

Domain-Driven Development

O **Domain-Driven Development (DDD)** é uma abordagem de design de software que coloca o **domínio do problema** no centro do desenvolvimento. Em vez de começar pelo banco de dados ou pela tecnologia, o foco está em entender profundamente as **regras de negócio** e modelá-las diretamente no código.

Criado por **Eric Evans** no livro ***Domain-Driven Design: Tackling Complexity in the Heart of Software*** (2003), o DDD visa alinhar desenvolvedores e especialistas do negócio, usando uma linguagem comum (**Ubiquitous Language**) que aparece tanto nas conversas quanto no código.

“É um conjunto de princípios com foco em domínio, exploração de modelos de formas criativas e definir e falar a linguagem Ubíqua, baseado no contexto delimitado.”

Antes de percorrermos com mais detalhes os principais pontos do DDD, é fundamental que consigamos entender de forma clara seus três pilares e como os mesmos têm relação entre si.

Domínio é o coração do negócio em que você está trabalhando. É baseado em um conjunto de ideias, conhecimento e processos de negócio. É a razão do negócio existir. Sem o domínio todo o sistema, todos os processos auxiliares, não servirão para nada.

Se uma empresa existe, é porque ela tem um core business e, normalmente, esse core business é composto pelo domínio principal.

Quando falamos em DDD – Domain Driven Design, não falamos apenas em desenvolver um software, mas sim em entender a modelagem do projeto como um todo. Se você não souber modelar o software, não conseguirá fazê-lo crescer e ser mantido a médio e longo prazo.

O DDD preza que os desenvolvedores façam parte do processo, entendendo o negócio e todos os seus modelos nos diferentes ângulos e não somente participando de reuniões com especialistas.

Fazendo um relacionamento com o mundo dos serviços, o novo perfil dos desenvolvedores faz com que o time participe de todo o processo, desde o levantamento de requisitos até o contato com o Domain Expert. Antigamente, o desenvolvedor apenas codificava.

Como exemplo, podemos citar um sistema de imobiliária. Você precisa conhecer toda a terminologia, como: pé direito, m², etc. Para isso é necessário sentar com o Domain Expert para entender o funcionamento do negócio e evitar problemas de comunicação.

Suponha que você irá desenvolver um ERP, um sistema bem grande, que tenha diversas áreas e contextos. Quando esse sistema começa a crescer, pode ficar impossível mantê-lo, não necessariamente devido ao código, mas sim pelo fato de não saber mais a relação de uma entidade com a outra, de um ponto com o outro e suas relações de negócio. Neste caso, o DDD ajudará no processo.

O Eric fala que o DDD é totalmente mutável e que pode se desenvolver conforme o tempo. Atualmente, no mundo dos microserviços está evidente o “boom” do DDD, e porque DDD e micro serviços fazem total sentido.

Normalmente o DDD é utilizado para aplicações complexas e é muito fácil de entender. Certamente você achará fácil e simples de entender, porém muito difícil de aplicar.

Podemos dizer que a grande sacada do DDD é que, na área técnica, ele utiliza vários padrões de projetos.

Preste atenção ao fato que, não é porque você utilizou alguns padrões em seu projeto, realizou algumas implementações, significa que você está utilizando DDD. Este é um erro comum entre as pessoas.

O livro do Eric Evans apresenta principalmente os conceitos DDD, porém, muitas pessoas o compram esperando aprender a codificar, com exemplos de código, etc; mas o livro apresenta os conceitos.

O DDD possui três pilares: **linguagem ubíqua**, **bounded contexts** e **context maps**. Se você entender esses três principais pontos, terá uma base conceitual para começar a trabalhar com DDD. Entender esses pontos, significa compreender o propósito do DDD, não significa que você irá codificar.

Resumindo, sempre que se falar em domínio, estaremos falando da razão daquele software existir. Sem aquele ponto principal, não há razão para o desenvolvimento do software.

