Microservices - DDD = Micro-monolith

#ThumbtackJavaMeetup

Vadim Anosov

27/03/2019

Agenda

I. What is Domain-Driven Design?

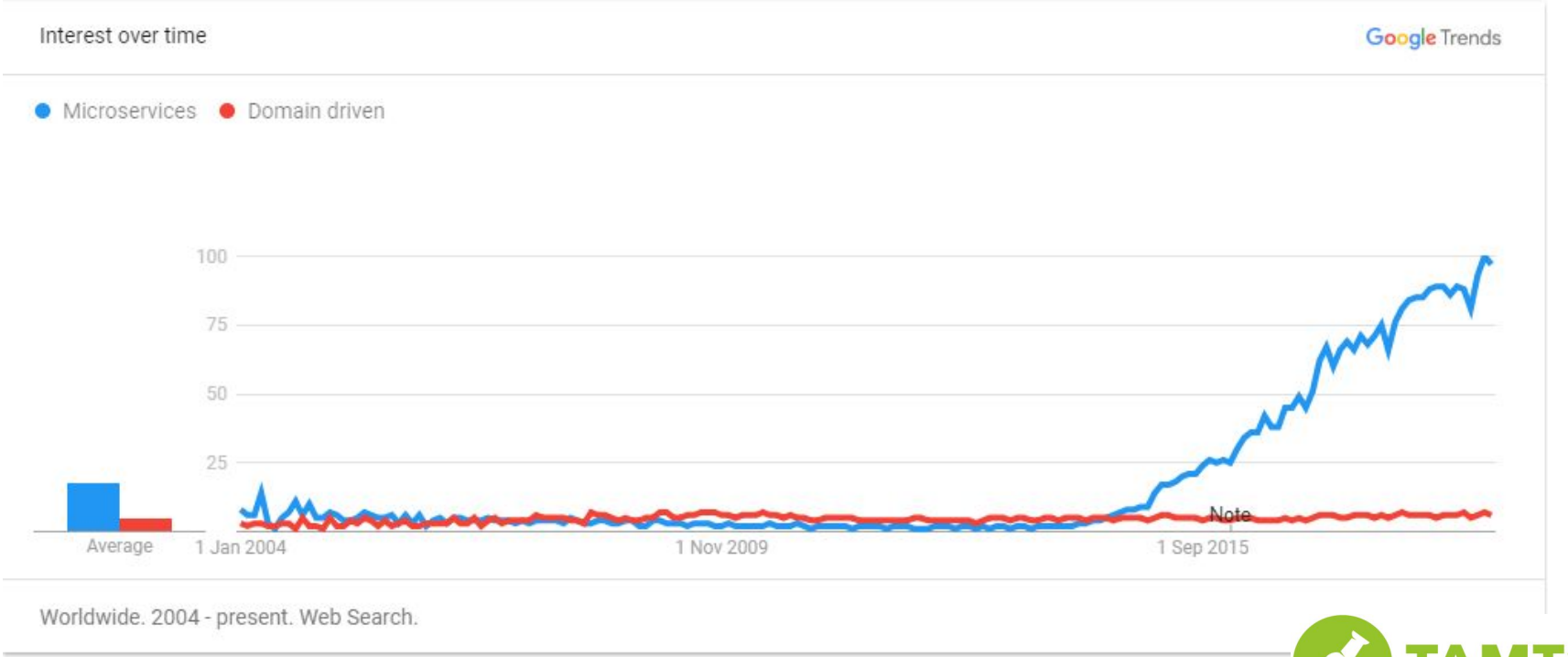
II. What are the key concepts of the DDD approach?

III. How DDD helps to define an application's microservice architecture?

IV. Why design microservice architecture without DDD concepts is the way to Micro-monolith?

Trends: Microservices vs Domain-Driven Design

Trends: Microservices vs Domain-Driven Design



"Data source: Google Trends (www.google.com/trends)"

Greater agility

Comprehensibility

Independent Development

Technology diversity

Testability

Better fault tolerance

Better scalability

Independent Deployment

Faster time to market

Faster development cycles

ТАМТЭК

What is the right size of a service in the microservice architecture?

What is the right size of a service in the microservice architecture?

ТАМТЭК

What is the right size of a service in the microservice architecture?

Microservice

=

Business capability

- ★ The most meaningful separation guided by **domain knowledge**
- ★ The emphasis isn't on the size, but instead on **business capabilities**

Agenda

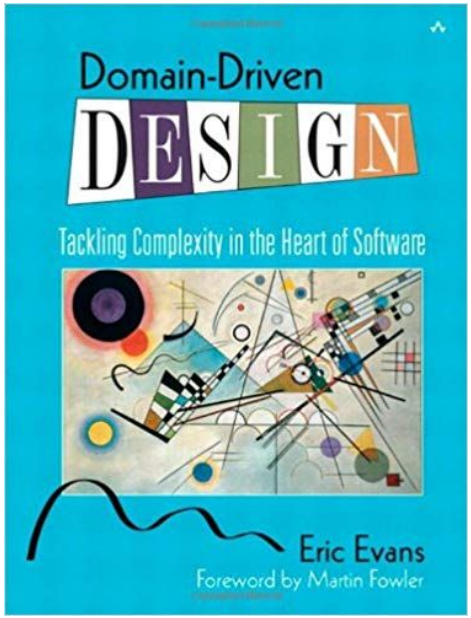
I. What is Domain-Driven Design?

