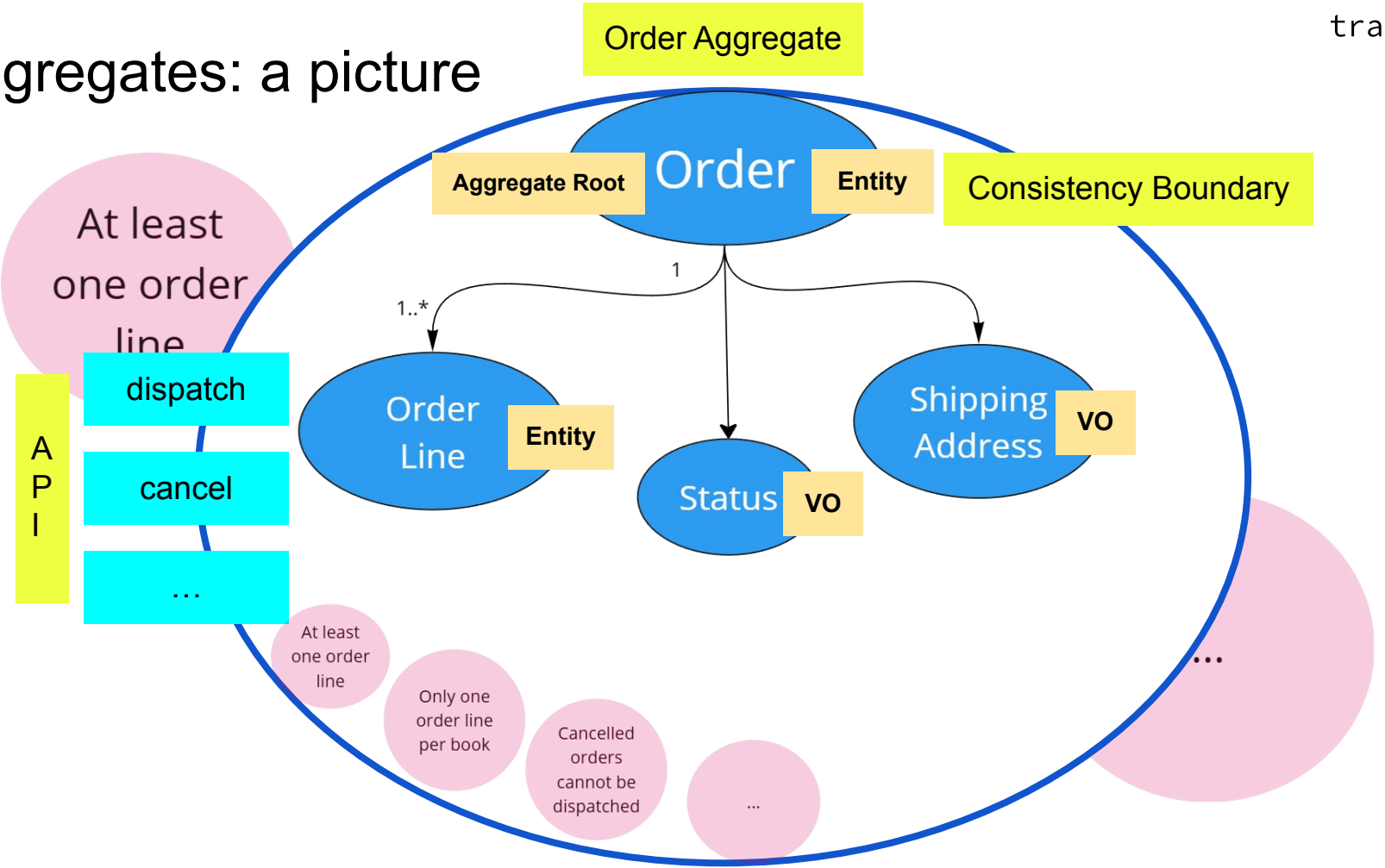
trainitek

# Entities: an example

# Aggregates

- **Graph of objects defending business rules** (invariants)
- Are treated as a single unit
- Are accessed through an entity - aggregate root
- Contain entities and value objects
- Can be unit tested
- Some like to think about them as a consistency boundary.

