



UniRuy & Área 1 | Wyden
PROGRAMA DE CIÊNCIA DA COMPUTAÇÃO
TEORIA DE COMPILADORES

JOÃO MARCELO TAVARES SOUZA MATOS

ARQUITETURA PARA
DESENVOLVIMENTO DE SISTEMAS
DISTRIBUIDOS DE ALTA PERFORMANCE
UTILIZANDO CQRS E DDD

Salvador - Bahia - Brasil

2022

JOÃO MARCELO TAVARES SOUZA MATOS

**ARQUITETURA PARA DESENVOLVIMENTO DE
SISTEMAS DISTRIBUIDOS DE ALTA
PERFORMANCE UTILIZANDO CQRS E DDD**

Trabalho Acadêmico elaborado junto ao programa de Engenharia UniRuy & Área 1 | Wyden, como requisito para obtenção de nota parcial da AV1 na disciplina Teoria de Compiladores no curso de Graduação em Ciência da Computação, que tem como objetivo consolidar os tópicos do plano de ensino da disciplina.

Orientador: Prof. MSc. Heleno Cardoso

Salvador - Bahia - Brasil

2022

da Tal, Aluno Fulano

Teoria de Compiladores: Resenha / Mapa Mental / Perguntas

– Aluno Fulano de Tal. Salvador, 2022.
18 f. : il.

Trabalho Acadêmico apresentado ao Curso de Ciência da Computação, UniRuy & Área 1 | Wyden, como requisito para obtenção de aprovação na disciplina Teoria de Compiladores.

Prof. MSc. Heleno Cardoso da S. Filho.

1. Resenha
2. Mapa Mental
3. Perguntas/Respostas (Mínimo de 03 – Máximo de 05)
4. Conclusão

I. da Silva Filho, Heleno Cardoso II. UniRuy & Área 1
| Wyden. III. Trabalho Acadêmico

CDD:XXX

TERMO DE APROVAÇÃO

JOÃO MARCELO TAVARES SOUZA MATOS

ARQUITETURA PARA DESENVOLVIMENTO DE SISTEMAS
DISTRIBUIDOS DE ALTA PERFORMANCE UTILIZANDO CQRS E
DDD

Trabalho Acadêmico aprovado como requisito para obtenção de nota parcial da AV1 na
disciplina Teoria de Compiladores, UniRuy & Área 1 | Wyden, pela seguinte banca
examinadora:

BANCA EXAMINADORA

Prof^o. MSc^o. Heleno Cardoso
Wyden

Salvador, 09 de Novembro de 2022

Dedico este trabalho acadêmico a todos que contribuíram direta ou indiretamente com
minha formação acadêmica.

Agradecimentos

Primeiramente agradeço a Deus. Ele, sabe de todas as coisas, e através da sua infinita misericórdia, se fez presente em todos os momentos dessa trajetória, concedendo-me forças e saúde para continuar perseverante na minha caminhada.

E a todos aqueles que contribuíram direta ou indiretamente para a minha formação acadêmica.

"A educaão tem raízes amargas, mas os seus frutos são doces".

Aristóteles.

Resumo

Entre os diversos problemas de sistemas de robustos de alta performance que usam sistemas distribuídos é como balancear a carga de escrita e leitura dessa aplicação, ainda mais quando se trata de sistemas que são feitos para terem maior leitura. A arquitetura usada por esse sistema nem sempre é a correta, as vezes o sistema foi pensado de uma forma e escalou de mais e não se comportou como devia, ou simplesmente achou-se que uma arquitetura mais simples daria conta. Por isso, muitos engenheiros debatem qual melhor solução para este problema e como fazer para escalar esses sistemas ao passar do tempo. O padrão Command Query Responsibility Segregation (CQRS) veio para sanar esse problema escalar e junto com o Domain Drive Design (DDD) e construir cada vez mais domínios ricos com desempenho favorável a todos os cenários de escalabilidade. Nesse artigo encontraremos um sistema feito em .NET Core 3.1 com os Commands (Escritas) e Querys (Leituras) segregadas e um domínio altamente rico com entidades e valores de objeto.

Palavras-chaves: CQRS, DDD, Arquitetura de software.

Abstract

Among the many problems of high-performance robust systems that use distributed systems is how to balance the write and application load, even more so when it comes to systems that are made for higher reading. The architecture used by this system is not always the correct one, sometimes the system was designed in one way and scaled too much and does not behave as it deviates, or simply thought that a simpler architecture would do. Therefore, many engineers debate the best solution to this problem and how to scale these systems over time. The Command Query Responsibility Segregation (CQRS) standard came to solve this scaling problem and together with Domain Drive Design (DDD) and build more and more rich domains with favorable performance for all scalability scenarios. In this article we will find a system made in .NET Core 3.1 with segregated Commands (Writes) and Querys (Reads) and a highly rich domain with entities and object values.