Princípios Fundamentais do DDD

1. Ubiquitous Language (Linguagem Ubíqua)

- Usar os mesmos termos no código e na comunicação com especialistas do domínio;
- Exemplo: se o negócio fala em *Pedido* e *Cliente*, não usar *OrderEntity* ou *Usr*, mas sim *Pedido* e *Cliente*.

2. Bounded Context (Contexto Delimitado)

- Cada parte do domínio tem limites claros;
- Evita que o mesmo conceito tenha significados diferentes em lugares distintos do sistema.

3. Entidades

- Objetos com identidade única (ex.: Cliente com CPF).

4. Value Objects (Objetos de Valor)

- Objetos imutáveis que não têm identidade, mas sim valor (ex.: Endereço, Dinheiro).

5. Agregados e Raízes de Agregado

- Estruturas que agrupam entidades e objetos de valor, garantindo consistência;
- Só a raiz pode ser acessada diretamente.

6. Repositórios

- Objetos que simulam coleções de entidades/agregados, cuidando da persistência.

7. Serviços de Domínio

- Usados quando uma operação não cabe naturalmente em uma entidade ou objeto de valor.

8. Camada de Aplicação

- Orquestra o fluxo entre domínio e infraestrutura, sem conter lógica de negócio.

Benefícios

- Maior alinhamento entre código e negócio;
- Redução de ambiguidade;
- Melhor manutenção em sistemas complexos;
- Facilita evolução contínua.

Desvantagens

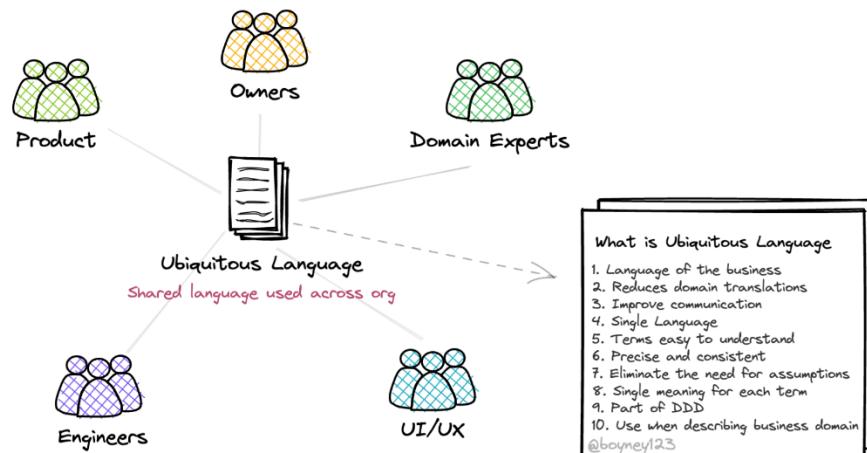
- Curva de aprendizado alta;
- Requer colaboração constante entre devs e especialistas do domínio;
- Pode ser pesado demais para sistemas simples.

1. Ubiquitous Language (Linguagem Ubíqua)

Conceito: Usar os mesmos termos do negócio tanto no código quanto na comunicação. A palavra ubíqua vem do latim ubique, que significa “em toda parte”. Algo ubíquo é aquilo que está presente em todos os lugares ao mesmo tempo.

No contexto do Domain-Driven Design (DDD), isso significa que a linguagem do domínio deve ser comum, clara e presente em todos os pontos do desenvolvimento: nas conversas entre desenvolvedores e especialistas do negócio, na documentação, e principalmente no código.

É o vocabulário compartilhado entre desenvolvedores e especialistas do domínio. Deve ser usado sem variação em código, reuniões, diagramas e documentação. Garante que todos entendam o mesmo conceito da mesma forma.



```
// O negócio fala em Cliente e Pedido, mas o dev usa nomes confusos
```

