INSTITUTO FEDERAL
Paraíba
Campus Cajazeiras

PROGRAMAÇÃO P/ WEB 2

3

6. PROJETANDO LÓGICA DE NEGÓCIO NUMA ARQUITETURA DE MICROSSERVIÇOS

PROF. DIEGO PESSOA

DIEGO.PESSOA@IFPB.EDU.BR

@DIEGOEP

CST em Análise e Desenvolvimento de Sistemas

VISÃO GERAL

- Desenvolver lógica de negócio é mais complexa numa arquitetura de microsservicos, já que a lógica está espalhada em múltiplos serviços.

- Principais desafios:
  - 1. Referências entre objetos em serviços diferentes
  - 2. Como encaixar a lógica de negócio dentro do gerenciamento de transações da arquitetura de microsserviços

VISÃO GERAL

- Solução: utilizar o padrão *Aggregate* do conceito de *Domain Driven Design* (DDD).
- Estrutura lógica de negócio como uma coleção de *Aggregates*
- *Aggregate* = conjunto de objetos que podem ser tratados como uma unidade

VISÃO GERAL

- Por que aggregates são úteis?
  - 1. Evitam referência de objetos espalhados além das fronteiras do serviço. Uma única chave primária como referência *inter-aggregate* é melhor do que várias referências para objetos internos.
  - 2. Uma transação pode criar ou modificar um único *aggregate*, o que se encaixa nas restrições do modelo de transações de microsserviços.

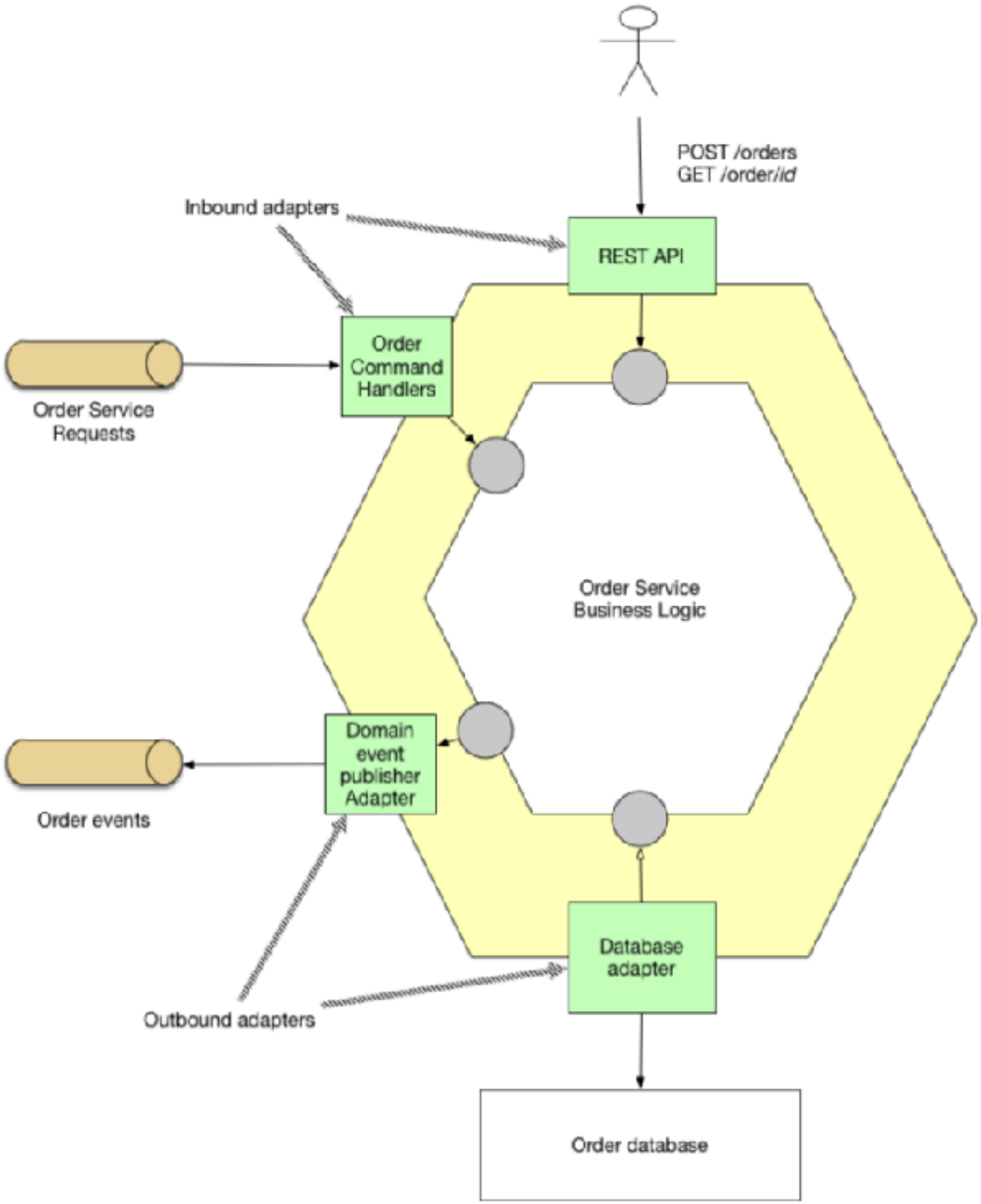
RESUMO

- 1. Aplicando padrões de organização de lógica de negócio: transaction script e domain model
- 2. Projetando lógica de negócio com Domain-driven design (DDD)
- 3. Aplicando o padrão Domain event numa arquitetura de microsserviços

PADRÕES PARA ORGANIZAÇÃO DE LÓGICA DE NEGÓCIO

ARQUITETURA TÍPICA DE LÓGICA DE NEGÓCIOS (HEXAGONAL)

- inbound/outbound adapters
- "Business logic" é geralmente a camada mais complexa de um serviço