II. What are the key concepts of the DDD approach?

III. How DDD helps to define an application's microservice architecture?

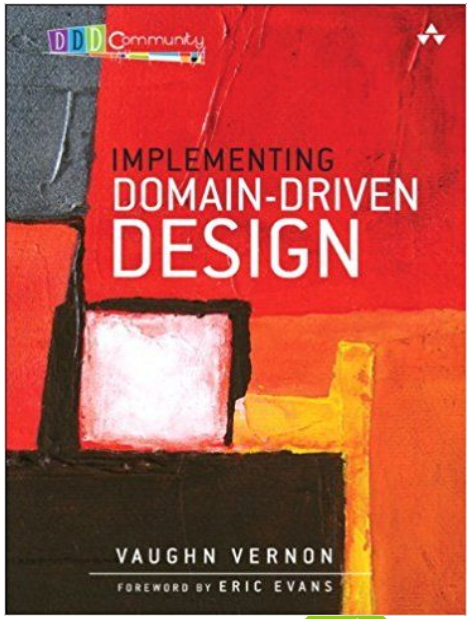
IV. Why design microservice architecture without DDD concepts is the way to Micro-monolith?

Domain-Driven Design



“Domain-Driven Design: Tackling Complexity in the Heart of Software”
by Eric Evans

“Implementing Domain-Driven Design”
by Vaughn Vernon



Domain-Driven Design

DDD is an approach for building **complex** software applications that is centered on the development of an object-oriented **domain model**.

Designing a city analogy

Unplanned

Big Ball Of Mud

Planned

Domain Driven Design

Agenda

I. What is Domain-Driven Design?

II. What are the key concepts of the DDD approach?

III. How DDD helps to define an application's microservice architecture?

IV. Why design microservice architecture without DDD concepts is the way to Micro-monolith?

Domain-Driven Design

Strategic patterns

- Sub-domain
- Ubiquitous Language
- Continuous Integration
- Domain
- Bounded Context
- Context Map
- Domain model

Tactical patterns

- Repositories
- Services
- Entities
- Factories
- Modules
- Layers
- Aggregates
- Value Objects

