Módulo II

7COMm
Serviços e Soluções em TI

Agenda

- Docker + Patterns de Regras de negócios
- Implementação DDD
  - Arquitetura Hexagonal
  - Eventos
- Padrões de APIs Externas
  - Api Gateway / BFF
  - Netflix Zuul
- CQRS
- Entregando software como SaaS
  - Twelve Factors + CI/CD

Regras de negócio em uma arquitetura de microsserviços + Docker

7COMm
Serviços e Soluções em TI

Agenda

- Docker
  - Conceitos básicos
  - Exemplos
- Patterns de Regras de Negócios
  - Transaction Script
  - Domain Model Pattern
  - DDD

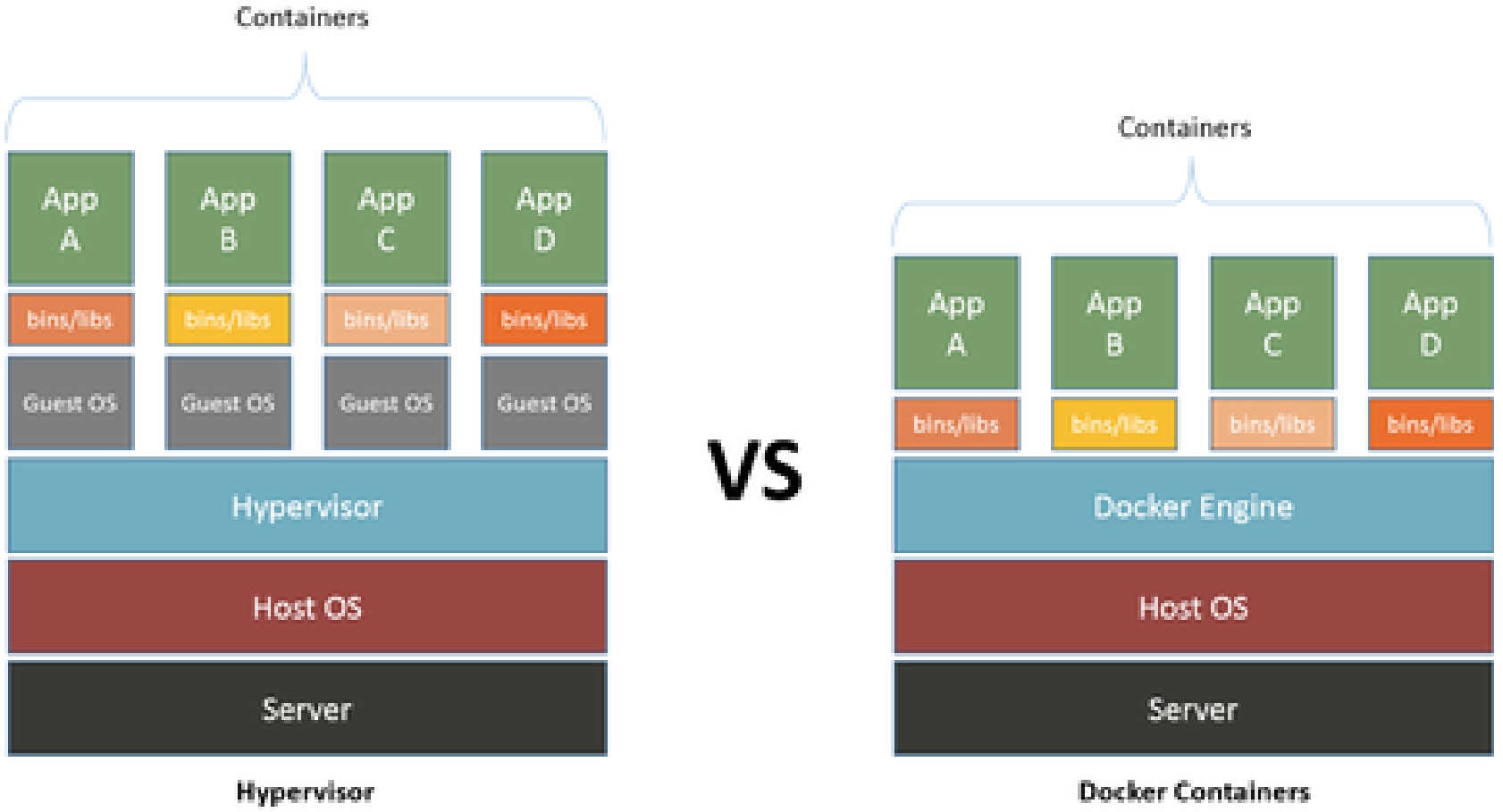
Docker – Conceitos Básicos

7COMm
Serviços e Soluções em TI

Docker - Conceitos Básicos

- Rodar Aplicações containerizadas baseadas em imagens
- Automatiza a implantação
- Torna comuns as configurações e dependências
- Compartilha o kernel do SO com a máquina HOST