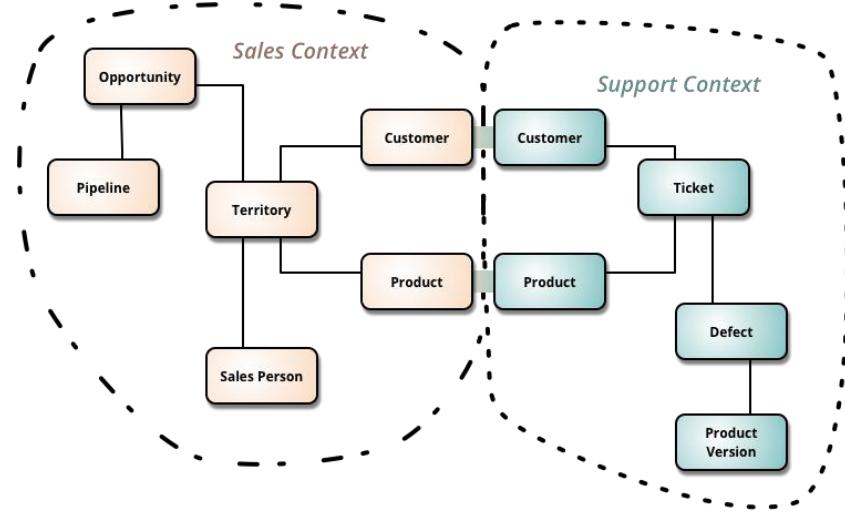
```
class UserData {  
    private String nm; // deveria ser nome  
    private int age; // deveria ser idade  
}
```

```
// Código e negócio usam os mesmos termos
```

```
class Cliente {  
    private String nome;  
    private int idade;  
    public Cliente(String nome, int idade) {  
        this.nome = nome;  
        this.idade = idade;  
    }  
}
```

2. Bounded Context (Contexto Delimitado)

Conceito: Um conceito pode ter significados diferentes em contextos distintos. Define **limites claros de responsabilidade** em um domínio. Isso lembra o **Single Responsibility Principle (SRP)** do SOLID, que diz que cada classe/módulo deve ter **uma única razão para mudar**.



Na figura abaixo, em outro exemplo, temos também dois contextos distintos:

- Contexto de Reservas (lado vermelho): Trabalha com comodidades, reservas, endereços. Usa Cliente e Apartamento dentro da lógica de reservar um espaço;
- Contexto de Avaliações (lado verde): Trabalha com comentários, avaliações, classificações. Também usa Cliente e Apartamento, mas dentro do sentido de avaliação.



O Bounded Context é um limite de significado dentro do domínio. Dentro de cada contexto, as palavras Cliente e Apartamento significam coisas diferentes.

No Contexto de Reservas, “Cliente” tem atributos como dataDaReserva, formaDePagamento. No Contexto de Avaliações, “Cliente” tem atributos como comentario, notaAtribuida. Assim, não existe confusão: “Cliente” no contexto de reservas não é o mesmo que “Cliente” no contexto de avaliações. Cada contexto tem seu modelo próprio, adaptado à sua necessidade.

“O Bounded Context evita que uma mesma palavra tenha **significados diferentes misturados** no código. Cada contexto tem sua própria versão dos objetos, de acordo com a necessidade daquele domínio.”

```
// "Produto" usado em dois contextos diferentes sem separar responsabilidade
class Produto {
    private String nome;
    private double preco;
    private int quantidadeEstoque; // Estoque
    private String descricaoCatalogo; // Catálogo
}
```

```
// Separação clara de contexto
```

```
class ProdutoEstoque {
    private String nome;
    private int quantidade;
}

class ProdutoCatalogo {
    private String nome;
    private String descricao;
    private double preco;
}
```

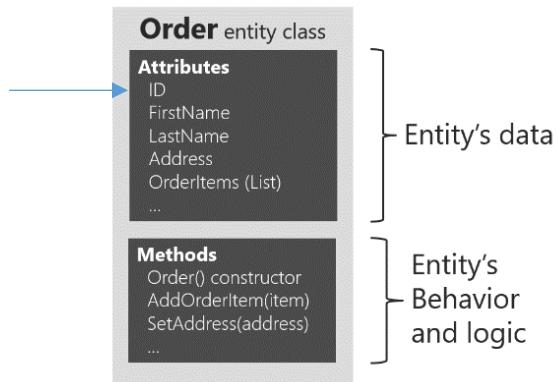
Ao separar contextos (ex.: *ProdutoEstoque* ≠ *ProdutoCatalogo*), você **reduz o acoplamento** entre áreas distintas do sistema. Isso conversa com o **Interface Segregation Principle (ISP)**, que evita interfaces genéricas que forçam dependências desnecessárias. Também ajuda no **Open/Closed Principle (OCP)**, pois cada contexto pode evoluir independentemente, sem quebrar o outro.

