Aula DDD Java Pagamentos

🎯 Objetivo da aula

Ao final da aula, o aluno vai conseguir:

- Entender o que é DDD e por que existe
- Saber **quando usar e quando não usar**
- Montar um modelo de domínio **rico**
- Criar código Java com:
  - **Entidades**
  - **Value Objects**
  - **Serviços de Domínio**
  - **Repositórios**
  - **Aplicação (Use Cases)**
- Aplicar DDD com exemplos **curtos e práticos**

⏱️ Roteiro de 1 hora (tempo sugerido)

0–10 min — O problema que o DDD resolve

- Sistemas crescem
- Regras de negócio ficam espalhadas
- “CRUDzão” vira bagunça
- Bug aparece onde não deveria

10–25 min — Conceitos principais do DDD

- Domínio
- Modelo rico
- Entidade vs Value Object
- Aggregate / Aggregate Root

- Repositório
- Domain Service
- Application Service
- Ubiquitous Language (linguagem do negócio)

25–55 min — 4 exemplos práticos curtos (com código e explicação)

- Exemplo 1: Entidade + invariantes (ContaCorrente)
- Exemplo 2: Value Object (Dinheiro)
- Exemplo 3: Serviço de Domínio (Pagamento)
- Exemplo 4: Use Case (Aplicação) + Repositório (interface)

55–60 min — Quando usar / onde usar / vantagens / fechamento

✅ O que é DDD (explicação direta)

DDD é uma forma de projetar software onde o **centro do sistema é o negócio** (o domínio).

Em vez de fazer "tabela → entity → controller → crud", você faz:

📌 **Regras do negócio → modelo → comportamento → persistência**

DDD busca:

✅ **Código que representa o mundo real**
✅ **Regras concentradas no domínio**
✅ **Menos if espalhado no sistema**
✅ **Mais previsibilidade e evolução**

✅ Vantagens do DDD (por que usar)

🔥 1) Código fica "autoexplicativo"

Exemplo: `conta.debitar(valor)` é muito mais negócio do que `conta.setSaldo(...)`.

🔥 2) Regras ficam no lugar certo

Nada de regra no Controller ou Service "gordão".

🔥 3) Menos bugs por regras repetidas

A regra existe em 1 lugar.

🔥 4) Facilidade pra evoluir o sistema

Negócio muda → você muda o domínio.

🔥 5) Testes ficam fáceis

Você testa o domínio sem banco, sem Spring, sem web.

✅ Onde usar DDD

DDD é perfeito para sistemas que têm:

- **regras complexas**
- muitos tipos de pagamento
- limites, taxas, validações
- integrações e fluxos
- casos de negócio "cheios de detalhe"

📌 Exemplo perfeito:
💳 Conta corrente + pagamento + estorno + saldo + limite + auditoria

✅ Quando usar (e quando NÃO usar)

✅ Use DDD quando:

- o sistema tem regra de negócio forte
- vai crescer com o tempo
- tem muitas mudanças frequentes
- tem equipe e precisa de organização

❌ NÃO use DDD quando:

- é um CRUD simples (cadastro puro)
- projeto pequeno e rápido

- regras quase inexistentes

DDD não é moda, é **arma pra guerra**.

🧠 Conceitos essenciais (bem direto)

✅ Entidade

Tem identidade única e muda com o tempo.
📌 Ex: ContaCorrente (tem número)

✅ Value Object

Não tem identidade, é valor.
📌 Ex: Dinheiro (R$ 100,00)

✅ Aggregate

Conjunto de entidades que devem ser consistentes juntas.
📌 Ex: ContaCorrente + lista de lançamentos

✅ Aggregate Root

A "porta de entrada" do aggregate.
📌 Ex: ContaCorrente (você não mexe no saldo por fora)

✅ EXEMPLO 1 — Entidade com regra (ContaCorrente)

📌 **Objetivo:** mostrar entidade rica com regra de negócio dentro dela.