Agenda

I. What is Domain-Driven Design?

II. What are the key concepts of the DDD approach?

III. How DDD helps to define an application's microservice architecture?

IV. Why design microservice architecture without DDD concepts is the way to Micro-monolith?

Microservices dilemma

Monolith first

Microservices first

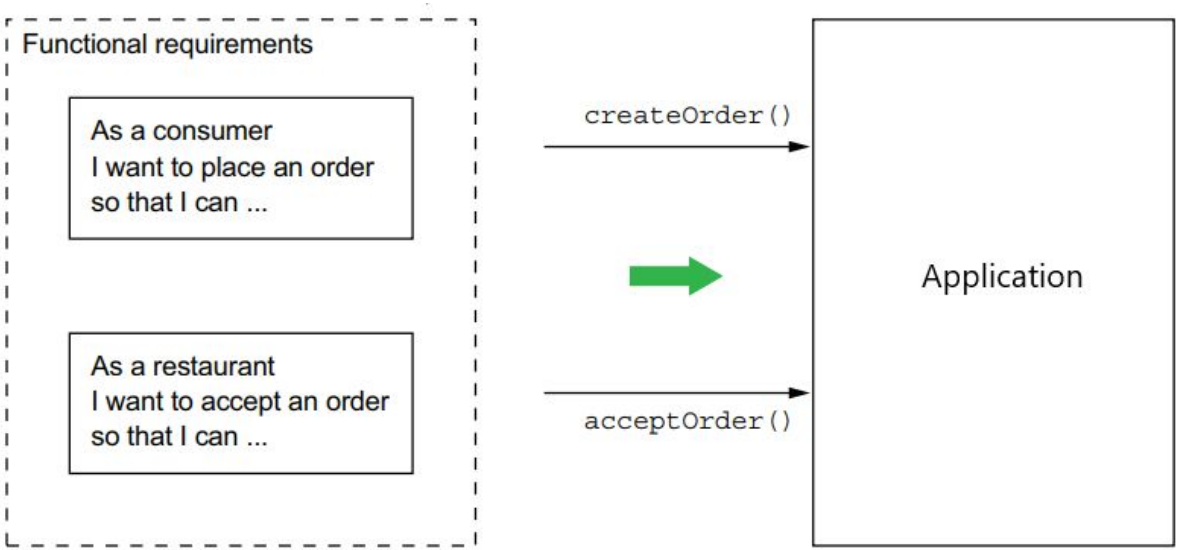
ТАМТЭК

Three steps to defining an application's microservice architecture

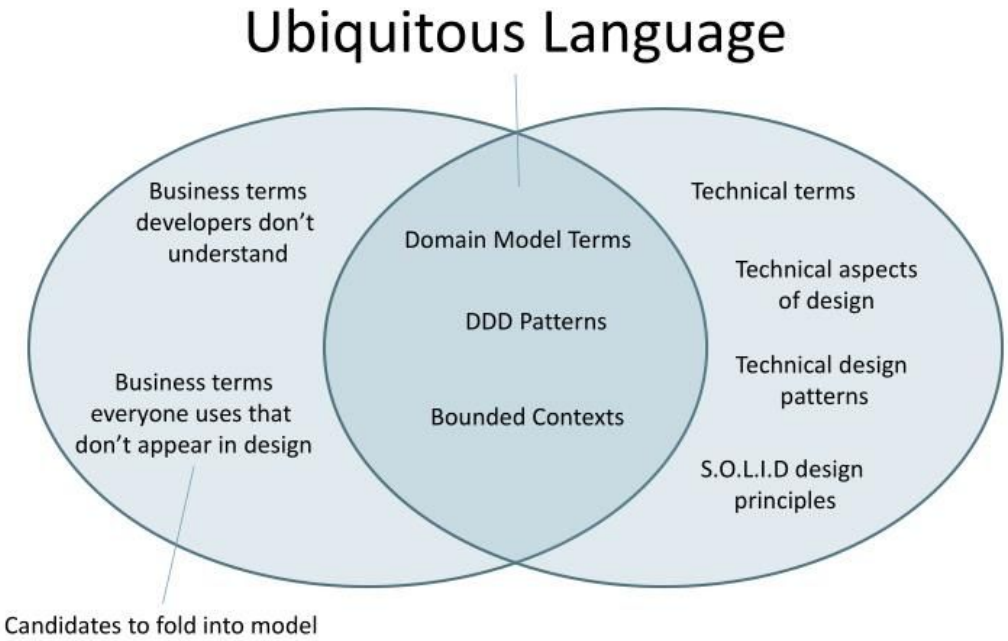
- ★ Identify system operations
- ★ Identify services
- ★ Define service APIs and collaborations

Three steps to defining an application's microservice architecture

★ Identify system operations:



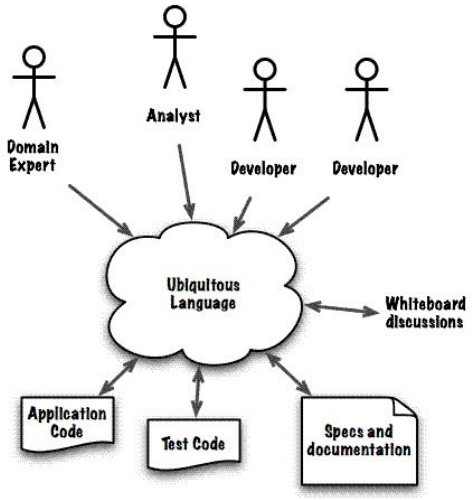
DDD toolbox: Ubiquitous Language



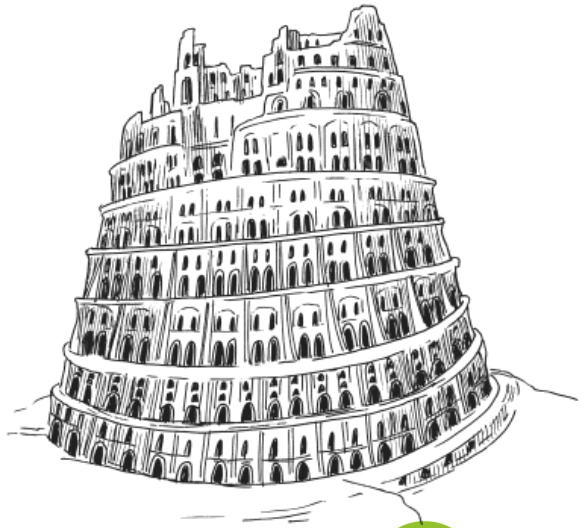
ТАМТЭК

DDD toolbox: Ubiquitous Language

Turn on

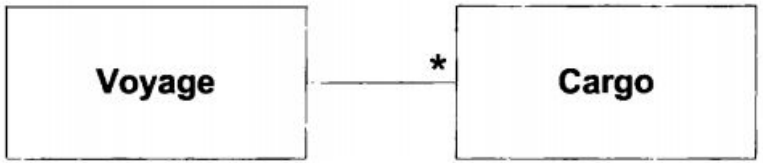


Turn off



ТАМТЭК

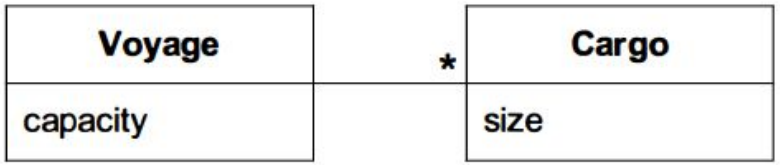
Ubiquitous Language: Extracting a Hidden Concept




```
public int makeBooking(Cargo cargo, Voyage voyage) {
   int confirmation = orderConfirmationSequence.next();
   voyage.addCargo(cargo, confirmation);
   return confirmation;
}
```