3. Entidades

Conceito: Objetos com **identidade única** (não apenas atributos). Possui identidade única (mesmo que seus atributos mudem). Essa identidade geralmente é representada por um identificador (ex.: CPF, CNPJ, matrícula, ID gerado). A Entidade é definida pela sua continuidade e identidade.

Exemplo da vida real: Um cliente chamado “Maria” pode mudar de endereço ou idade, mas ainda é o mesmo cliente, identificado pelo seu CPF.

Domain Entity pattern



```
// Dois clientes iguais se forem comparados só por nome e idade
```

```
class Cliente {  
    String nome;  
    int idade;  
}
```

```
// Cliente é identificado por CPF (identidade única)
```

```
class Cliente {  
    private String cpf; //Ainda assim, pensar no Object Calistenics  
    private String nome; //Ainda assim, pensar no Object Calistenics  
    private int idade; //Ainda assim, pensar no Object Calistenics  
    public Cliente(String cpf, String nome, int idade) {  
        this.cpf = cpf;  
        this.nome = nome;  
        this.idade = idade;  
    }  
}
```

```
@Override  
public boolean equals(Object o) {  
    if (!(o instanceof Cliente)) return false;  
    Cliente c = (Cliente) o;  
    return this.cpf.equals(c.cpf);  
}  
  
@Override  
public int hashCode() {  
    return cpf.hashCode();  
}  
}
```

O atributo `cpf` representa a identidade do cliente. Mesmo que nome ou idade mudem, o objeto continua sendo o mesmo cliente. A comparação de objetos (`equals` e `hashCode`) se baseia na identidade, não nos demais atributos.

4. Value Objects (Objetos de Valor)

Conceito: Sem identidade, apenas valor. São **imutáveis**. Um Value Object (VO) é um objeto do domínio que:

- Não possui identidade própria → é definido somente por seus atributos;
- É imutável → uma vez criado, não deve mudar de estado;
- Se dois objetos têm os mesmos valores nos atributos, eles são iguais;
- Geralmente representam conceitos descritivos do domínio, como:
 - Endereço;
 - Dinheiro;
 - Período de tempo;
 - Coordenada geográfica.

Exemplo da vida real: Dois endereços iguais em “Rua A, nº 10” são considerados o mesmo endereço, não importa em qual objeto estejam.

```
// Endereço mutável
```

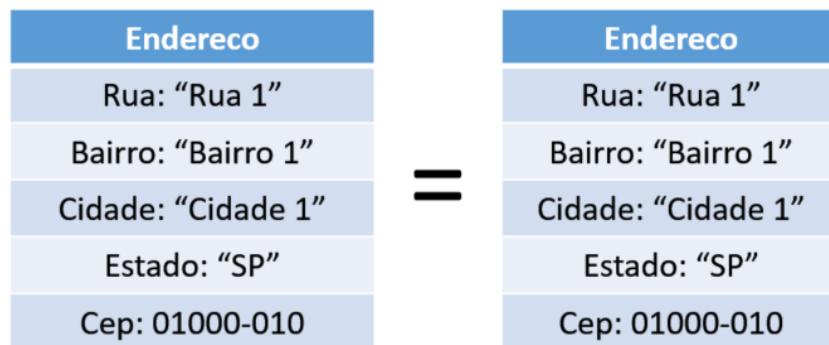
```
class Endereco {  
    String rua;  
    String cidade;  
}
```

Não é imutável : atributos podem ser alterados. Igualdade depende de referência de memória (==), não dos valores.