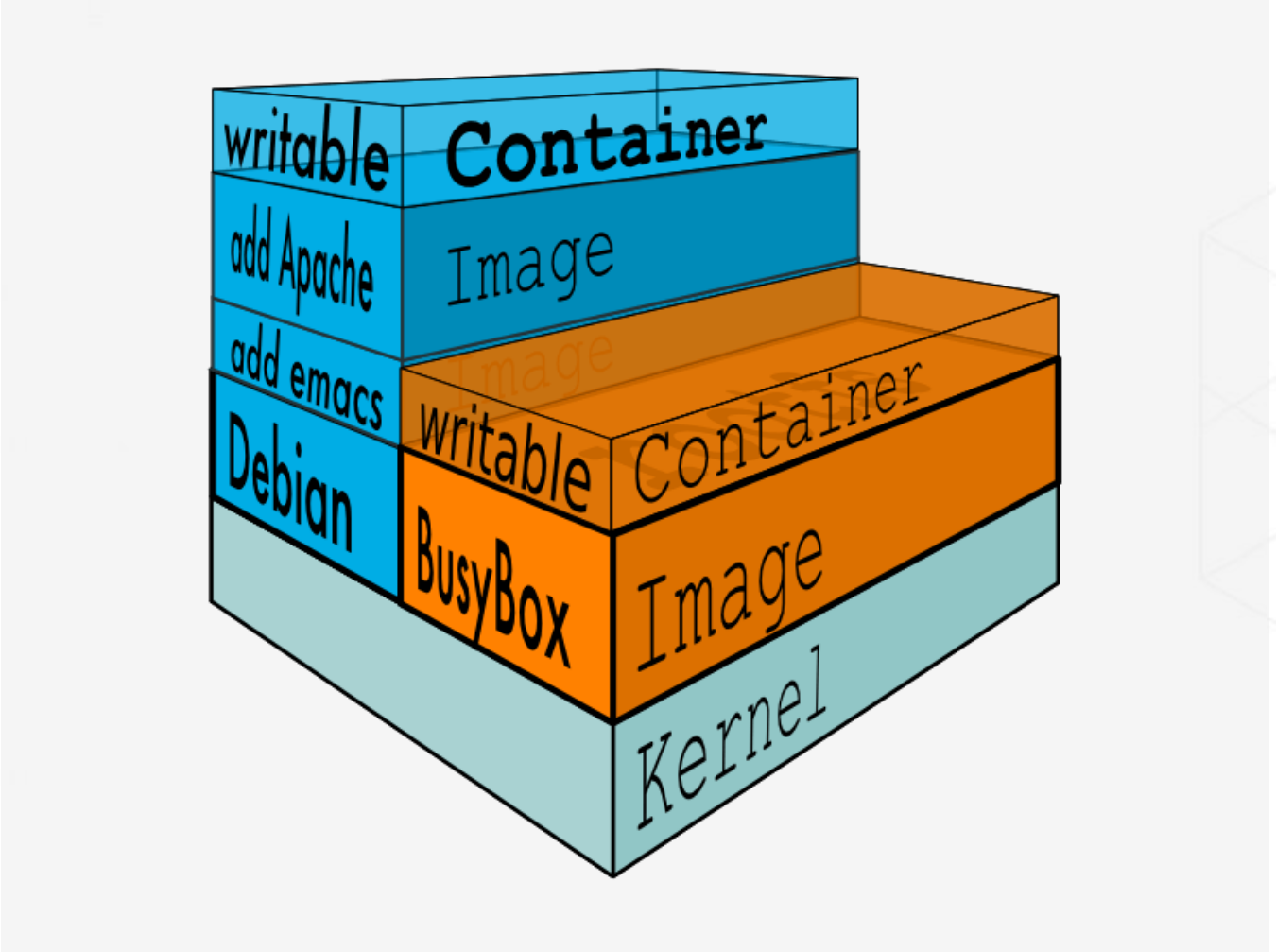
Docker - Conceitos Básicos



Docker - Imagens

- Camadas sobrepostas de elementos
- Fábrica de Containeres
- Somente leitura
- Disponibilizada através de *registries*
- *Processo* que o container precisa para rodar seu objetivo

Docker - Imagens



Docker - Containeres

- Espécie de *instancia* de uma imagem
- *Stateless*
- Ambiente de execução propriamente dito
- Roda em isolamento dentro do sistema operacional

Docker - Dockerfile

- Arquivo com instruções para geração da imagem
  - Cópias de Arquivos
  - Variáveis de Ambientes
  - Parâmetros de inicialização
  - ...

