
HACK
THE
DOMAIN

O Sistema Está Quebrado

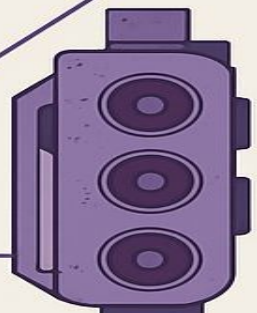
01

Você está dentro de uma corporação. Sistemas legados, integrações mal-feitas, código espaguete. Uma teia emaranhada de dependências. Nada responde como deveria. Seu deploy leva 40 minutos e quebra todo sábado à noite. Bem-vindo ao caos.

Você pensa que é só falta de testes. Mas não. O buraco é mais embaixo. O problema não está só na camada de controle. Está no domínio, que foi sufocado, esquecido.

As regras de negócio se escondem em controllers, em serviços genéricos, às vezes direto no banco. Ninguém mais entende o que o sistema realmente faz.

É aqui que entra o Domain-Driven Design. Não como uma bala de prata. Mas como uma arma de resistência. Uma forma de retomar o controle, modelar o caos e criar software com significado.



Domínio: A Verdade Está no Centro

02

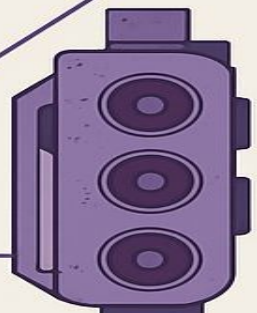
O domínio é o coração da aplicação. Tudo o que importa para o negócio vive aqui. É onde você modela as regras reais, a lógica que importa, a inteligência que diferencia seu sistema de um CRUD genérico.

O domínio deve ser limpo, isolado. Livre de frameworks, livre de dependências externas. Um núcleo puro. Um manifesto contra a dependência do mundo externo.

```java
// Exemplo de entidade em Java
public class Pedido {
    private final List<Item> itens;
    private final Cliente cliente;

    public Pedido(List<Item> itens, Cliente cliente) {
        if (itens == null || itens.isEmpty()) throw new IllegalArgumentException("Pedido vazio");
        this.itens = itens;
        this.cliente = cliente;
    }

    public BigDecimal calcularTotal() {
        return itens.stream()
            .map(Item::getPrecoTotal)
            .reduce(BigDecimal.ZERO, BigDecimal::add);
    }
}
```



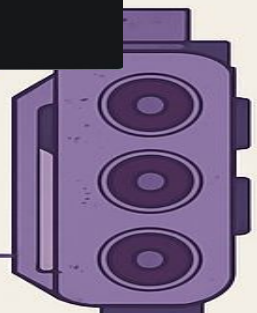
Ubiquitous Language: Fale Como o Domínio

03

O Você não programa sozinho. Você modela junto com especialistas de negócio. E para isso, precisam falar a mesma língua. Linguagem Ubíqua é o vocabulário comum entre devs e negócio. Ele nasce das conversas, dos contextos, das histórias reais. E ele vive no código.

Se o usuário fala em "faturamento parcial", o código não pode chamar isso de *FaturamentoParcialHandler*. Ele precisa dizer exatamente o que o domínio diz: *FaturamentoParcialService*. Sem traduções. Sem abstrações genéricas.

```
public class FaturamentoParcialService {
    public NotaFiscal gerarNota(List<Item> itens, Cliente cliente) {
        // lógica que segue o vocabulário do domínio
    }
}
```



Bounded Contexts: Quebrar Para Controlar

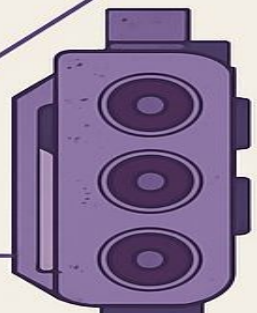
04

Em sistemas grandes, o domínio se fragmenta. O que "cliente" significa para o faturamento não é o mesmo para o suporte. O mesmo nome, significados distintos.

Bounded Context é a técnica para criar fronteiras claras no modelo. Dentro de um contexto, o vocabulário é consistente. Fora dele, pode mudar. E está tudo bem.

```
[Pedido] --(clienteId)--> [ClienteContextoFaturamento]
         --(clienteId)--> [ClienteContextoSuporte]
```

Separar contextos reduz acoplamento e aumenta clareza. Com ele, a organização do código reflete a organização do negócio. É como dividir a matrix em zonas seguras.



Value Objects:
O Significado Importa

05

Nem tudo precisa ser entidade. Às vezes, o valor é o que importa. Dois CPFs com o mesmo número são o mesmo CPF, mesmo que instâncias diferentes.

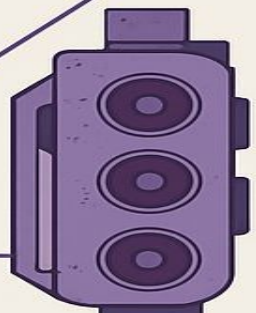
Value Objects são imutáveis, comparáveis por valor, e carregam significado. São pequenos guardiões da integridade do domínio.

```java
public class CPF {
    private final String valor;

    public CPF(String valor) {
        if (!valor.matches("\\d{11}")) {
            throw new IllegalArgumentException("CPF inválido");
        }
        this.valor = valor;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        CPF cpf = (CPF) o;
        return valor.equals(cpf.valor);
    }

    @Override
    public int hashCode() {
        return Objects.hash(valor);
    }
}
```



Value Objects:
O Significado Importa

06

Agregados são clusters de entidades e objetos de valor que funcionam como uma unidade de consistência.

Um Pedido é um agregado. Ele contém Itens, Cliente, Pagamento. Mas toda alteração deve passar pelo Pedido. Ele é a raiz.

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

    public void adicionarItem(Item item) {
        if (item == null || item.getQuantidade() <= 0) {
            throw new IllegalArgumentException("Item inválido");
        }
        itens.add(item);
    }

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

    public void finalizar() {
        if (itens.isEmpty()) {
            throw new IllegalStateException("Não é possível finalizar um pedido vazio.");
        }
    }
}
```