Observe que se fizermos uma string, o endereço é aquele e ponto. Se criarmos um objeto de valor onde a variável endereço é do tipo endereço, o endereço agora pode ter propriedades como: rua, número, bairro, cidade. Quando trabalhamos com esse tipo de objeto, conseguimos tipificar, melhorando as relações entre as entidades e também é possível validar as informações que são inseridas e garantir que estão no formato correto.

“Um Value Object não tem identidade própria. Se dois valores são iguais, o objeto é o mesmo. Ele existe para **descrever algo**, não para representá-lo como único.”

“Um **Value Object** é como uma etiqueta: se duas etiquetas têm os mesmos dados, elas são iguais. Não importa qual objeto criou, importa apenas o **valor**.”



```

// Endereço como Objeto de Valor (imutável)

final class Endereco {

    private final String rua;
    private final String cidade;

    public Endereco(String rua, String cidade) {
        this.rua = rua;
        this.cidade = cidade;
    }

    public String getRua() { return rua; }
    public String getCidade() { return cidade; }

    // Igualdade baseada nos valores, não na referência
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof Endereco)) return false;
        Endereco e = (Endereco) o;
        return rua.equals(e.rua) && cidade.equals(e.cidade);
    }

    @Override
    public int hashCode() {
        return rua.hashCode() + cidade.hashCode();
    }
}

```

Imutabilidade conversa com o SRP (Single Responsibility Principle). O VO só descreve valores, não carrega lógica adicional. Evita efeitos colaterais quando compartilhado entre objetos. Alinhado ao LSP (Liskov Substitution Principle), pois objetos imutáveis podem ser trocados com segurança.

Entidade x Value Object

```
// Entidade Cliente (identidade pelo CPF)

class Cliente {

    private final String cpf;

    private String nome;

    private Endereco endereco; // Value Object

    public Cliente(String cpf, String nome, Endereco endereco) {

        this.cpf = cpf;

        this.nome = nome;

        this.endereco = endereco;

    }

    public Endereco getEndereco() { return endereco; }

}
```

```
// Value Object Endereco

final class Endereco {

    private final String rua;      private final String cidade;

    public Endereco(String rua, String cidade) {

        this.rua = rua;

        this.cidade = cidade;

    }

    @Override

    public boolean equals(Object o) {

        if (!(o instanceof Endereco)) return false;

        Endereco e = (Endereco) o;

        return rua.equals(e.rua) && cidade.equals(e.cidade);

    }

    @Override

    public int hashCode() {

        return rua.hashCode() + cidade.hashCode();

    }

}
```

O Cliente é uma Entidade : identificado pelo CPF. Para serem iguais, precisam ter o mesmo CPF.

O Endereço é um Value Object : se dois endereços forem iguais, eles são o mesmo valor. Não levado em consideração o valor de apontamento de suas referências.

Métodos equals, hash code e toString

Existem 3 **métodos fundamentais em Java** que são essenciais para **diferenciar Entidade de Value Object**:

- equals()
- hashCode()
- toString()

Eles são a **base prática** para implementar corretamente o princípio de **Entidade e Value Object** no DDD.

1. equals(Object o)

- Serve para **comparar dois objetos**;
- Por padrão (na classe Object), compara apenas **referências na memória**;
- Em Entidades e Value Objects, **sobrescrevemos** para comparar pelo que importa:
 - **Entidade:** compara **identidade** (ex.: CPF, matrícula, ID).
 - **Value Object:** compara **atributos** (todos os valores).

2. hashCode()

- Usado em coleções como HashSet e HashMap;
- Sempre que sobrescrevemos equals, precisamos sobrescrever hashCode;
- Garantia: se dois objetos são **iguais pelo equals**, devem ter o **mesmo hashCode**.

3. toString()

- Retorna uma **representação textual** do objeto;
- É útil para depuração, logs e para tornar o objeto mais “legível”;
- Ajuda a reforçar a **linguagem ubíqua**: código + saída do sistema falam o mesmo idioma.