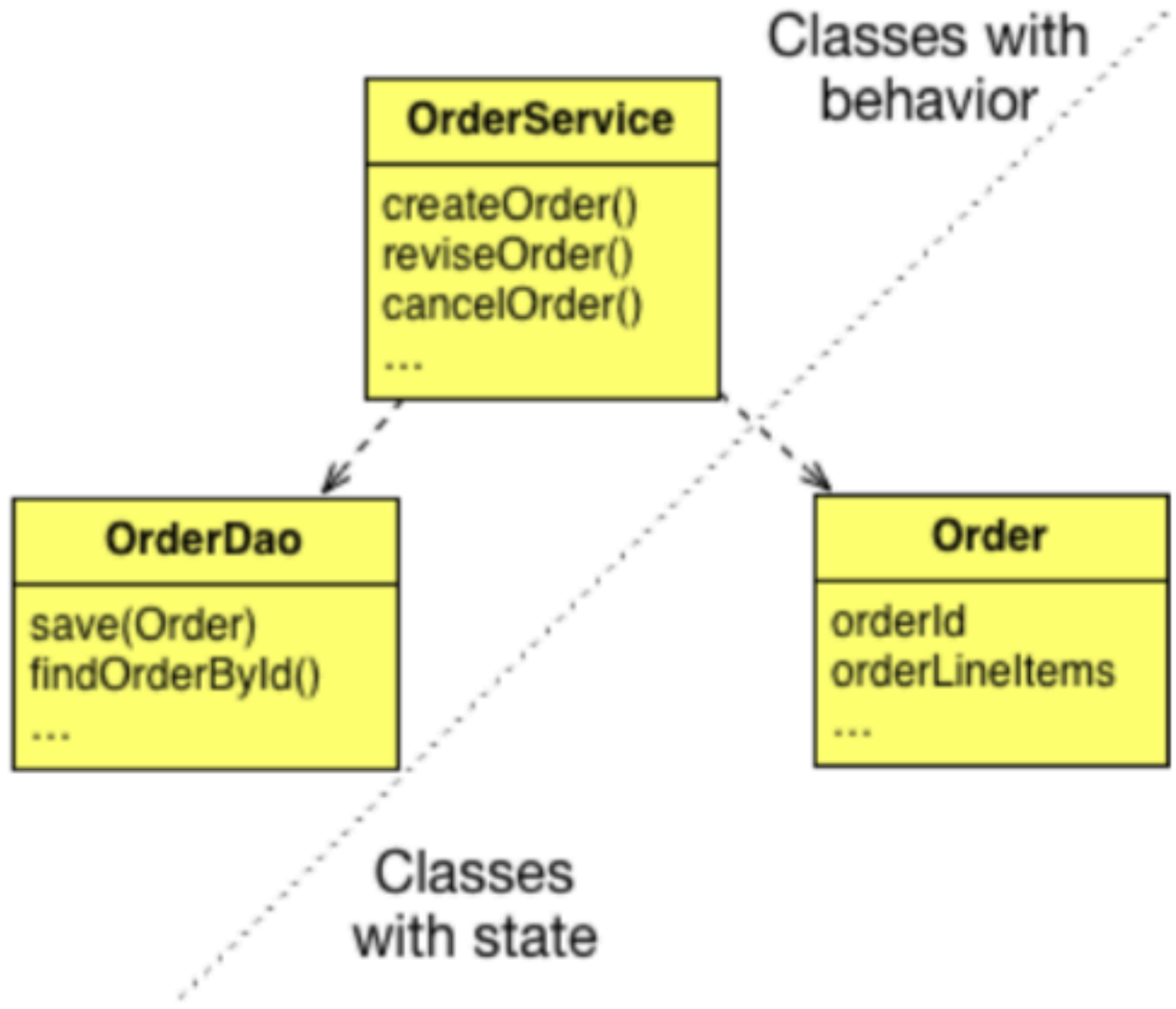


ORGANIZA A LÓGICA DE NEGÓCIO COMO UMA COLEÇÃO DE SCRIPTS PROCEDURAIS, UM PARA CADA TIPO DE REQUISIÇÃO

Padrão Transaction Script

USANDO O PADRÃO "TRANSACTION SCRIPT"

- Um conjunto de classes implementa **comportamento** e outro conjunto guarda **estado**
- Os "transaction scripts" são organizados em classes que não guardam estado
- Os scripts usam classes de dados, que tipicamente não gerenciam comportamento



PADRÃO TRANSACTION SCRIPT

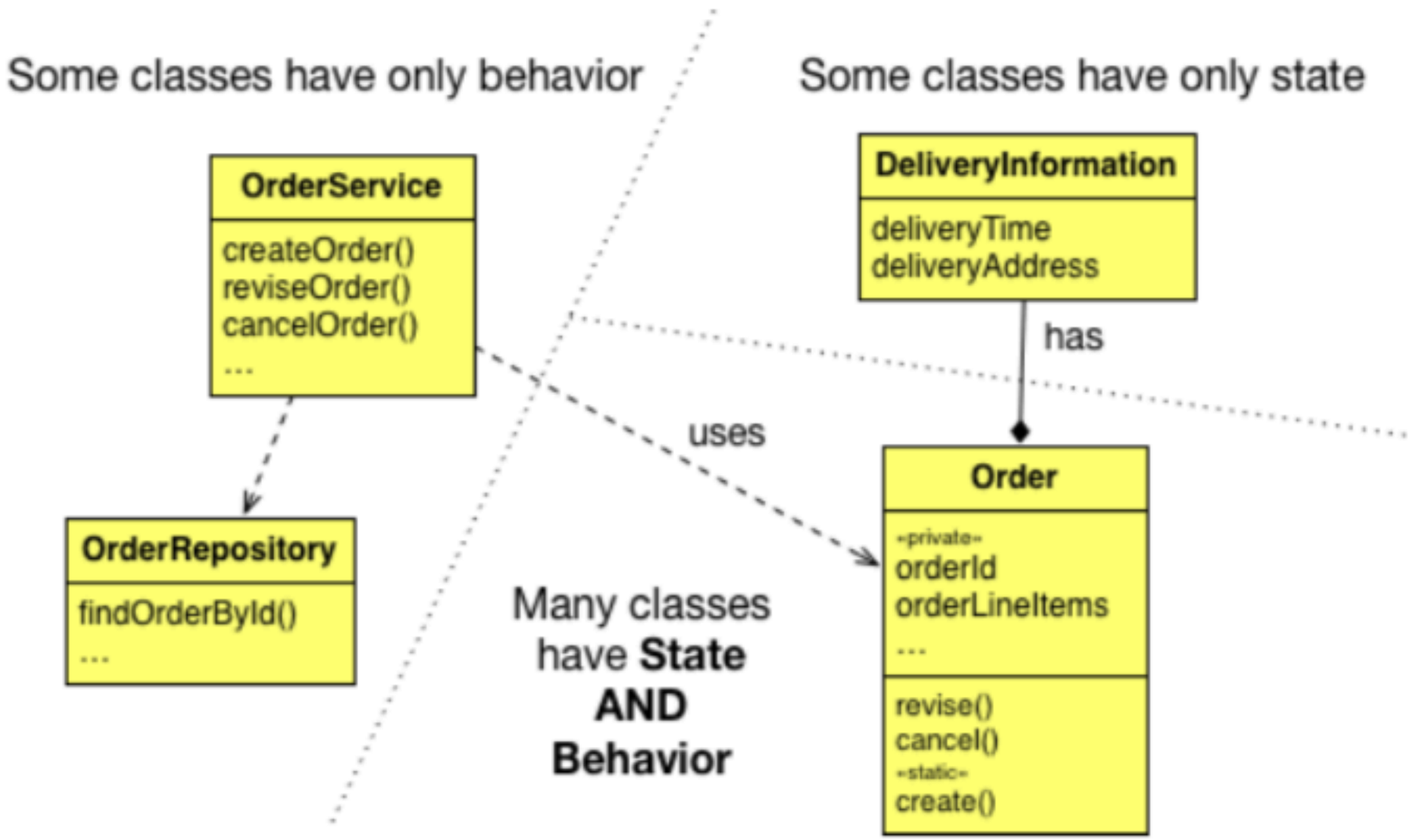
- Scripts geralmente concentrados em classes de Serviço
- Classe de serviço tem um método para cada requisição, que implementa a lógica de negócio para ela
- Classe de serviço acessa DAOs.
- Os dados de objetos são puros com pouco ou nenhum comportamento