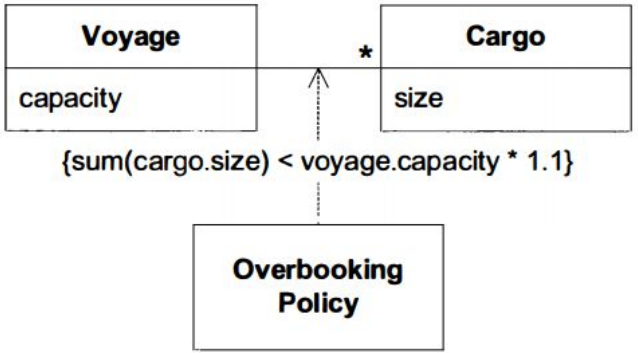
ТАМТЭК

Ubiquitous Language: Extracting a Hidden Concept



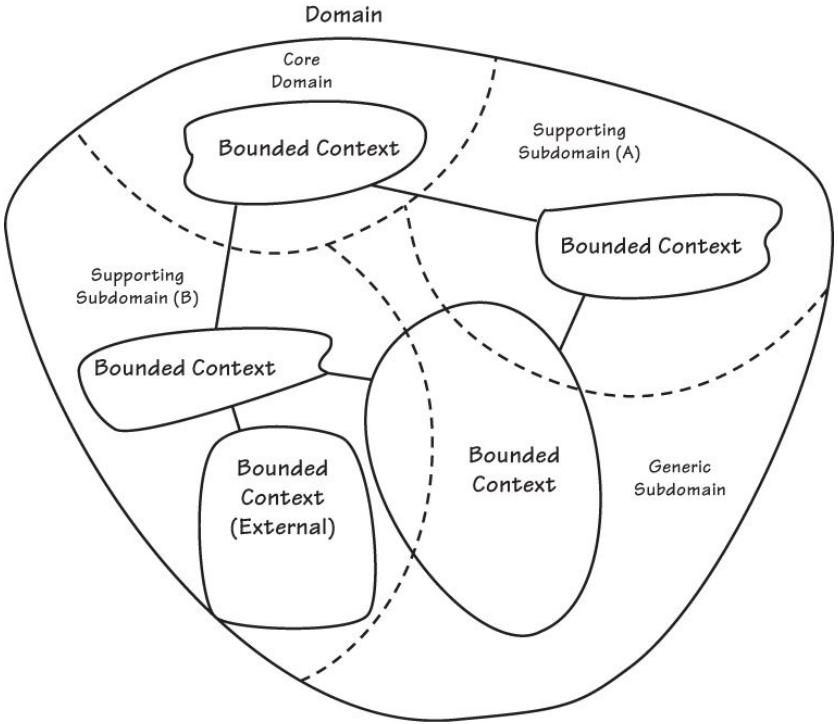

```
public int makeBooking(Cargo cargo, Voyage voyage) {
    double maxBooking = voyage.capacity() * 1.1;
    if ((voyage.bookedCargoSize() + cargo.size()) > maxBooking)
         return -1;
    int confirmation = orderConfirmationSequence.next();
    voyage.addCargo(cargo, confirmation);
    return confirmation;
}
```

Ubiquitous Language: Extracting a Hidden Concept




```
public int makeBooking(Cargo cargo, Voyage voyage) {
   if (!overbookingPolicy.isAllowed(cargo, voyage)) return -1;
   int confirmation = orderConfirmationSequence.next();
   voyage.addCargo(cargo, confirmation);
   return confirmation;
}
```