Docker - Dockerfile

Docker - Volumes

- Quaisquer dados salvos no container são perdidos quando este é parado ou excluído
- Para persistir informações é preciso criar uma estrutura externa, somente *vinculada* ao container
- Essa estrutura, podemos entender como volumes

Docker Compose

- Tecnologia para definição de vários containeres compartilhados
- Define através de um arquivo *yaml* quais containeres e suas respectivas configurações

Docker Compose

```yaml
wordpress:
    image: wordpress
    links:
     - mariadb:mysql
    environment:
     - WORDPRESS_DB_PASSWORD=123456
     - WORDPRESS_DB_USER=root
    ports:
     - "8082:80"
    volumes:
     - ./html:/var/www/html
mariadb:
    image: mariadb
    environment:
     - MYSQL_ROOT_PASSWORD=123456
     - MYSQL_DATABASE=wordpress
    volumes:
     - ./database:/var/lib/mysql
```

Regras de negócio em uma arquitetura de microsserviços

7COMm
Serviços e Soluções em TI

Regras de negócio

- Coração de toda aplicação.
- Regras complexas em uma arquitetura de microsserviços são desafiadoras.
- Espalhadas em vários serviços.

Business Logic Patterns

- Parte mais complexa do serviço.
- É comum utilizar orientação a objetos, mas não mandatório.
- Deve-se, idealmente, desacoplar todas as regras de negócio do código da infraestrutura.
- Os três principais padrões para desenvolver a lógica de negócios são: *transaction script pattern, domain model pattern e DDD*.

Transaction Script Pattern

- Pattern descrito por **Martin Fowler** em *Patterns of Enterprise Application Architecture.*
- https://martinfowler.com/eaaCatalog/transactionScript.html
- Uma maneira de tratar regras de negócio como um script procedural.
- É criado um método para manusear o *request*.
- Utiliza poucas *features* de programação orientada a objetos.

Transaction Script Pattern

- Ao invés de quaisquer códigos orientados a objeto, cria-se um método para cada *request*.
- Este método realiza a funcionalidade por completo.
- É comum a utilização de linguagens eficientes em **código procedural**.

Transaction Script Pattern – Formas de implementação

- Vários *Transactions Scripts* dentro de uma classe
- Uma classe para cada *transaction script*, com uma interface definindo o método de execução

Transaction Script Pattern – Formas de implementação

- Exemplos...

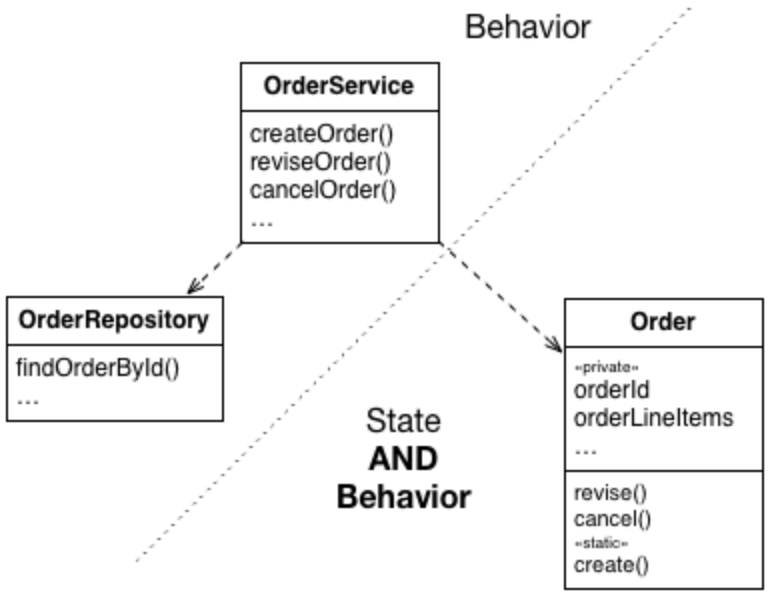
Transaction Script Pattern

- Para lógicas simples funciona bem.
- Para lógicas complexas, não é o ideal.
- Para lógicas que vão evoluir, deve ser repensado.
- https://java-design-patterns.com/patterns/transaction-script/

Domain Model Pattern

- Organização de serviços através de modelos que consistem em estado e comportamento.
- Utiliza os conceitos de **SOLID** e **orientação a objeto**.