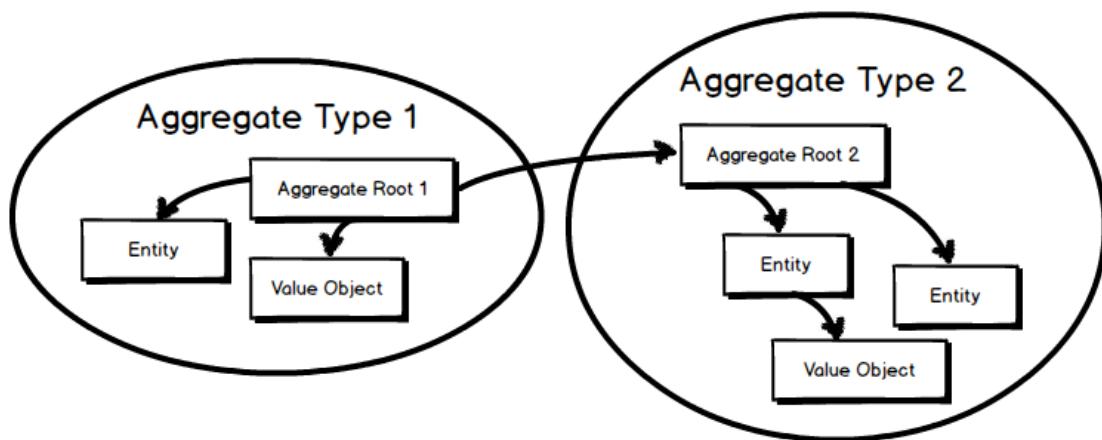
5. Agregados e Raiz de Agregado

Um agregado é um conjunto de objetos, e apenas a **raiz** deve ser acessada.

Aggregate Objects são diversas entidades que estão no mesmo contexto e que se consomem. Como exemplo, para cada pagamento realizado há um registro de uma transação. Logo, na tabela, para cada pagamento realizado há um registro ID de transação relacionado. Sempre que falamos em agregador, devemos informar quem é o root aggregator. Em nosso exemplo é o pagamento.

Um ponto importante para ressaltar é que quando falamos em DDD e Entidades, essas possuem as propriedades, métodos, validadores, para garantir que todas as informações que elas possuem estejam corretas. Vale lembrar que não estamos falando em Banco de Dados.

O DDD prega que as entidades devem desconhecer a existência do Banco de Dados, diferentemente no caso da utilização do Active Record onde a entidade conhece o Banco de Dados.



```
// Pedido expõe lista de itens diretamente
class Pedido {
    List<Item> itens = new ArrayList<>();
}
```

```
class Pedido {
    private List<Item> itens = new ArrayList<>();

    public void adicionarItem(String produto, int quantidade) {
        itens.add(new Item(produto, quantidade));
    }

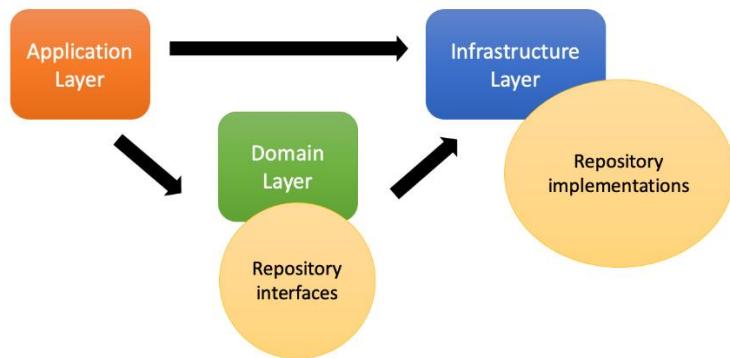
    public List<Item> getItens() {
        return Collections.unmodifiableList(itens);
    }
}
```

```
class Item {  
    private String produto;  
    private int quantidade;  
  
    public Item(String produto, int quantidade) {  
        this.produto = produto;  
        this.quantidade = quantidade;  
    }  
}
```

6. Reppositórios

Conceito: Abstraem a persistência como se fosse uma **coleção em memória**.