ORGANIZA A LÓGICA DE NEGÓCIO COMO CLASSES CONSISTENTES CONTENDO ESTADO E COMPORTAMENTO

Padrão Domain Model

PADRÃO DOMAIN MODEL

- A maior parte das classes de negócio gerenciam estado e comportamento



PADRÃO DOMAIN MODEL

- Da mesma forma que o "Transaction Script", a classe de serviço possui um método para cada requisição.
- Porém, como o serviço sempre delega as operações aos objetos persistentes, eles se tornam mais simples
- Um serviço pode simplesmente carregar um objeto de domínio do banco e invocar um de seus métodos

PADRÃO DOMAIN MODEL

- Benefícios:
  - 1. Mais fácil de entender/manter ao invés de ter uma única classe que faz tudo
    - Consiste de um número pequeno de classes que possui um pequeno número de responsabilidades
      - Classes como “Conta” e “TransaçãoBancária" se tornam mais próximas do mundo real, o que torna mais fácil o entendimento
  - 2. Mais fácil de estender um projeto orientado a objetos porque é possível usar padrões como Strategy e Template Method [GoF] para definir formas de estender um objeto sem modificar o código
- No entanto, para uma arquitetura de microsserviços, esse padrão apresenta uma série de problemas, que demandam um refinamento chamado *Domain Driven Design*.

DOMAIN DRIVEN DESIGN (DDD)

- Ao usar DDD, cada serviço possui seu próprio modelo de domínio. Subdomínios e o conceito de "Bounded Context".

- DDD também aplica vários padrões para auxiliar na modelagem de componentes de um domínio, cada um com seu papel

DOMAIN DRIVEN DESIGN (DDD)

- Componentes:
  - Entity - objeto que possui uma identidade persistente. Duas entidades com os mesmos valores ainda são diferentes. (Classe anotada com @Entity)
  - Value Object - objeto que é uma coleção de valores. Podem ter os mesmo valores que podem ser usados de maneira intercambiável. (Ex.: classe *Money*)
  - Factory - objeto ou método que implementa a lógica para criação de um objeto
  - Repository - um objeto que provê acesso a entidades persistentes e encapsula o mecanismo para acessar o banco
  - Service - um objeto que implementa lógica de negócio que não pertence a uma entidade ou value object.

DOMAIN DRIVEN DESIGN (DDD)

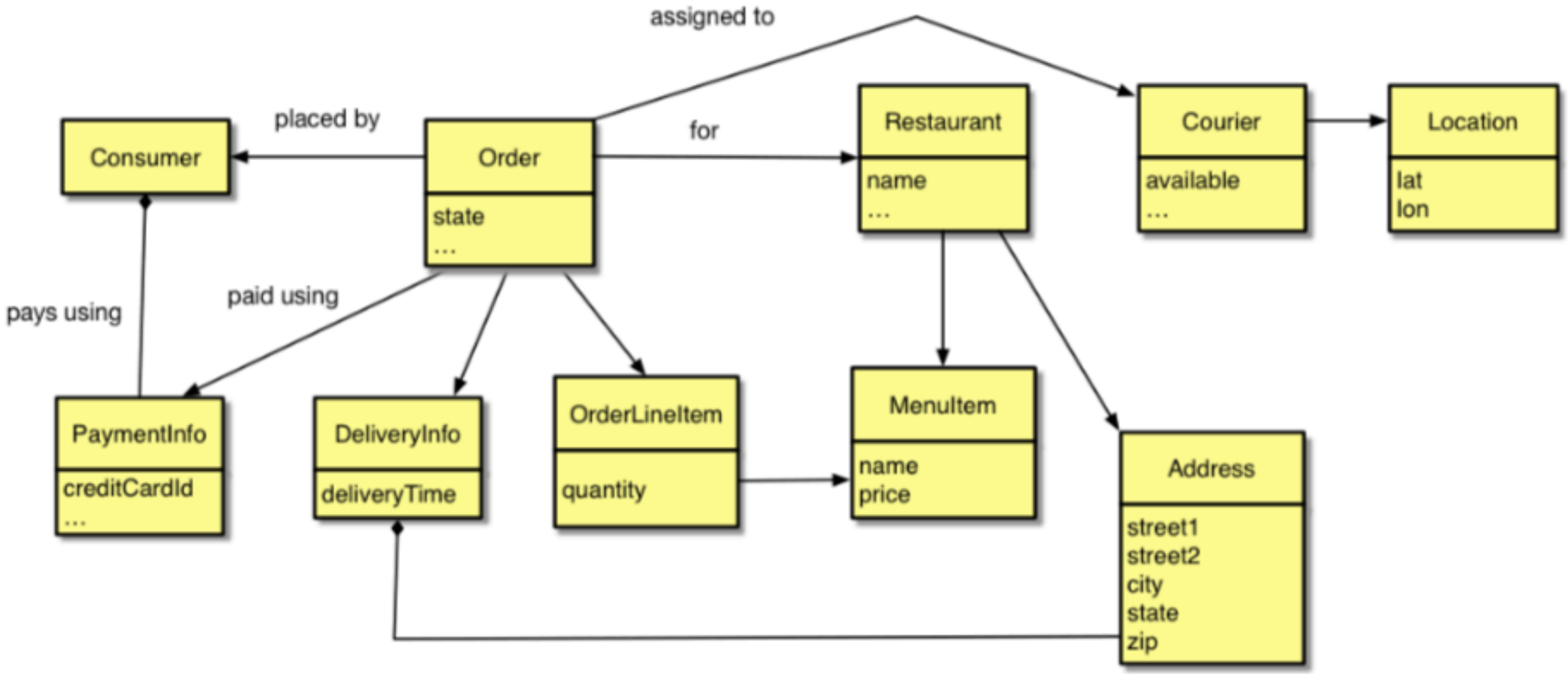
- Esses componentes são utilizados por muitos desenvolvedores e muitos são suportados por vários frameworks, como JPA e Spring Framework.