Domain Model Pattern



Domain Model Pattern

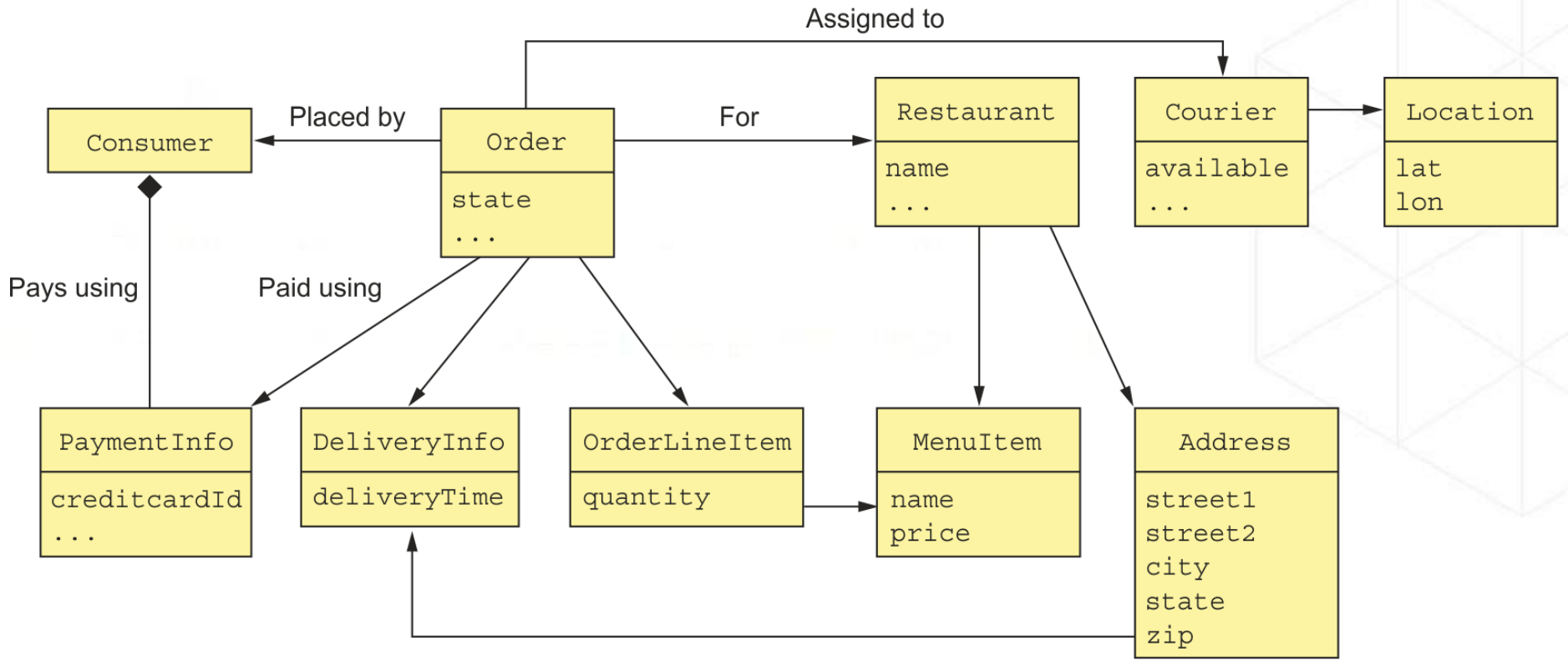
- Métodos mais simples.
- Tarefas delegadas a outros elementos.
- Classes mais numerosas, porém com menos responsabilidade.

Domain Model Pattern

- Desenho mais próximo do que vemos no mundo real.
- Fácil de testar.
- Pode-se aplicar um grande número de *patterns*, principalmente para escalar a funcionalidade.

Domain Model Pattern

Figure 5.4. A traditional domain model is a web of interconnected classes. It doesn't explicitly specify the boundaries of business objects, such as `Consumer` and `Order`.



Domain Model Pattern

- Apesar de termos uma visão do todo, temos gaps
  - Não é especificado quais classes fazem parte do *negócio* de Order
  - Não possuem **fronteiras explícitas** entre os negócios
  - **Interdependência** é grande
  - Impacto de mudança pode ser enorme

Domain Model Pattern

- https://java-design-patterns.com/patterns/domain-model/
- https://martinfowler.com/eaaCatalog/domainModel.html

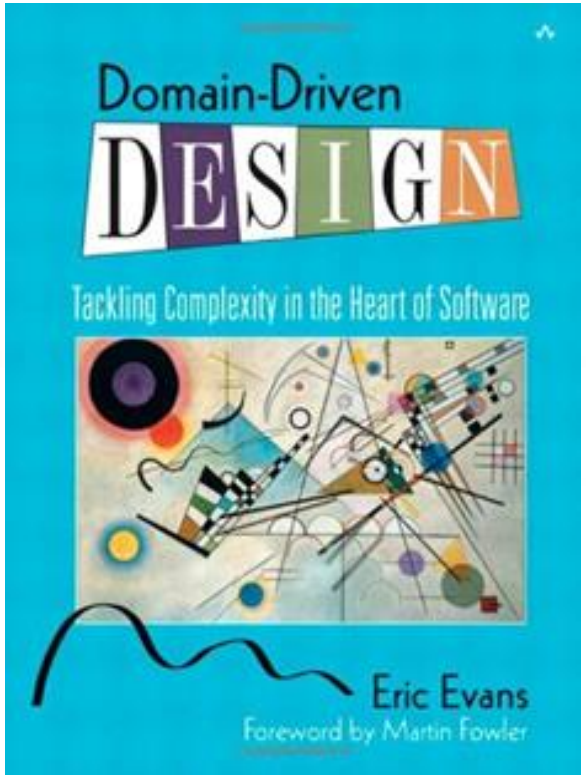
Contextos transacionais (Boundered Contexts)