# Aggregates: a picture



trainitek

# Business logic exists inside aggregates

```
class DispatchOrderHandler {
…
public void handle(DispatchOrder command) {
  var order = …
  order.dispatch();
} order.getOrderLines().forEach(ol -> ol.setStatus(DISPATCHED));
  order.setDispatchedAt(clock);
  …
  if (...) {
  }
}
```

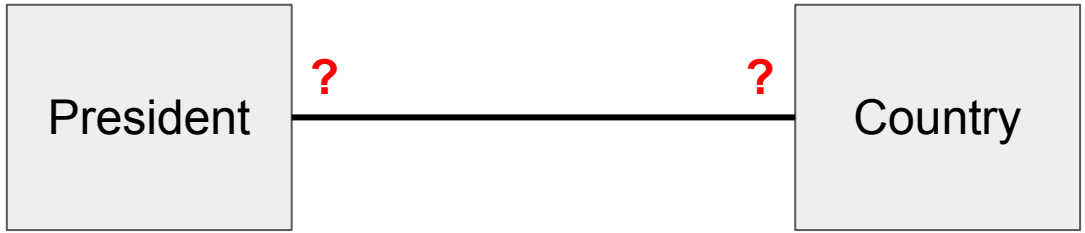
# Are there more building blocks?

- Yes: Domain Services, Specifications, Repositories, and others

# Important tips for designing models

# Tip #1: Don’t mirror the reality

- You shouldn’t mirror the reality in code as the reality is too complex.
- Focus on the problem you need to solve!

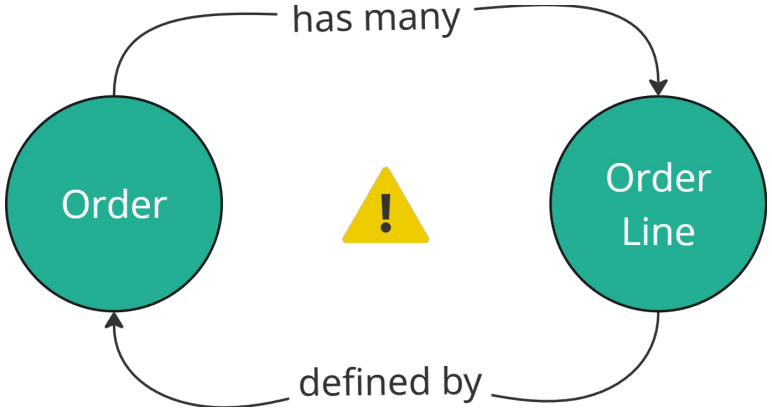


trainitek

# Tip #2: It’s OK to use CRUD with DDD

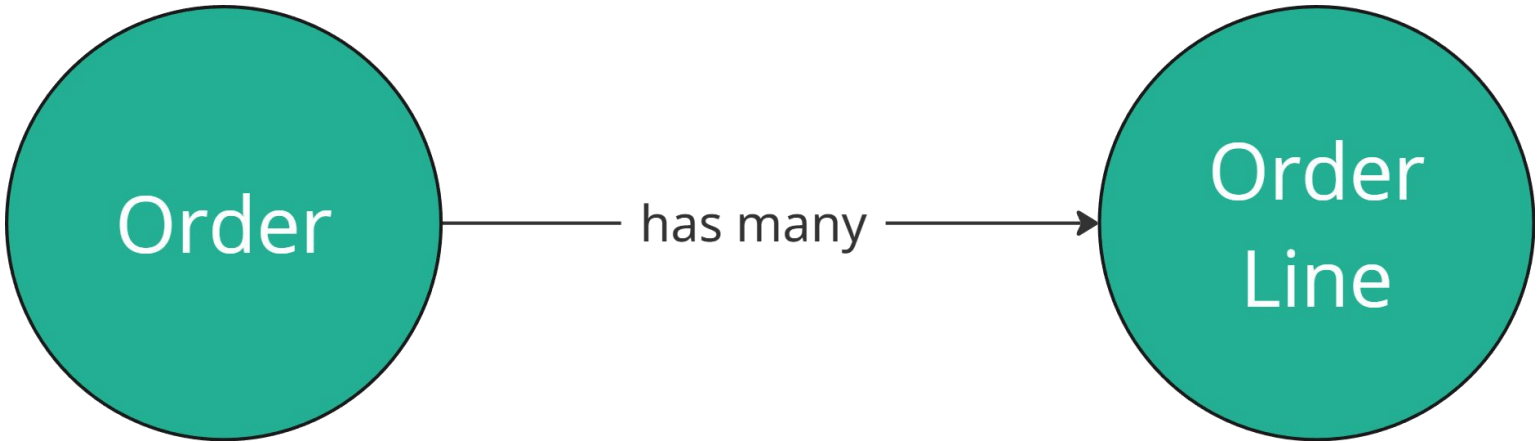
- Tactical DDD gives you a toolbox
- It doesn’t mean that you have to use all the tools