- No entanto, há um componente que geralmente é ignorado: *aggregates*

- Por outro lado, o conceito de *Aggregates* é extremamente útil para desenvolver microsserviços

PROJETANDO MODELO DE DOMÍNIO COM O PADRÃO DDD AGGREGATE

- O modelo de dados tradicional OO é uma coleção de classes organizadas em pacotes e relacionamentos.



PROJETANDO MODELO DE DOMÍNIO COM O PADRÃO DDD AGGREGATE

- Exemplo possui muitas entidades correspondentes ao negócio: Consumer, Order, Restaurant, Courier.
- Mas as fronteiras de cada objeto faltam nesse modelo tradicional
- Ex.: Ele não específica quais classes são parte do domínio Order.
- A falta dessas fronteiras pode eventualmente causar problemas, especialmente se tratando de uma arquitetura de microsserviços

O PROBLEMA COM A FALTA DE FRONTEIRAS

- Como tratar restrições em objetos relacionados?
- A checagem de concorrência é feita no objeto principal
- Exemplo: Pedido -> ItemDePedido com orçamento máximo e acesso colaborativo por dois usuários

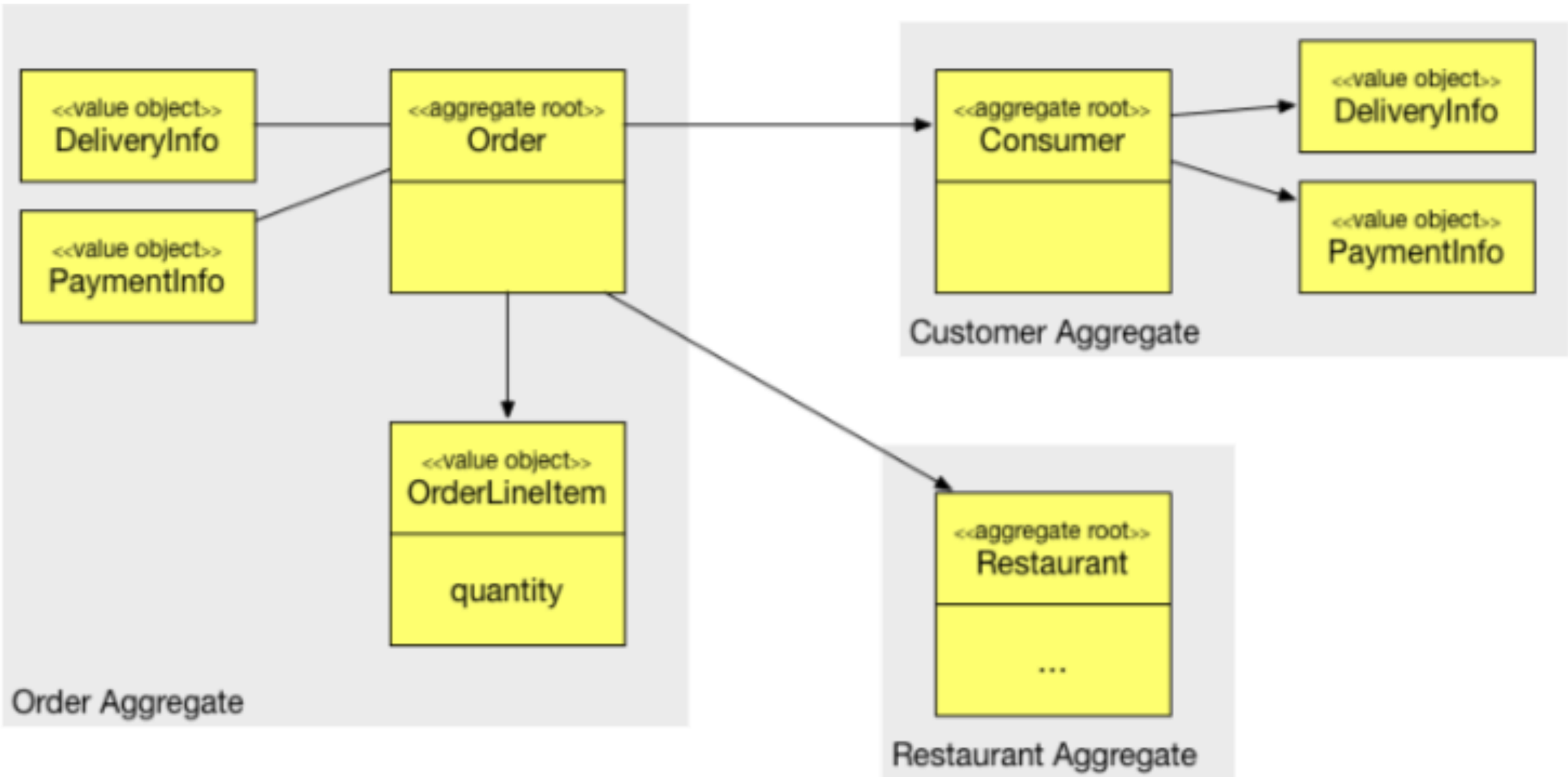
AGGREGATES POSSUEM FRONTEIRAS EXPLÍCITAS

- Um *aggregate* é um conjunto de objetos de domínio dentro de uma fronteira que pode ser tratada como unidade
- Consiste de uma entidade "raiz" e possivelmente uma ou mais entidades e value objects.
- Muitos "business objects" são modelados como *aggregates*.
- Em exemplos anteriores temos visto Pedido, Consumidor e Restaurante como *aggregate*.