✅ Regra do negócio

- Não pode debitar valor <= 0

- Não pode sacar se saldo insuficiente

Código (curto e direto)

```java
import java.math.BigDecimal;

public class ContaCorrente {
    private final String numero;
    private BigDecimal saldo;

    public ContaCorrente(String numero, BigDecimal saldoInicial) {
        this.numero = numero;
        this.saldo = saldoInicial;
    }

    public void depositar(BigDecimal valor) {
        if (valor.compareTo(BigDecimal.ZERO) <= 0)
            throw new IllegalArgumentException("Depósito inválido");
        saldo = saldo.add(valor);
    }

    public void debitar(BigDecimal valor) {
        if (valor.compareTo(BigDecimal.ZERO) <= 0)
            throw new IllegalArgumentException("Débito inválido");

        if (saldo.compareTo(valor) < 0)
            throw new IllegalStateException("Saldo insuficiente");

        saldo = saldo.subtract(valor);
    }

    public BigDecimal getSaldo() {
        return saldo;
    }

    public String getNumero() {
        return numero;
    }
}
```

Explicação (DDD)

✅ O saldo **não é alterado por fora**
✅ As regras estão **dentro da entidade**
✅ O domínio está "falando a língua do negócio"

✅ EXEMPLO 2 — Value Object (Dinheiro)

📌 **Objetivo:** garantir consistência e evitar bugs com valores inválidos.

Problema comum sem DDD

Passar `BigDecimal` pra todo lado e esquecer validações.

Código (curto e direto)

```
java

import java.math.BigDecimal;

public record Dinheiro(BigDecimal valor) {

    public Dinheiro {
        if (valor == null || valor.compareTo(BigDecimal.ZERO) <= 0)
            throw new IllegalArgumentException("Valor inválido");
    }

    public Dinheiro somar(Dinheiro outro) {
        return new Dinheiro(this.valor.add(outro.valor));
    }

    public Dinheiro subtrair(Dinheiro outro) {
        return new Dinheiro(this.valor.subtract(outro.valor));
    }
}
```

Explicação (DDD)

✅ `Dinheiro` vira um **conceito do domínio**
✅ Evita valores negativos e null
✅ Cria um modelo mais "real"

✅ EXEMPLO 3 — Serviço de Domínio (Pagamento)

📌 **Objetivo:** quando uma regra envolve **mais de uma entidade**.

Regra do negócio

Pagamento transfere valor:

- Debita da conta origem
- Deposita na conta destino

Código (curto e direto)

```
java

public class PagamentoService {

    public void pagar(ContaCorrente origem, ContaCorrente destino,
Dinheiro valor) {
        origem.debitar(valor.valor());
        destino.depositar(valor.valor());
    }
}
```

Explicação (DDD)

📌 Isso é **Domain Service** porque:

- a regra não pertence só a uma conta
- envolve duas contas + valor
- é uma ação do domínio: "pagar"

✅ EXEMPLO 4 — Use Case (Aplicação) + Repositório

📌 **Objetivo:** separar domínio da aplicação e persistência.

Regras

- Buscar contas no repositório
- Executar pagamento

- Salvar contas alteradas

Código (curto e direto)

```java
public interface ContaCorrenteRepository {
    ContaCorrente buscarPorNumero(String numero);
    void salvar(ContaCorrente conta);
}
```

```java
public class RealizarPagamentoUseCase {

    private final ContaCorrenteRepository repository;
    private final PagamentoService pagamentoService;

    public RealizarPagamentoUseCase(ContaCorrenteRepository
repository) {
        this.repository = repository;
        this.pagamentoService = new PagamentoService();
    }

    public void executar(String contaOrigem, String contaDestino,
Dinheiro valor) {
        ContaCorrente origem =
repository.buscarPorNumero(contaOrigem);
        ContaCorrente destino =
repository.buscarPorNumero(contaDestino);

        pagamentoService.pagar(origem, destino, valor);

        repository.salvar(origem);
        repository.salvar(destino);
    }
}
```

Explicação (DDD)

Aqui temos camadas claras:

✅ **Domínio**: ContaCorrente, Dinheiro, PagamentoService
✅ **Aplicação**: RealizarPagamentoUseCase
✅ **Infra**: implementação do repository (JPA, JDBC, etc)

DDD separa responsabilidades com elegância.

✅ Resumo final (pra fechar a aula)

DDD é ideal quando o sistema tem:

- pagamento
- saldo
- regras financeiras
- consistência
- validações e fluxo

📌 DDD faz seu código parecer com o negócio:

- `debitar()`
- `depositar()`
- `pagar()`
- `Dinheiro`

> "DDD não é sobre pastas, é sobre colocar o negócio no centro e impedir que a regra vire bagunça."