Keywords: CQRS, DDD, Software Architecture.

Lista de figuras

Figura 1 – Ciclo de Leitura e Escrita simplificada - Elaborado pelo autor	16
Figura 2 – Entidade Estudantes no .NET Core 3.1	17
Figura 3 – Entidade Pagamento no .NET Core 3.1	17
Figura 4 – Pagamento por boleto no .NET Core 3.1	18
Figura 5 – Pagamento por Cartão de Crédito no .NET Core 3.1	19
Figura 6 – Pagamento por PayPal no .NET Core 3.1	19
Figura 7 – Entidade Assinatura no .NET Core 3.1	20
Figura 8 – Objeto de valor endereço	21
Figura 9 – Command Handler de criação de Assinatura - Parte 1	22
Figura 10 – Command Handler de criação de Assinatura - Parte 2	22
Figura 11 – Command de criação de Assinatura por boleto	23
Figura 12 – Command de Resultado	23
Figura 13 – Query que busca informações do estudante pelo parâmetro Documento	25

Lista de abreviaturas e siglas

CQRS - Command Query Segregation Responsibility

CQS - Command Query Segregation

DDD - Domain Drive Design

QUERY - Comando de Consulta

COMMAND - Comando de escrita

.NET - framework criado pela Microsoft

VALUE OBJECT - Objetos de Valor

Sumário

1	Introdução	12
2	Referencial Teórico	13
2.1	Command Query Responsibility Segregation	13
2.2	Domain Drive Design	13
3	Metodologia	15
3.0.1	Modelagem do domínio	16
3.0.1.1	Estudantes (Students)	16
3.0.1.2	Pagamentos (Payments)	17
3.0.1.3	Pagamento por Boleto	18
3.0.1.4	Pagamento por Cartão de Crédito	18
3.0.1.5	Pagamento por Paypal	18
3.0.1.6	Assinatura (Subscriptions)	19
4	Value objects ou Objetos de valor	21
5	Commands ou Escrita	22
6	Query ou leitura	25
7	Resultados e Discussões	26
8	Considerações Finais	27
	Referências¹	28

¹ De acordo com a Associação Brasileira de Normas Técnicas. NBR 6023.

1 Introdução

As crescentes necessidades por soluções rápidas, confiáveis e escaláveis, normalmente envolvendo armazenamento de grande volume de dados, demandaram por novas formas de arquitetura de software. Nesse contexto, os dois modelos arquiteturais chamados CQRS (Command Query Responsibility Segregation) e DDD (Domain-Driven Design) demonstraram atender muito bem a essas necessidades.

De forma simplista, o CQRS separa os comandos que armazenam os dados, das consultas que os leem. O principal benefício do CQRS é que, separando o modelo de leitura do de escrita, você consegue otimizar esses modelos de forma independente para se obter o máximo desempenho de cada um deles.

No processo de escrita o foco se concentra diretamente nos comportamentos de domínio da aplicação, já na leitura na apresentação/visualização dos dados. Esta simples decisão que introduz algumas mudanças na arquitetura clássica, é capaz de produzir efeitos altamente positivos no desempenho da aplicação final.

Diferente do CRUD (Create, Read, Update e Delete) o CQRS trás uma diferenciação em sistemas complexos e potencialmente complexos por apresentar uma solução positiva de desempenho quando é apresentado grandes volumes de dados, tanto de entrada, tanto de saída envoltos de regras de negócio e validações extremamente rígidas.

Por isso, a estratégia de combinar o DDD e o CQRS encaixa bem em sistemas complexos de domínios ricos pois teremos um balanceamento de cargas nas leituras e na escrita.

Então, o desafio de criar um sistema complexo com domínio rico é basicamente como balancear a escrita e leitura e tornar o sistema mais leve mesmo robusto.

Já o DDD, é uma abordagem para lidar com domínios altamente complexos, tirando o foco do desenvolvimento e voltando-o para o domínio/negócio da aplicação.

É um modelo de software que utiliza uma profunda compreensão do domínio da aplicação no mundo real, constituindo-se numa nova maneira de lidar com projetos, uma nova abordagem de trabalho. Diversos Design Patterns, tais como: Value Object, Entity, Service, Aggregate, Repository, Application, Specification, Factories, IoC, ORM, AOP, etc são introduzidos para complementar esse modelo.