ORGANIZA UM MODELO DE DOMÍNIO COMO UMA COLEÇÃO DE AGGREGATES, EM QUE CADA UMA É UM CONJUNTO DE OBJETOS TRATADOS COMO UNIDADE

Padrão Aggregate

EXEMPLO DE MODELO DE DOMÍNIO COM AGGREGATES



AGGREGATES SÃO FRONTEIRAS CONSISTENTES

- Atualizar uma *aggregate* inteira ao invés de suas partes resolve problemas de consistência (exemplo anterior)
- Operações de atualização são invocadas na entidade raiz.
- A concorrência é tratada com locking, por exemplo, usando um número de versão
- Obs.: nem sempre é necessário atualizar toda a aggregate, só quando houver essa demanda

IDENTIFICAR AGGREGATES É A CHAVE

- Em DDD, a parte essencial do projeto de um modelo de dados é identificar os *aggregates*, suas fronteiras e suas entidades raizes.

- Os detalhes da estrutura interna das *aggregates* se tornam secundários

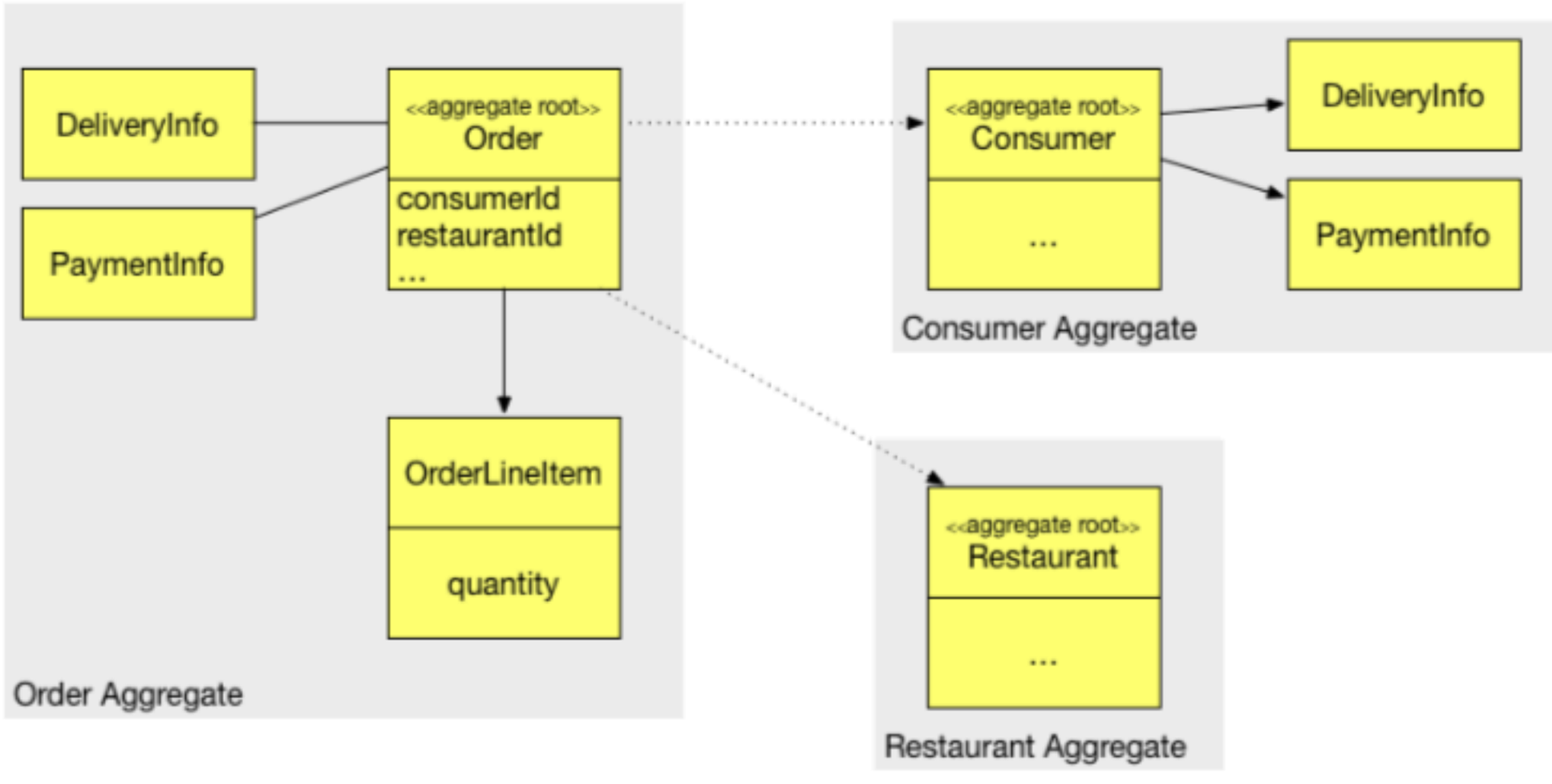
- O benefício das *aggregates*, no entanto, vai além da modularização, porque elas devem seguir regras.