# Tip #3: Avoid bidirectional associations




- Increased complexity and coupling
- It may be harder to maintain consistency
- It may be harder to support this on the infrastructure level

# Tip #3: Prefer uni-directional associations



# Tip #4: Throw away your 1st model (and maybe the 2nd one)

- Be ready to experiment with your model and build the first draft
- The domain model in your code is not your baby
- It’s OK to throw the first, and even second, attempt
- Of course, it depends on the complexity and your skills

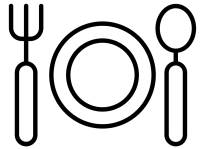
trainitek

# Tip #5: Good models are born outside the office

- Preload the knowledge in your brain, and then, you will get good ideas while walking or taking a shower







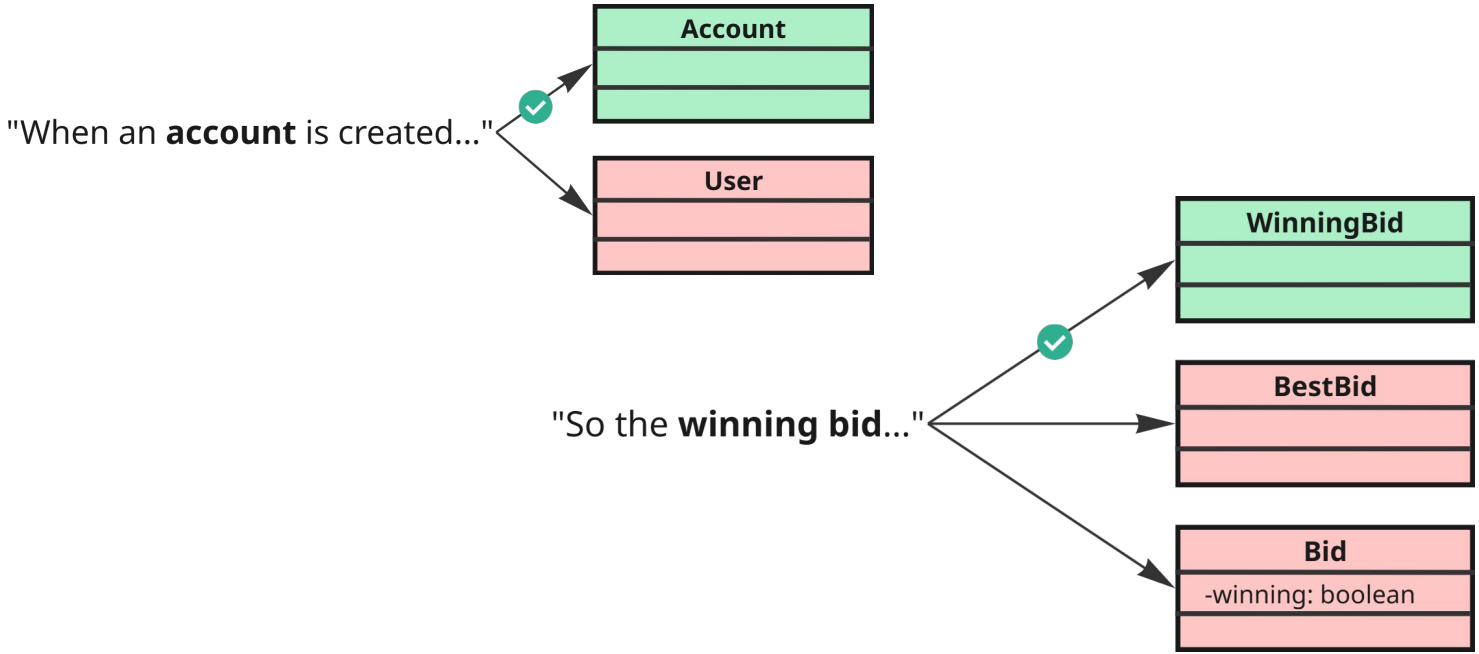
# Tip #6,7,8…

- There are more tips, however, we have limited time

# Important tips for writing code

# Tip #1: Stick to the Language

- The names you hear from domain experts should appear in the code.



trainitek

# Tip #2: Use Value Objects more often

- Usually, **primitive obsession** is not a good idea
- Domain concepts represented by Value Objects can positively affect:
  - Readability
  - Validation
  - Encapsulation
  - Testing
  - Maintenance

```
public class UserAccount {

    String firstName;
    String lastName;
```

```
record PersonName(@NonNull String firstName,
                  @NonNull String lastName) {
}
```

```
private BigDecimal priceAmount;
private Currency priceCurrency;
```

```
public final class Price {

    private BigDecimal amount;
    private Currency currency;
```

trainitek

# Tip #3: Think about behaviour rather than private fields

```
