Domain-Driven Design (DDD) e TypeScript: Conceitos e Exemplos Práticos

Introdução

Este documento aborda os conceitos do Domain-Driven Design (DDD), uma abordagem para o desenvolvimento de software que se concentra no domínio do problema e na linguagem utilizada para descrevê-lo. Serão apresentados exemplos práticos em TypeScript para ilustrar como aplicar o DDD em projetos reais.

Domain-Driven Design (DDD): Uma Visão Geral

DDD é uma filosofia de desenvolvimento de software que enfatiza a colaboração entre especialistas de domínio (pessoas que entendem o problema de negócio) e desenvolvedores de software. O objetivo é criar um modelo de software que reflita com precisão o domínio do problema e que seja expresso em uma linguagem comum, chamada de Linguagem Ubíqua.

Principais Conceitos do DDD

Linguagem Ubíqua (Ubiquitous Language): É um vocabulário comum, compartilhado entre especialistas de domínio e desenvolvedores, que descreve o domínio do problema de forma clara e precisa.

Entidades (Entities): São objetos que possuem uma identidade única e um ciclo de vida. Elas representam os conceitos mais importantes do domínio e possuem atributos e comportamentos que refletem suas características.

Objetos de Valor (Value Objects): São objetos que não possuem uma identidade única e são definidos pelos seus valores. Eles representam características ou propriedades de entidades e são

imutáveis.

Agregados (Aggregates): São grupos de entidades e objetos de valor que são tratados como uma unidade. Eles possuem uma entidade raiz que é responsável por garantir a consistência e integridade do agregado.

Repositórios (Repositories): São responsáveis por persistir e recuperar agregados do armazenamento de dados.

Serviços de Domínio (Domain Services): Encapsulam a lógica de negócio que não se encaixa naturalmente em entidades ou objetos de valor.

Fábricas (Factories): São responsáveis por criar objetos complexos, como agregados, garantindo que eles sejam criados em um estado consistente.

Exemplos Práticos com TypeScript

1. Entidade (Entity)

```typescript
class Customer {
  private id: string;
  private name: string;

  constructor(id: string, name: string) {
    this.id = id;
    this.name = name;
  }
```

```
  getId(): string {
    return this.id;
  }

  getName(): string {
    return this.name;
  }

  changeName(newName: string): void {
    this.name = newName;
  }
}
```

2. Objeto de Valor (Value Object)

```typescript
class Address {
  private street: string;
  private city: string;
  private state: string;
  private zipCode: string;
```

```
  constructor(street: string, city: string, state: string, zipCode: string) {
    this.street = street;
    this.city = city;
    this.state = state;
    this.zipCode = zipCode;
  }

  // ... (getters)
}
```

3. Agregado (Aggregate)

```typescript
class Order {
  private id: string;
  private customer: Customer;
  private items: OrderItem[];

  constructor(id: string, customer: Customer) {
    this.id = id;
    this.customer = customer;
    this.items = [];
  }
```

```
  // ... (métodos para adicionar/remover itens, calcular total, etc.)
}
```

4. Repositório (Repository)

```typescript
interface OrderRepository {
  findById(id: string): Promise<Order>;
  save(order: Order): Promise<void>;
}
```

5. Serviço de Domínio (Domain Service)

```typescript
class OrderService {
  constructor(private orderRepository: OrderRepository) {}

  async placeOrder(customer: Customer, items: OrderItem[]): Promise<Order> {
    const order = new Order(uuidv4(), customer);
    for (const item of items) {
      order.addItem(item);
```

```
    }
    await this.orderRepository.save(order);
    return order;
  }
}
```

Conclusão

O Domain-Driven Design oferece uma abordagem poderosa para construir software complexo que reflete com precisão o domínio do problema. Ao usar o DDD em conjunto com TypeScript, você pode criar aplicações mais robustas, escaláveis e fáceis de manter.