REGRAS DE AGGREGATES

- Regra #1 - Só referencie uma entidade como raiz
  - Concentrar a raiz como um único objeto resolve o problema de atualizar diretamente um objeto relacionado
- A classe raiz é o único ponto de acesso fora da aggregate e ela será o único ponto que poderá ser invocado para atualizações

REGRAS DE AGGREGATES

- Regra #2 - Referências Inter-aggregates devem usar chaves primárias



REGRAS DE AGGREGATES (REGRA #2)

- Esta abordagem é um pouco diferente da modelagem de objetos tradicional, que consideraria uma violação a existência de foreign keys na entidade do modelo.
- No entanto, essa abordagem possui uma série de benefícios:
  - 1. Menor acoplamento
  - 2. Garante que as fronteiras entre aggregates são bem definidas e evita acidentalmente atualizar uma aggregate diferente (ex.: cascade)
  - 3. Se uma aggregate é parte de outro serviço, não há problema referenciar um ID que está em um serviço separado (já que usa outro banco)

REGRAS DE AGGREGATES (REGRA #2)

- Esta abordagem também simplifica a persistência, visto que a aggregate é a unidade de armazenamento
- Isso também facilita o armazenamento em bancos NoSQL, como MongoDB
- Isso também elimina a necessidade de tratar lazy loading e seus problemas associados
- Escalar o banco de dados através da fragmentação de aggregates (em serviços diferentes) também é relativamente fácil

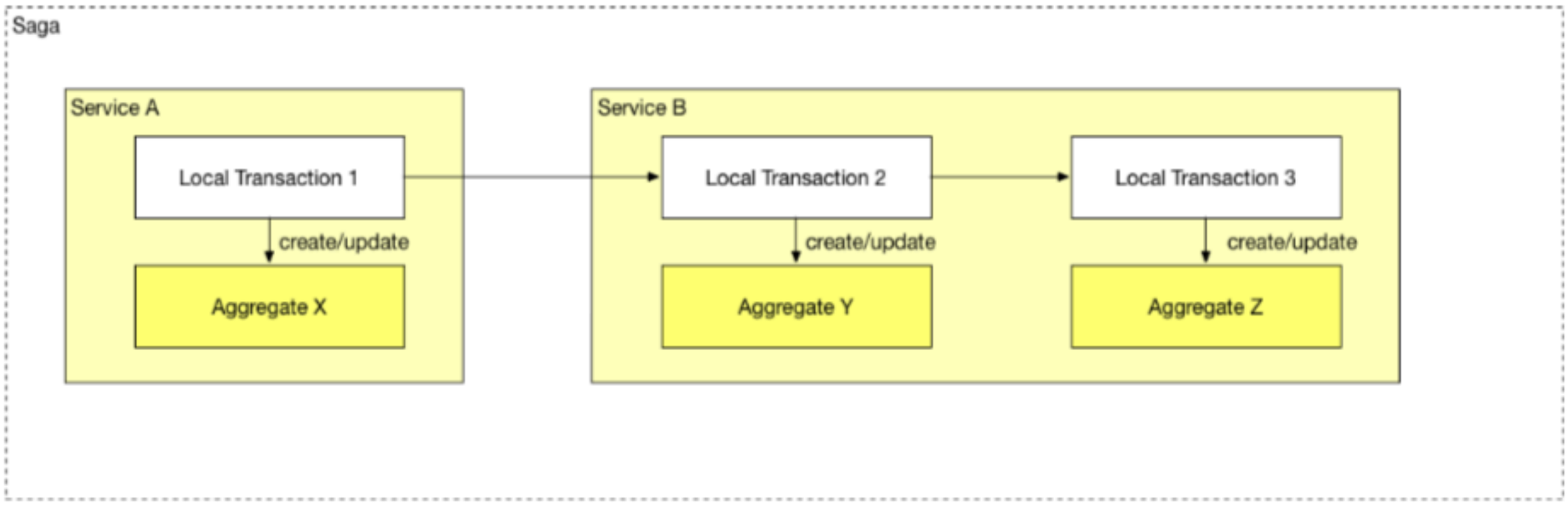
REGRAS DE AGGREGATES

- Regra #3: uma transação cria ou atualiza uma *aggregate*
  - Quando se usava uma aplicação monolítica, era normal atualizar várias aggregate numa mesma transação
- No entanto, usar uma transação por aggregate garante que uma transação está contida dentro de um serviço, o que se encaixa perfeitamente na arquitetura de microsserviços.

REGRAS DE AGGREGATES (REGRA #3)

- Essa regra é mais complicada de implementar. Mas o uso de sagas foi pensado para resolver esse problema.

- Cada etapa de uma saga cria ou atualiza exatamente um aggregate.