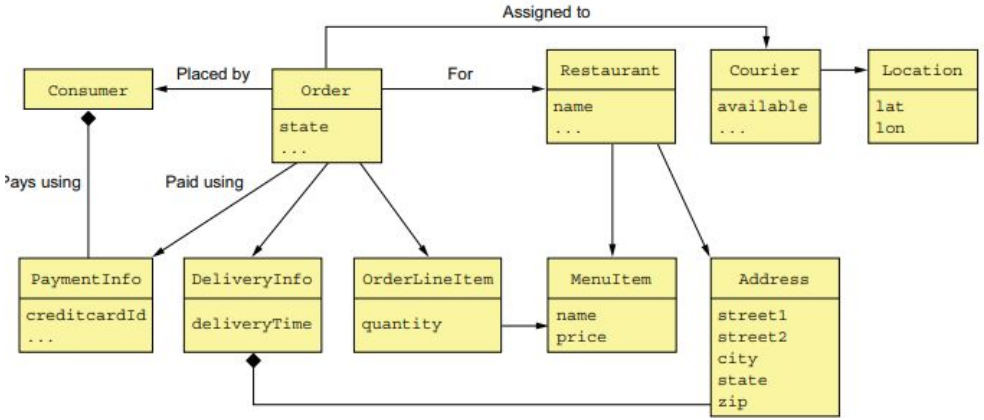
DDD toolbox: Domain, Subdomain

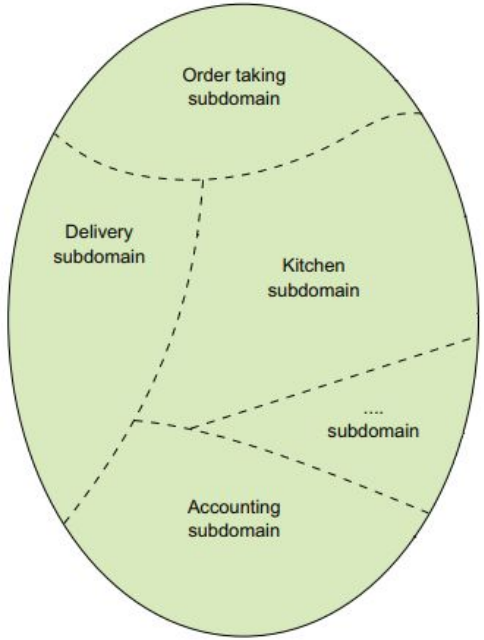


ТАМТЭК

Result of using Ubiquitous Language, Domain and Subdomain

```
Given a consumer
  And a restaurant
  And a delivery address/time that can be served by that restaurant
  And an order total that meets the restaurant's order minimum
When the consumer places an order for the restaurant
Then consumer's credit card is authorized
  And an order is created in the PENDING_ACCEPTANCE state
  And the order is associated with the consumer
  And the order is associated with the restaurant
```

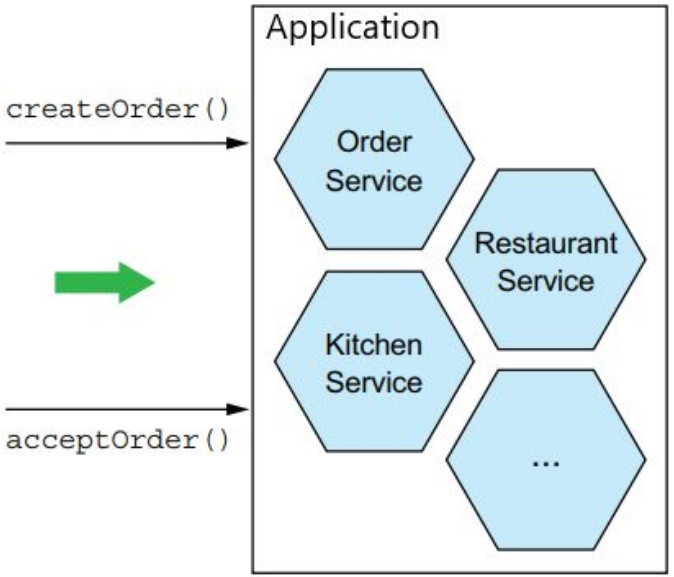




ТАМТЭК

Three steps to defining an application's microservice architecture

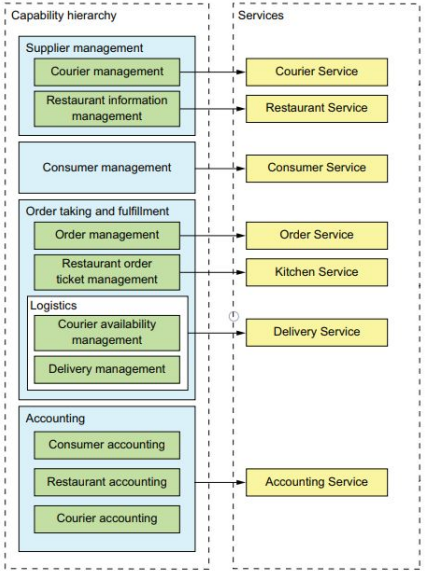
★ Identify services:



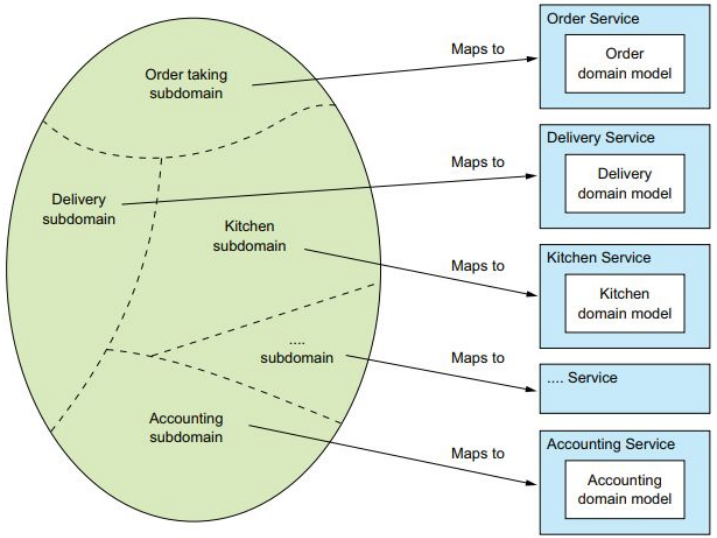
ТАМТЭК

Patterns for decomposing an application into services

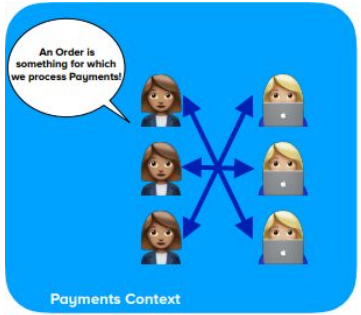
Decompose by business capability



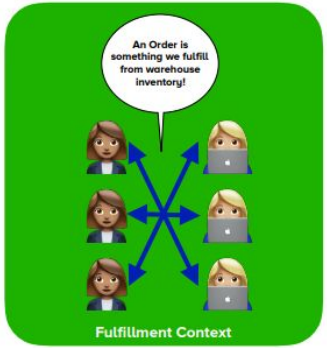
Decompose by subdomain

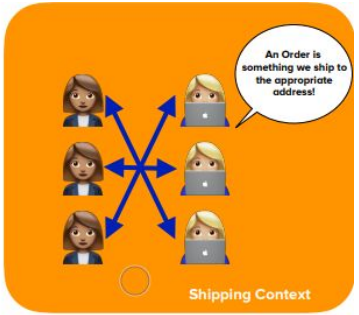


DDD toolbox: Bounded Context



Explicitly define the context within which a model is applied





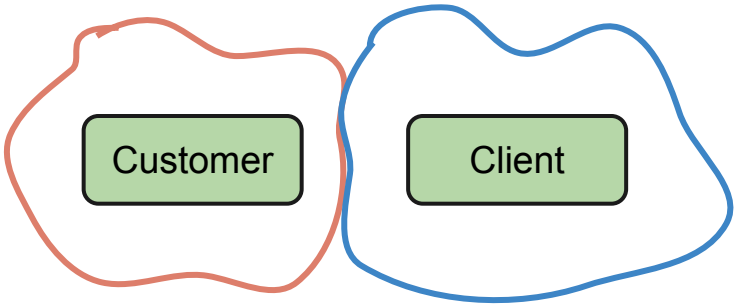
Keep the model strictly consistent within these bounds

Explicitly set boundaries in terms of team organization

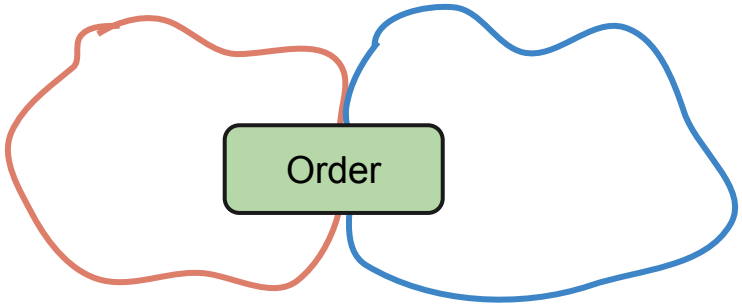
ТАМТЭК

Bounded Context: possible problems

Duplicate concepts



False cognates



God classes preventing decomposition

```
public class Order {

    private OrderTotal orderTotal;

    private DateTime deliveryTime;

    private DateTime pickupTime;

    private BigInteger transactionId;

                . . .

    public void createOrder(){...}

    public void cancelOrder(){...}

    public void acceptOrder(){...}

    public void rejectOrder(){...}

    public Note noteReadyForPickup(){...}

    public void assignCourier(Courier courier){...}

    public Note notePickedUp(){...}

    public Note noteDelivered(){...}

                . . .

}
```