Os Reppositórios possuem acesso direto a camada de dados podendo persistir dados e realizar as consultas. Lembrando que devemos sempre utilizar um repositório por agregação. Um repositório também pode consultar diretamente serviços externos.



```
// Aplicação acessa banco diretamente
class App {
    public static void main(String[] args) {
        // select * from cliente where cpf=...
    }
}
```

```
interface ClienteRepositorio {
    void salvar(Cliente cliente);
    Cliente buscarPorCpf(String cpf);
}
```

```
class ClienteRepositoryEmMemoria implements ClienteRepository {  
    private Map<String, Cliente> banco = new HashMap<>();  
  
    public void salvar(Cliente cliente) {  
        banco.put(cliente.getCpf(), cliente);  
    }  
  
    public Cliente buscarPorCpf(String cpf) {  
        return banco.get(cpf);  
    }  
}
```

7. Serviços de Domínio

Conceito: Regras que não cabem em uma entidade ou objeto de valor. Se a regra não cabe como **comportamento de uma Entidade** (porque não depende só dela), nem como **atributo de um VO**, então ela deve estar em um **Serviço de Domínio**.

“O Serviço de Domínio existe para representar **ações do negócio** que não pertencem a nenhuma entidade específica. Ele é como um **verbo** dentro do domínio.”

Os Serviços de Domínio implementam a lógica de negócios a partir da definição de um expert de domínio. Trabalham com diversos fluxos de diversas entidades e agregações, utilizam os repositórios como interface de acesso aos dados e consomem recursos da camada de infraestrutura, como: enviar email, disparar eventos, entre outros.

Exemplos reais:

- **Processar pagamento** (envolve Cliente, Pedido, Dinheiro);
- **Calcular frete** (envolve Endereço de origem e destino, Tabela de preços);
- **Gerar relatório de reservas** (envolve várias Reservas e Clientes).



```
// Regra de desconto está no App
```

```
class App {  
    public static void main(String[] args) {  
        double preco = 100.0;  
        double desconto = preco * 0.1; // lógica aqui  
    }  
}
```

```
class CalculadoraDeDesconto {  
    public double aplicarDesconto(double preco) {  
        return preco * 0.9;  
    }  
}
```

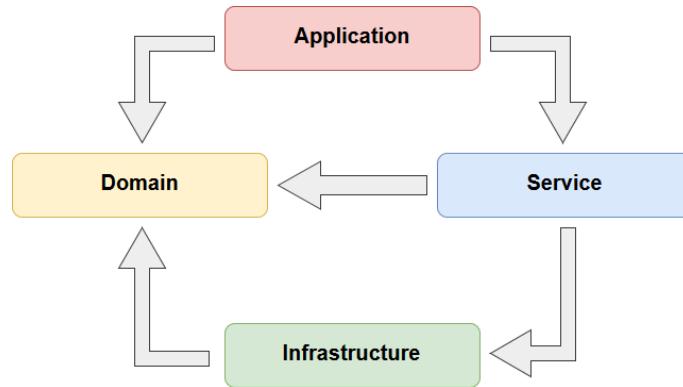
De acordo com SOLID:

- Single Responsibility Principle (SRP): evita “inflar” entidades com regras que não pertencem a elas;
- Interface Segregation Principle (ISP): separa regras em serviços especializados, sem forçar dependências desnecessárias.

8. Camada de Aplicação

Conceito:

Orquestra chamadas sem conter regras de negócio.



```
// Aplicação decide a regra
class App {
    public static void main(String[] args) {
        Cliente cliente = new Cliente("123", "João", 20);
        if (cliente.getIdade() >= 18) {
            System.out.println("Pode comprar!");
        }
    }
}
```

```
// Regra dentro da entidade, App só orquestra
class App {
    public static void main(String[] args) {
        Cliente cliente = new Cliente("123", "João", 20);

        if (cliente.podeComprar()) { //OU diretamente comprar
            System.out.println("Pode comprar!");
        }
    }
}
```

Ainda aqui, sugiro uma revisão de Tell, don't ask! (Object Calisthenics).