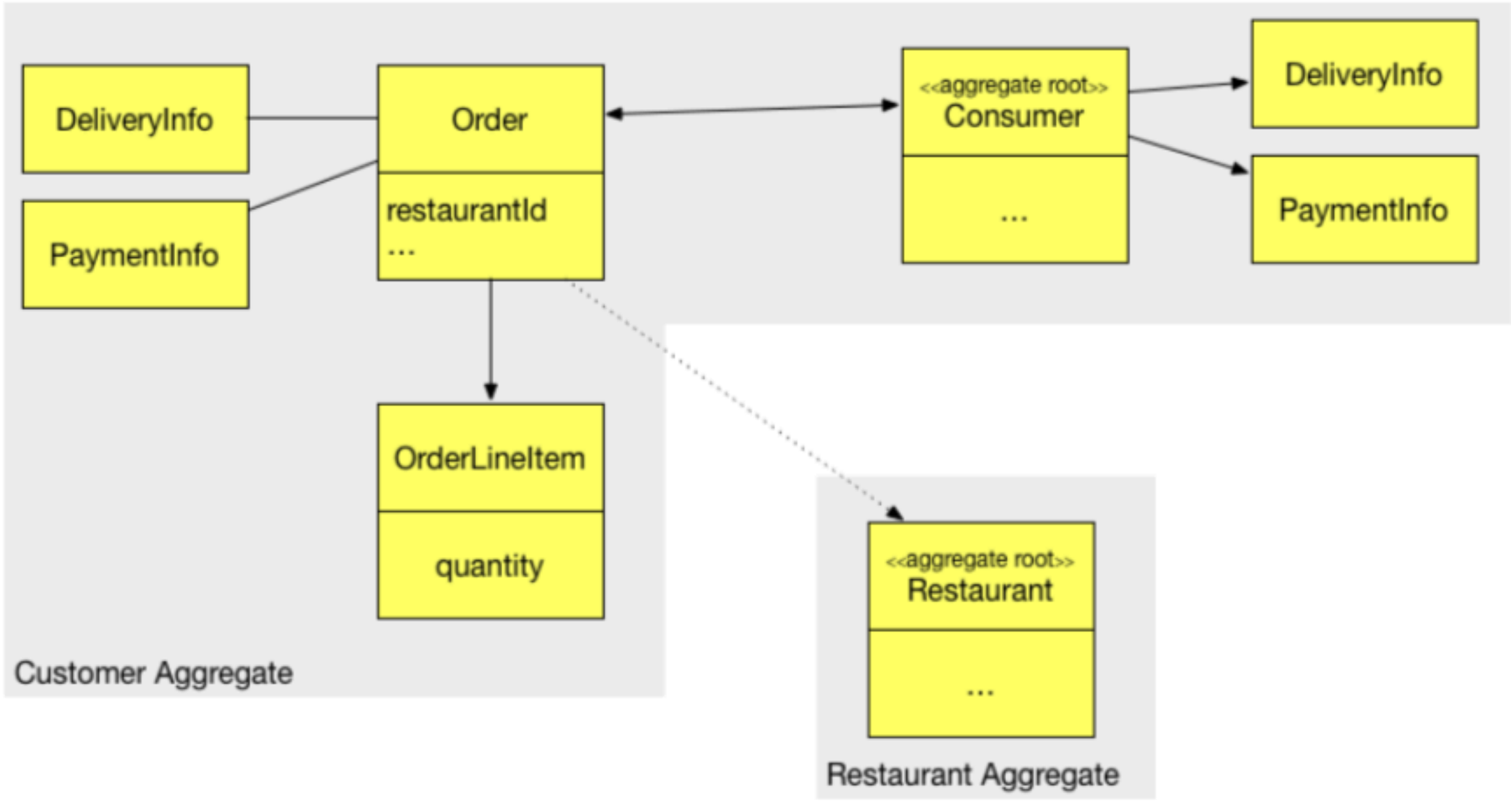


GRANULARIDADE DE UMA AGGREGATE

- Uma decisão chave é definir o quão larga será uma aggregate
- Uma aggregate deve ser pequena, visto que cada aggregate é "serializada".
- Aggregates menores melhoram a escalabilidade
- Por outro lado, como uma aggregate está no escopo de uma transação, é preciso que ela seja larga o suficiente para fazer uma atualização particular ser atômica

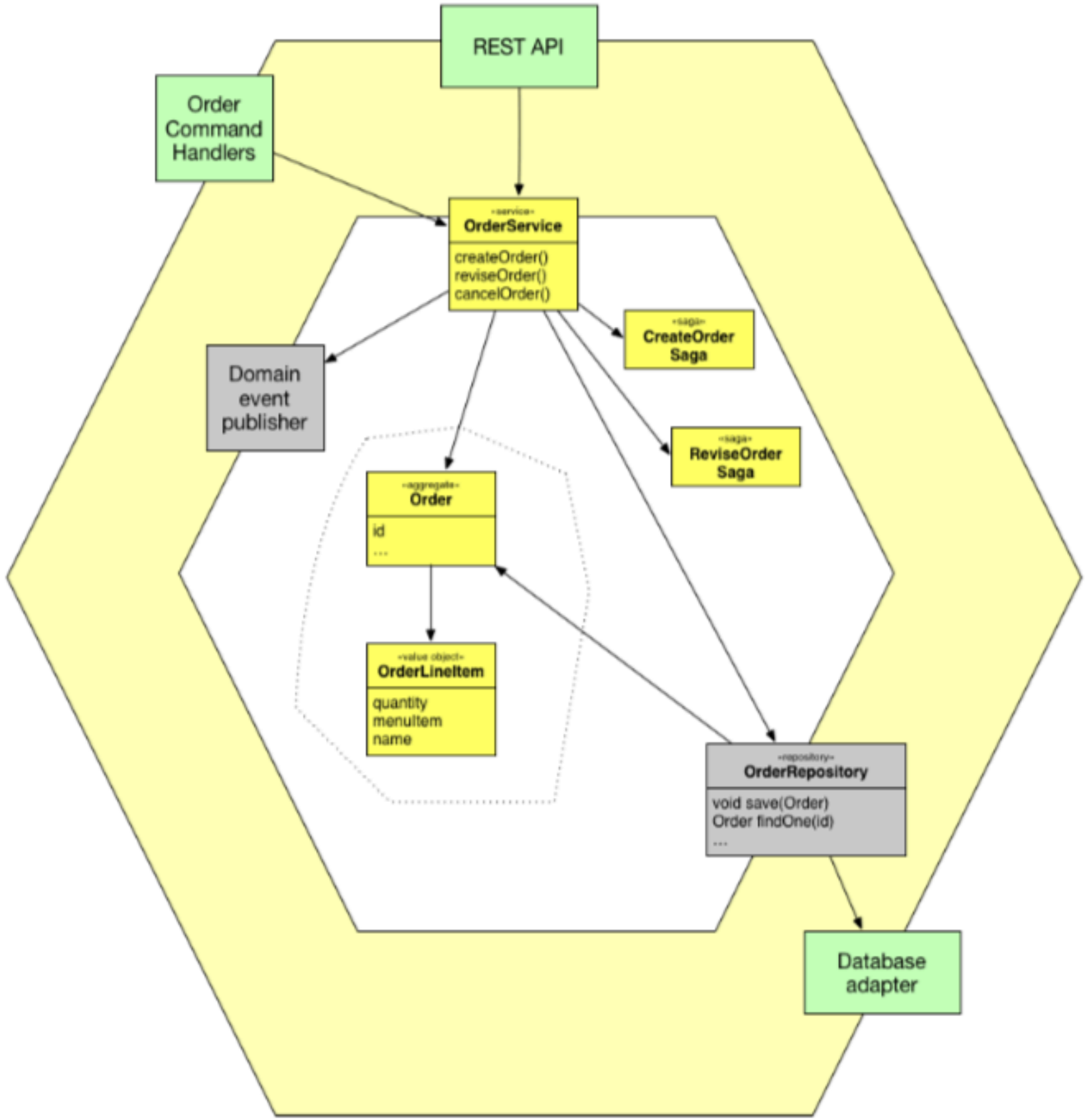
GRANULARIDADE DE AGGREGATES

- Alternativa fazendo a aggregate mais ampla



MODELANDO LÓGICA DE NEGÓCIO COM AGGREGATES

- Corpo da business logic consiste de aggregates



PUBLICANDO EVENTOS DE DOMÍNIO

PUBLICANDO EVENTOS DE DOMÍNIO

- Merriam-Webster lista várias definição da palavra “evento" including:
- a : something that happens : occurrence
  b : a noteworthy happening
  c : a social occasion or activity
  d : an adverse or damaging medical occurrence a heart attack or other cardiac event –
- Merriam-Webster https://www.merriam-webster.com/dictionary/event