def "A new bid can be placed on the started auction"() {
    given: "started auction"

    when: "a new bid is placed"

    then: "the bid is accepted"
    and: "a new winning bid appears"
    and: "an event is added"

    where:
    startingPrice | bidPrice
    1.0           | 1.1
    1.0           | 10.0
}
```

```
public class Auction {

    ???
}
```

```
public class Auction {

    private List<Bid> bids;
    private Bid winningBid;

    public plac

    …

    winningB

    }
```

```
def "A new bid with a higher price outbids the previous bid"() {
    given: "auction with some bid"

    when: "a new bid is placed"

    then: "the bid is accepted"
    and: "the winning bid is changed"
    and: "an event is added"

    where:
    currentBidPrice | currentBidMaxPrice | newBidPrice | newBidMax
    500             | 500                | 600         | 600
    500             | 500                | 600         | 700
}
```

# Tip #3: Think about behaviour rather than private fields

- Define the first behaviour (API) and use value objects. No implementation.
- Write tests as future requirements for your model.
- Satisfy the tests by implementing the model. Add private fields only when you really need them.
- Verify the solution, consider improvements.

# Tip #4,5,6…

- There are more tips, however, we have limited time

# How to persist models

# First of all: be pragmatic

- On the one hand, you should avoid thinking about DB tables, FKs, etc.
- On the other hand, it can be really hard to avoid influence of technology when having strict requirements (high performance, low latency, etc.)
- Seek for the balance!