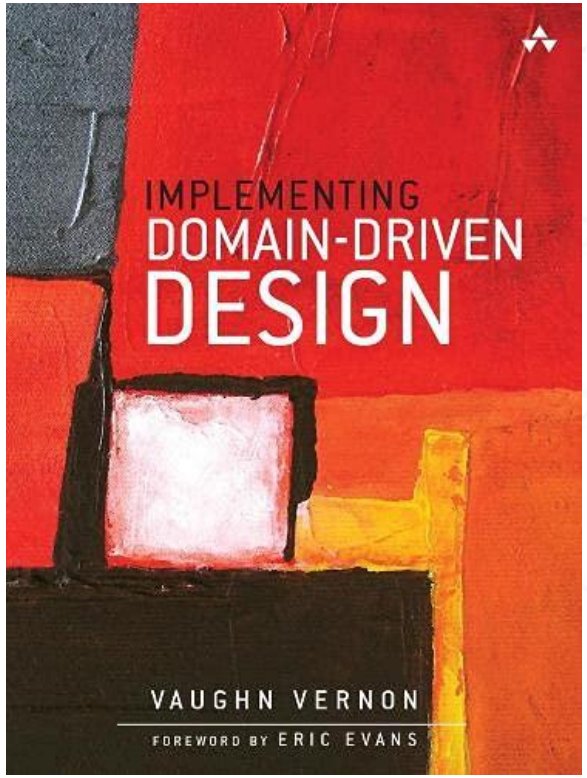
- Devemos excluir um pedido da base
  - Qual é, de fato, o escopo dessa exclusão?
- Uma Regra de negócio deve ser desenhada para ter uma lógica **invariante**, que SEMPRE deve ser aplicada
- Caso não seja aplicada, mais de um *request* pode tornar o objeto inconsistente

Domain Driven Design (DDD)

- Criado em 2003, por Eric Evans.
- Refinamento do **DOO** aplicado a regras de negócio complexas.
- Cada módulo tem seu próprio modelo de domínio.
- Base de alguns dos *patterns* mais adotados no mercado.
- Design centrado em conceitos de domínio de negócios.

Domain Driven Design (DDD)