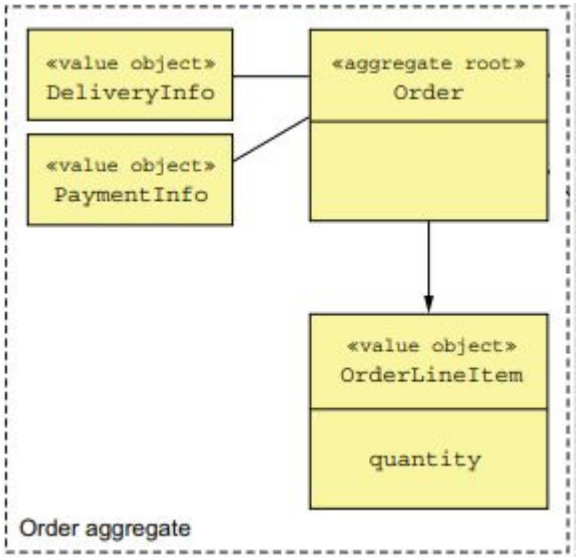
God classes preventing decomposition

```
public class Order {

    private OrderTotal orderTotal;

    private DateTime deliveryTime;

    private DateTime pickupTime;                    <<delivery>>

    private BigInteger transactionId;               <<billing>>

                . . .

    public void createOrder(){...}
                                                    <<orderTaking>>
    public void cancelOrder(){...}

    public void acceptOrder(){...}

    public void rejectOrder(){...}                  <<restaurant>>

    public Note noteReadyForPickup(){...}

    public void assignCourier(Courier courier){...}

    public Note notePickedUp(){...}                 <<delivery>>

    public Note noteDelivered(){...}

                . . .

}
```

ТАМТЭК

DDD toolbox: Aggregate

Only accessed through its Root Entity.



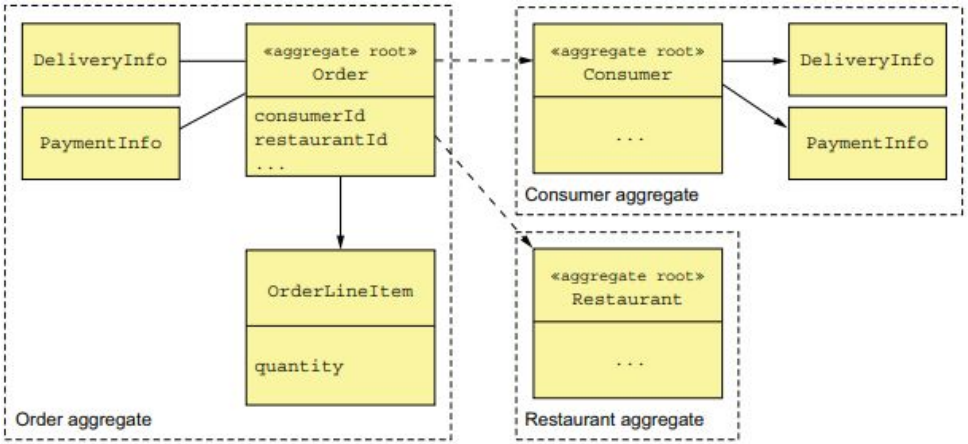
A cluster of objects treated as a single unit.

Responsible for maintaining any/all business invariants.

The atomic unit for any transactional behavior.