# The price of being a purist

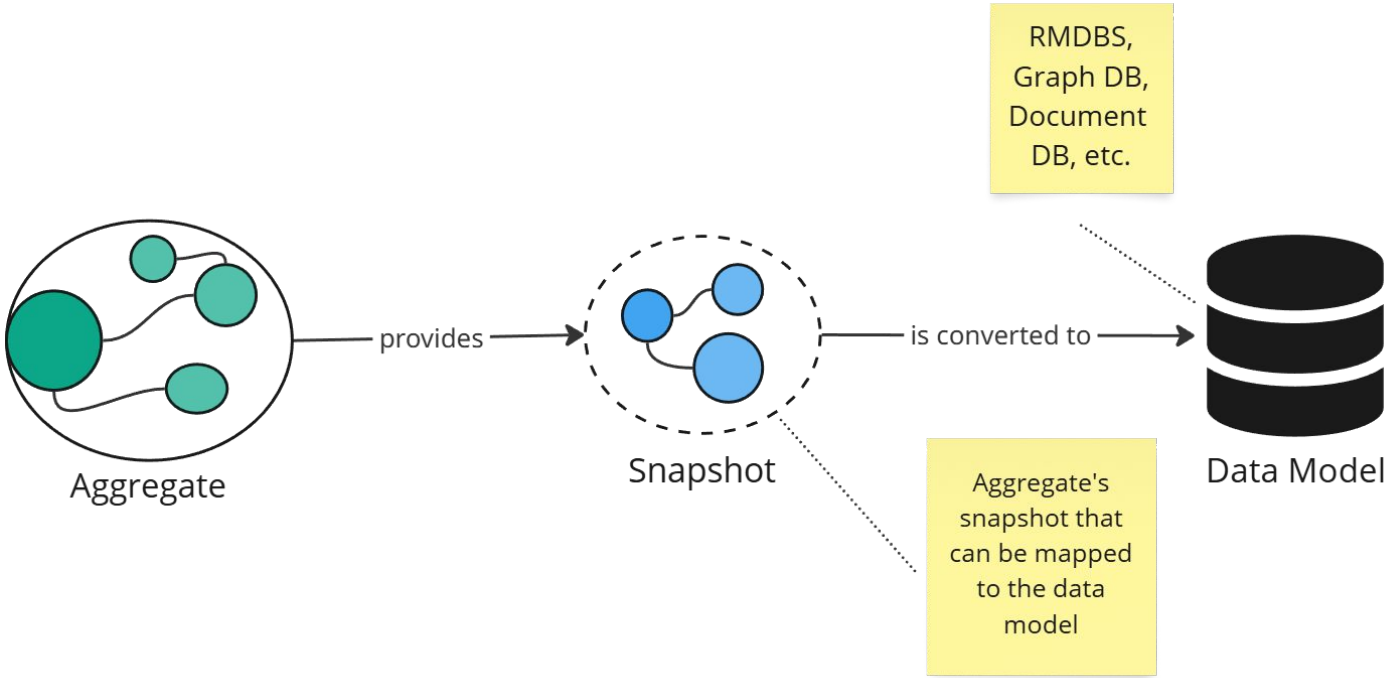
- In many cases, it’s just fine to use an ORM and have annotations/attributes in your domain model

- Yes, you will be restricted from writing any code you want, however, it’s always a compromise

```
@Entity
public class UserAccount extends UuidAggregateRoot {
```

# Consider Memento

- Sometimes, it’s hard/impossible to “map” your model to a database schema



# Repositories: main characteristics

- They keep domain knowledge away from the persistence framework
- They represent an explicit contract (API) to access aggregate roots
- A typical usage:

```
public UUID handle(ChangeBookDetails cmd) {
    Book book = repository.load(cmd.getId());
    book.changeDetails(cmd.getTitle(), cmd.getAuthor(),
            cmd.getDescription(), cmd.getKeywords(), clock);
    Book saved = repository.save(book);
    return saved.getId();
}
```

**Find an aggregate**

**Execute business operation**

**Persist the change**

# Repositories are NOT for

- Having hundreds of methods, e.g. for reporting.
  - In this case, move it somewhere else, e.g. using CQRS.

- Having vague methods like
    - `List<Aggregate> findAllMatching(Predicate query);`
    - `IEnumerable<Aggregate> FindAllMatching(Expression<Func<T, bool>> query);`
  - Use the domain language, e.g.
    - `List<Book> findBooksToCensor(Clock clock);`

- Returning partial aggregates
  - Aggregates should always be ready for use.

# Conclusions

- **Domains** provide a **context** for a **problem space**
- **Models** lives in the **solution space** and **addresses specific problems**
- **All models are wrong**
- You **should** create **small enough** and **useful models**
- Tactical DDD goes **after** Strategic DDD
- Tactical DDD provides a toolbox
- We went through building blocks and general tips for designing models and writing code
- **Be pragmatic**

# Where to learn more?

- You can contact me, of course.
- Read books like “Patterns, Principles, and Practices of Domain-Driven Design”
- Watch our talks like:
  - https://www.youtube.com/watch?v=iv5QGfWzqJ4
  - https://www.youtube.com/watch?v=jiT5AUpeAKs

trainitek

# Thank You!

**Presentation Slides**

vadim-prudnikov