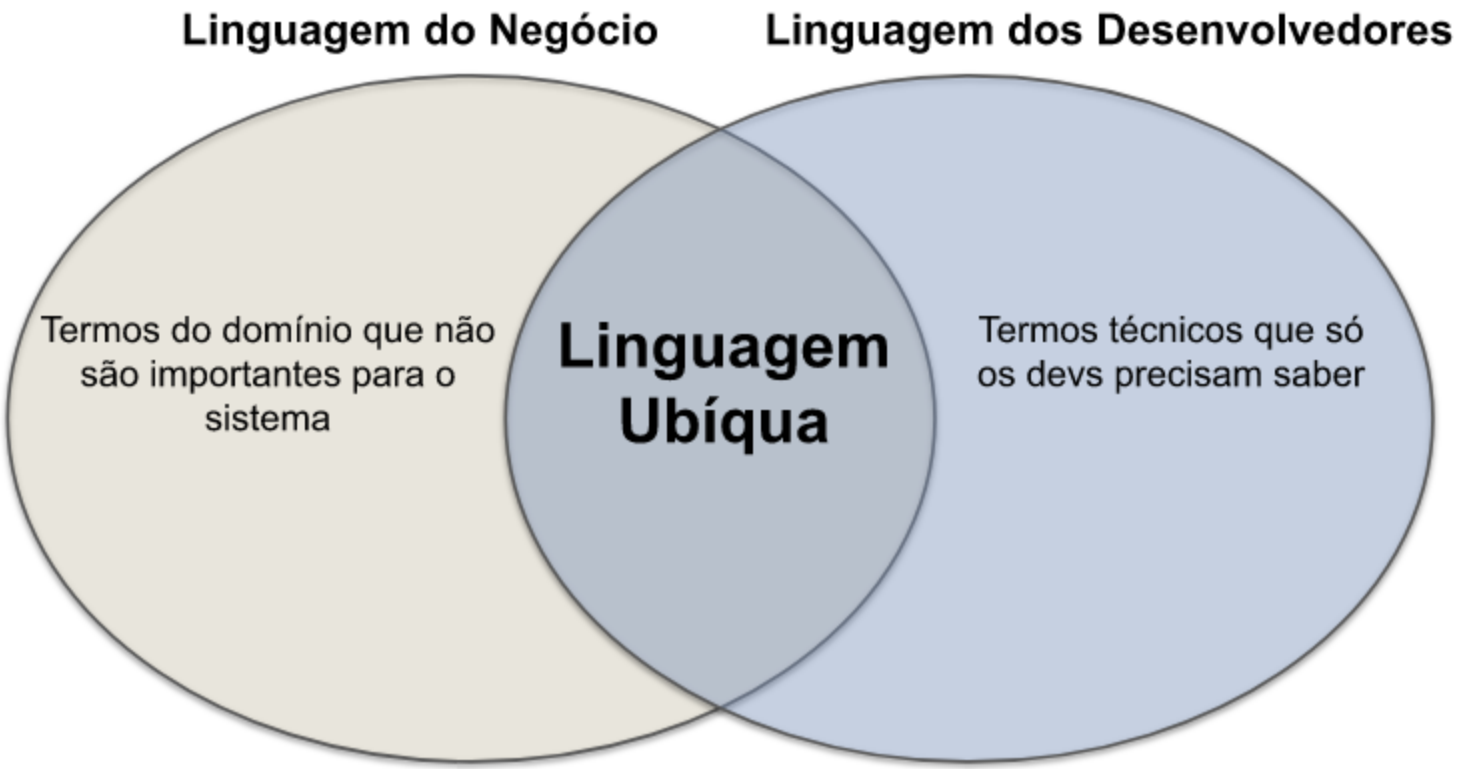
DDD

- Desenvolvedores devem ter profundo conhecimento do domínio.
- Defende a separação entre *domínio* e tecnologias.
- Diferentes estilos arquiteturais podem ser utilizados.

Linguagem Ubíqua

- Conceito Central no DDD.
- Termos entendidos perfeitamente entre experts do domínio e desenvolvedores.
- A definição desses termos fará toda diferença no processo de desenvolvimento e comunicação na aplicação.

Linguagem Ubíqua



Patterns do DDD

- Entity.
- Value Object.
- Factory.
- Repository.
- Service.

Entities

- Objeto definido por uma **identidade única**.
- Seu estado não é fundamental para sua identificação.
- Entidades deve projetar com cuidado sua persistência.
- Devem existir regras que regem a criação/remoção.
    - Ex. Não remover um usuário se ele tem uma entrega pendente.

Value Objects

- Não possuem identificador único.
- Objetos simples.
- Caracterizados *apenas* por seu estado.
- ***Devem* ser imutáveis**.
  - Ex. Para alterar o endereço de um usuário deve-se excluir o antigo e adicionar o novo.
- Possuem tipagem forte ao invés da utilização de dados primitivos.
- https://martinfowler.com/bliki/ValueObject.html

Services

- Operações importantes não caracterizadas como *Entity* ou *VO*.
- Stateless.
- Pode incluir tanto *Entity* quanto *VO* em suas assinaturas.
- É comum sua implementação com *singletons*.

Repositories

- Abstração para recuperar objetos do **domínio de dados**.
- Normalmente utilizados pra recuperar **entities** ou **agregados** (a seguir).
- Também pode consultar diretamente serviços externos.

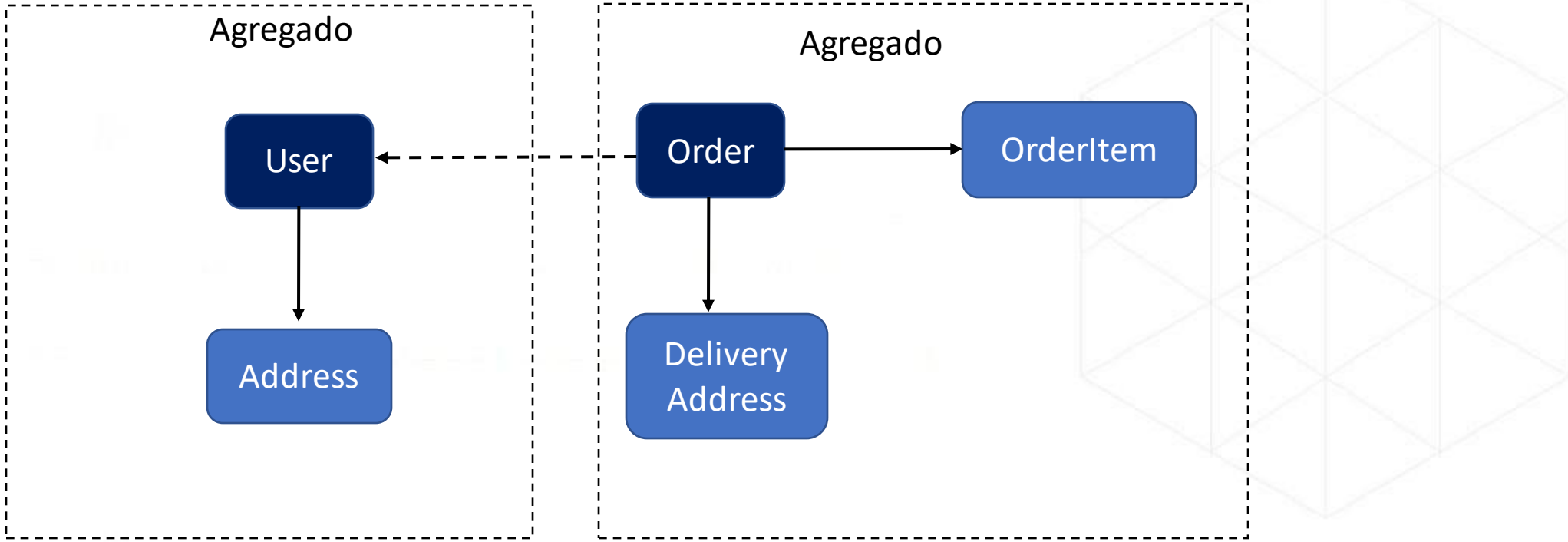
Repository vs DAO

- DAO
  - Abstração de persistência de dados
  - Mais próximo do banco de dados
  - Centrado na tabela
- Repository
  - Coleção de objetos
  - Lida com agregados (a seguir)
  - Interface mais sucinta