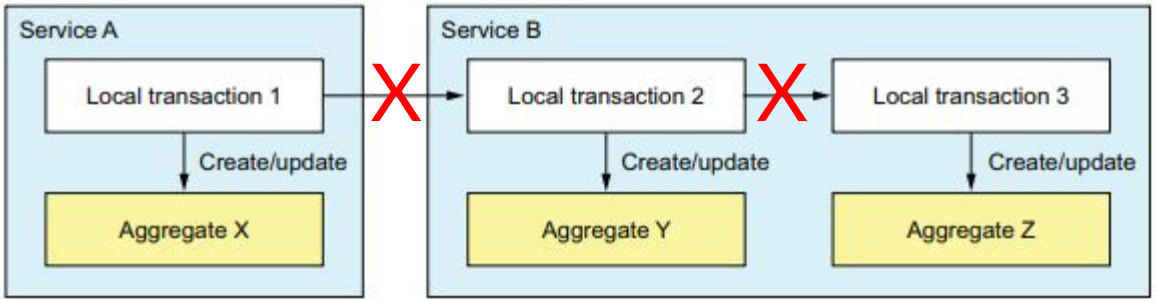
ТАМТЭК

Aggregate: Rule #1



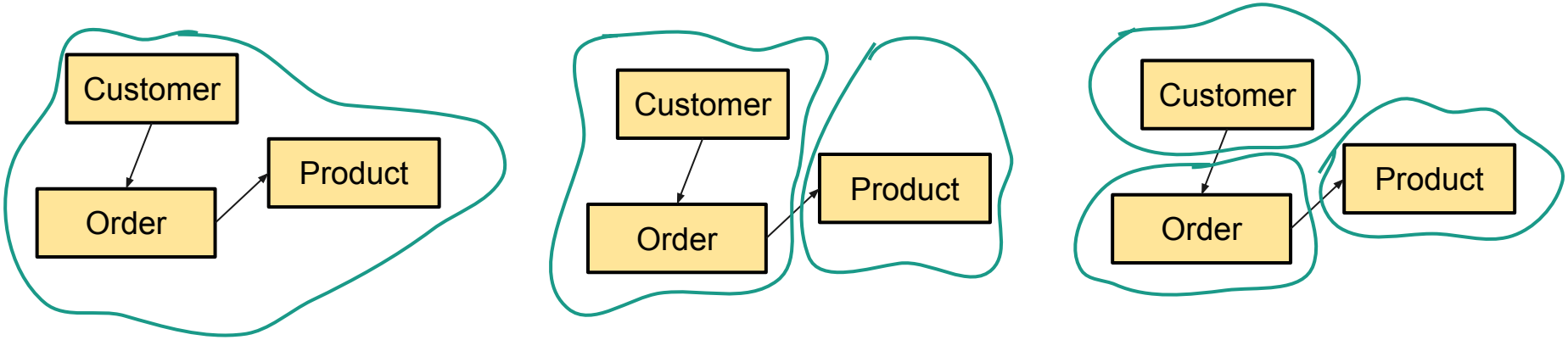
Reference other aggregate roots
via identity (primary key)

Aggregate: Rule #2



One transaction creates or
updates one aggregate
(Transaction scope = service)

Aggregate granularity



Consistency ⟷ Scalability

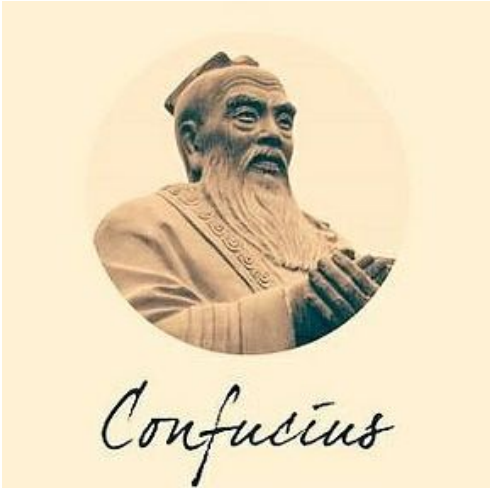
DDD & Microservices

★ Apply strategic DDD **to identify microservices** (bounded context, ubiquitous language, context map)

★ Apply tactical DDD **to design individual services** (aggregator, value object, service)

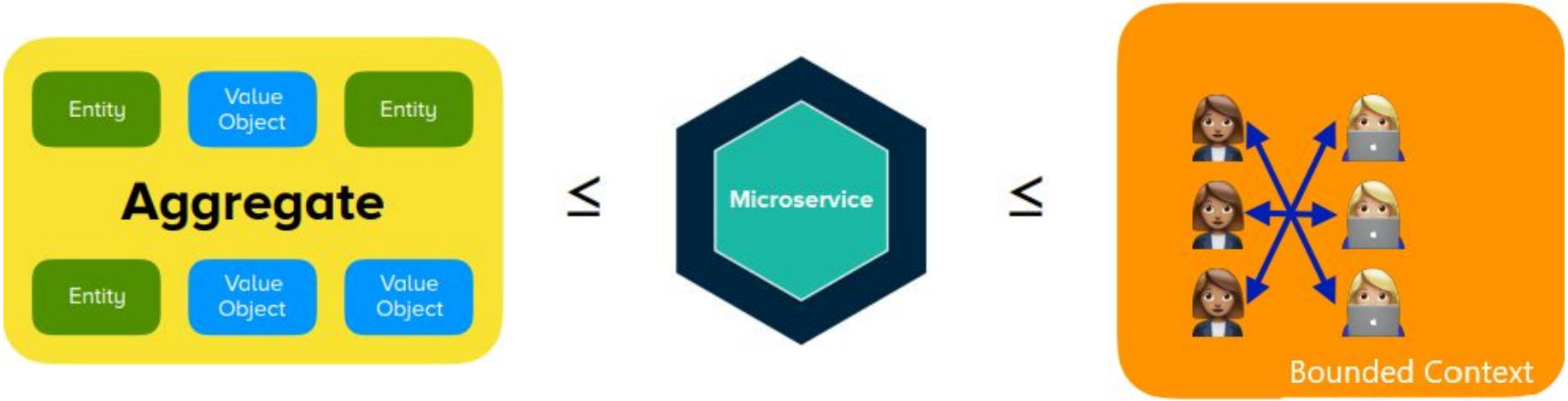
What is the right size of a service in the microservice architecture?

"...Microservice should be **no smaller than an aggregate**, and **no larger than a bounded context...**"



ТАМТЭК

What is the right size of a service in the microservice architecture?



TAMTЭK

Useful links

Domain-Driven Design: Tackling Complexity in the Heart of Software

Implementing Domain-Driven Design

Microservices Patterns: With examples in Java

Building Microservices: Designing Fine-Grained Systems

Martin Fowlers blog: DDD

DDD Europe

ТАМТЭК

Thank you for your attention!

ТАМТЭК