PUBLICANDO EVENTOS DE DOMÍNIO

- No contexto de DDD, um evento de domínio é algo que acontece à uma aggregate
- É representado por uma classe no modelo de domínio
- Um evento geralmente representa uma mudança de estado
- Exemplo: Pedido Criado, Pedido Cancelado, Pedido Entregue.
- Um *aggregate* de Pedido deve, se há consumidores interessados, publicar um evento cada vez que uma transição de estado acontece.

AN AGGREGATE PUBLISHES A DOMAIN EVENT WHEN IT IS CREATED OR UNDERGOES SOME OTHER SIGNIFICANT CHANGE.

Padrão: Domain Event

POR QUE PUBLICAR EVENTOS?

- Porque outras partes - usuários, aplicações ou componentes da mesma aplicação - estão interessados em saber sobre as mudanças de estado da aggregate. Exemplos:
- Manter consistência de dados entre serviços usando sagas baseadas em coreografia
- Notificar um serviço que mantém uma réplica que o estado do dado mudou (conhecido como CQRS)
- Notificar uma aplicação diferente registrada via webhook ou message broker para disparar o próximo passo de um processo de negócio

POR QUE PUBLICAR EVENTOS?

- Notificar um componente diferente da mesma aplicação, para por exemplo, enviar uma mensagem para o browser do usuário ou atualizar um banco de busca.
- Enviar notificações - mensagens de texto ou e-mails - informando por ex., que um pedido foi entregue, que o avião atrasou.
- Monitorar eventos de domínio para verificar se o comportamento da aplicação está correto
- Analisar eventos para modelar o comportamento do usuário

O QUE É UM EVENTO DE DOMÍNIO

- Classe com nome no particípio passado. Cada propriedade pode ser um valor primitivo ou um value object. Ex.: classe PedidoCriado com propriedade idPedido
- Também pode ter metadados, como id e um timestamp.
- Alternativamente, os metadados podem ser um objeto "envelope", que encobrem o objeto do evento
- O id da aggregate que é emitido também pode ser parte do envelope, ao invés de ser uma propriedade do evento.

EXEMPLO DE EVENTO DE DOMÍNIO

```
interface DomainEvent {}

interface OrderDomainEvent extends DomainEvent {}

class OrderCreated implements OrderDomainEvent {}
```

```
class DomainEventEnvelope<T extends DomainEvent> {
private String aggregateType;
private Object aggregateId;
private T event;
... }
```

ENRIQUECIMENTO DE EVENTOS

- Suponha que você está escrevendo um consumidor de eventos para os eventos de processamento de pedidos
- O evento OrderCreated captura a essência do que aconteceu. No entanto, o consumidor precisa saber os detalhes do processamento do OrderService
- A desvantagem do consumidor fazer uma query para o serviço para buscar a aggregate é o overhead causado por cada requisição
- Uma abordagem alternativa é conter as informações que o consumidor precisa no evento. Isso simplifica o consumo, já que não demanda ao consumidor o carregamento dos dados a partir do serviço.

ENRIQUECIMENTO DE EVENTOS - EXEMPLO

```
class OrderCreated implements OrderEvent {
private List<OrderLineItem> lineItems;
private DeliveryInformation deliveryInformation;
private PaymentInformation paymentInformation;
private long restaurantId;
private String restaurantName;
...
}
```

ENRIQUECIMENTO DE EVENTOS

- Apesar do enriquecimento de eventos simplificar o consumo, a desvantagem é o risco de tornar os eventos menos estáveis.
- Mudanças em requisitos podem demandar mudanças nos eventos. Isto pode dificultar a manutenção.
- Além disso, mudança nos eventos teria impacto em todas as aplicações que consomem aquele evento.

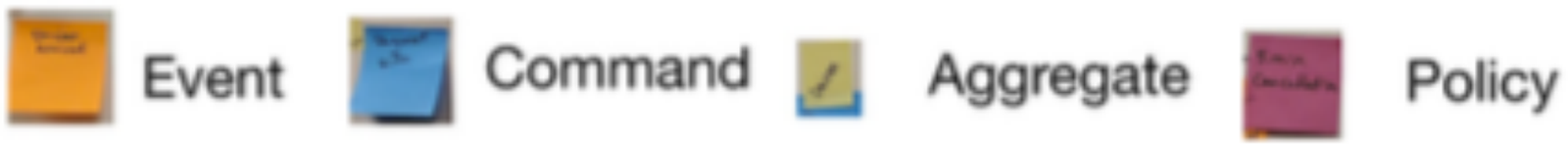
IDENTIFICANDO EVENTOS DE DOMÍNIO

- Geralmente os requisitos irão definir cenários em que notificações são exigidas. Ex.: "Quando X acontece, faça Y" ou "Quando um pedido é feito, envie um e-mail para o cliente".
- Outra abordagem é usar "*event storming*", que envolve reunir um grupo de especialistas para fazer um levantamento dos eventos de uma determinada aplicação

IDENTIFICANDO EVENTOS DE DOMÍNIO – EVENT STORMING

- Event Storming consiste de três passos principais:
  - 1. Brainstorm de eventos
  - 2. Identificar gatilhos de eventos (ações do usuário, sistema externo, outro evento, tempo)
  - 3. Identificar aggregates

IDENTIFICANDO EVENTOS DE DOMÍNIO – EVENT STORMING



GERANDO E PUBLICANDO EVENTOS DE DOMÍNIO

- Comunicação baseada em eventos é uma forma de comunicação assíncrona
- Exemplo prático: modelando, gerando e publicando eventos