Aggregate

- *Pattern* para redução de complexidade técnica
- Cluster de objetos tratados como uma única unidade
- Representa um conceito de domínio

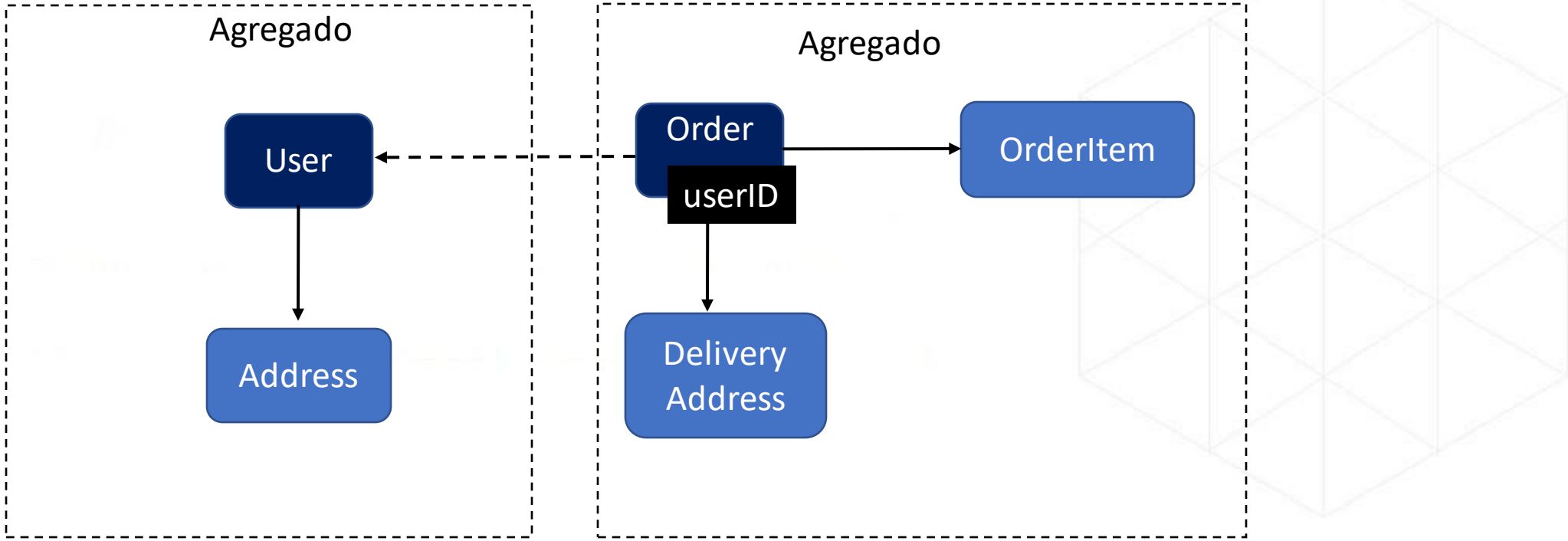
Aggregate


Agregado
User
Address
Agregado
Order
OrderItem
Delivery Address

Aggregate – Elemento Raiz

- Somente este elemento deve ser referenciado
- É identificado através de uma chave.
- Quaisquer elementos internos do agregado devem ser alterados **através** do elemento raiz.
- Coordena todas as mudanças de estado do agregado

Aggregate


Agregado
User
Address
Agregado
Order
userID
OrderItem
Delivery Address

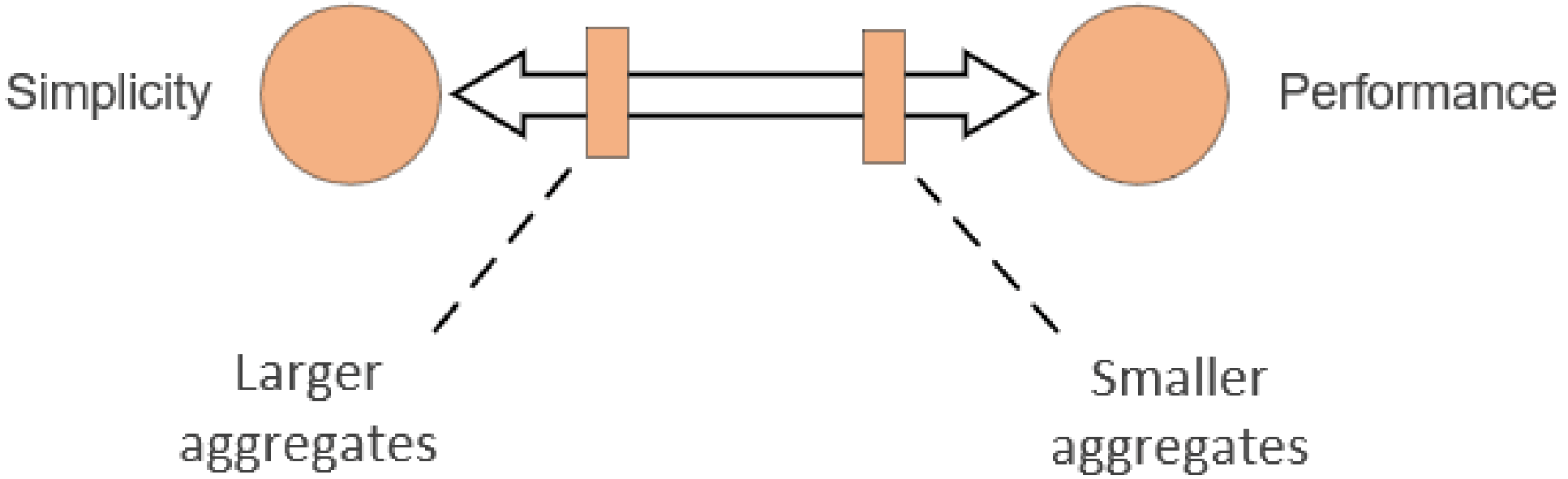
Referências a outros agregados

- **Jamais** devem ser através de referência de objetos.
- Devem ser através das chaves, como em uma *primary key*.
- *Foreign key* não é um *code smell*, faz parte do processo.
- Facilita a persistência de dados em bases **NoSQL**.

Agregados - Criação e atualização

- Uma, e somente uma, transação deve criar ou atualizar o agregado.
- Não deve ser atualizado mais de um agregado ao mesmo tempo.
- Deve-se manter os agregados no menor tamanho possível.

Agregados - Granularidade



Agregados - Implementação

- As implementações com ORM podem ser problemáticas
- Pode não ser trivial utilizar bancos relacionais
- Um banco de documentos pode ser considerado

Agregados - Implementação

```java
class Order {
    private final List<OrderLine> orderLines;
    private Money totalCost;

    Order(List<OrderLine> orderLines) {
        checkNotNull(orderLines);
        if (orderLines.isEmpty()) {
            throw new IllegalArgumentException("Order must have at least one order line item");
        }
        this.orderLines = new ArrayList<>(orderLines);
        totalCost = calculateTotalCost();
    }

    void addLineItem(OrderLine orderLine) {
        checkNotNull(orderLine);
        orderLines.add(orderLine);
        totalCost = totalCost.plus(orderLine.cost());
    }

    void removeLineItem(int line) {
        OrderLine removedLine = orderLines.remove(line);
        totalCost = totalCost.minus(removedLine.cost());
    }

    Money totalCost() {
        return totalCost;
    }

    // ...
}
```

Agregados - Implementação

```java
@Aggregate
public class ProductAggregate {

    @AggregateIdentifier
    private String productId;
    private String name;
    private BigDecimal price;
    private Integer quantity;

    @CommandHandler
    public ProductAggregate(CreateProductCommand createProductCommand) {
        //Você pode fazer toda validação aqui
        ProductCreatedEvent productCreatedEvent = new ProductCreatedEvent();
        BeanUtils.copyProperties(createProductCommand, productCreatedEvent);

        AggregateLifecycle.apply(productCreatedEvent);
    }

    public ProductAggregate() {
    }


    @EventSourcingHandler
    public void on(ProductCreatedEvent productCreatedEvent) {
       this.quantity = productCreatedEvent.getQuantity();
        this.productId = productCreatedEvent.getProductId();
        this.price = productCreatedEvent.getPrice();
        this.name = productCreatedEvent.getName();
    }
}
```

Próximos Passos

- Implementação do OrderService
- Eventos de Domínio
- Event Sourcing

Para saber mais

- https://docs.docker.com/desktop/windows/install/
- https://martinfowler.com/eaaCatalog/transactionScript.html
- https://java-design-patterns.com/patterns/transaction-script/
- https://java-design-patterns.com/patterns/domain-model/
- https://martinfowler.com/eaaCatalog/domainModel.html
- https://martinfowler.com/bliki/ValueObject.html
- https://www.youtube.com/watch?v=1AEOcQWQR2o

OBRIGADO!

Centro
Rua Formosa, 367 - 29º andar Centro, São Paulo - SP, 01049-000

Alphaville
Avenida Ipanema, 165 - Conj. 113/114 Alphaville, São Paulo - SP,06472-002

+55 (11) 3358-7700
contact@7comm.com.br

7COMm
Serviços e Soluções em TI