2 Referencial Teórico

2.1 Command Query Responsibility Segregation

O CQRS (Command Query Responsibility Segregation) foi descrito pela primeira vez por Greg Young, (YOUNG, 2010) e abordado por Martin Fowler (FOWLER, 2011) onde desenvolvedor do software pode usar um modelo diferente para atualizar as informações do modelo que ele usará para ler as mesmas informações. Para (FOWLER, 2011) é indispensável tomar cuidado ao usar esse PATTERN pois ele geralmente adiciona uma complexidade desnecessária ao sistema, o uso é realmente recomendado para sistemas complexos ou sistemas que poderão se tornar complexos ao longo do tempo.

Quando Greg Young descreveu em seu livro “CQRS Documents” (YOUNG, 2010) sobre o CQRS onde a separação de funções gera no sistema uma arquitetura mais eficaz e estudos nos quais outros Patterns foram aplicados junto com o CQRS. A Origem do CQRS originou-se do CQS de Bertrand Meyer (MEYER, 1986) onde a ideia principal é que a aplicação não pode ter em conjunto Commands e Queries, somente Commands separados de Queries. O maior diferencial entre o CQRS e CQS é que no CQRS os objetos são divididos em dois objetos, um com os comandos e outro com as consultas.

Para (YOUNG, 2010), essa separação traz benefícios que tendem a ser bastante uteis ao ser aplicados em sistemas complexos, alguns deles são: Consistência: no Command é muito mais fácil processar transações com dados consistentes do que lidar com os casos extremos e na Query a maioria dos sistemas pode eventualmente ser consistente no lado da consulta Escalabilidade: Nos sistemas o lado do Command geralmente processa menos dados transacionais do que a Query, por isso, escalabilidade é mais importante do lado da Query do que para o Command.

2.2 Domain Drive Design

DDD (Domain Driven Design) foi publicado pela primeira vez pelo autor (EVANS, 2014). Em seu livro “Domain-Driven Design: Tackling Complexity in the Heart of Software”, ele descreve o DDD como “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.” grosso modo, o DDD tem o foco de criar domínios ricos e facilitar o processo de

negócio quando transcrevido no software. Ao desenvolver softwares é necessário entender o contexto em que ele vai ser inserido, usado, qual ramo da empresa e quais as suas necessidades.

A Modelagem como um todo pode ser a peça-chave do sucesso desse software ao tratar o domínio como o coração do negócio a quem será servido o software (EVANS, 2014).

Mas, o que é o domínio nesse contexto? Domínio do Domain Drive Design é nada mais que o conjunto de regras, ideias, conhecimento e processo do negócio que esse software será inserido, como por exemplo o Prato Chefe de um restaurante é o Core Business dessa empresa, certamente ele será o Domínio principal desse negócio.

O DDD possui três grandes premissas, a linguagem Ubiquia, que é a linguagem falada no dia a dia, utilizada no sistema como termos que podem ser entendidos tanto pelo usuário final chamados por (EVANS, 2014) de especialistas no domínio e os desenvolvedores que implementam e dão manutenção no sistema.

Os Bounded Contexts ou contextos delimitados, que traz para o sistema as responsabilidades de cada contextos traduzidas e definidas para a linguagem Ubíqua, esse processo de contextualização ajuda no levantamento do escopo do projeto e auxilia o processo de delimitação dele. E por fim o Context Map ou mapeamento do contexto, que se é a relação entre os domínios.

3 Metodologia

Ao aplicarmos diretamente os ensinamentos de (YOUNG, 2010) no manual do CQRS, o “CQRS Documents”, junto com o que foi ensinado por (EVANS, 2014), em “Domain-Driven Design: Tackling Complexity in the Heart of Software” teremos uma aplicação robusta com o domínio altamente estruturado e diversos modelos para consulta e comandos, tornando o sistema mais flexível para cenários variados e complexos, diferentemente de um sistema que contém apenas um DTO para as operações CRUD (FOWLER, 2011). Para exemplificarmos em nosso estudo de caso sobre esses dois padrões Arquiteturais utilizarei o .NET Core 3.1 para o sistema.

Em nosso sistema, desenvolveremos com foco em uma Loja de cursos que o modelo de venda dessa loja é por assinatura mensal, semestral ou anual e existem alguns tipos de pagamentos, por boleto, cartão de crédito ou Paypal. O maior desafio dessa loja é o grande fluxo de leitura que recebe diariamente e o baixo volume de escrita, para isso, adaptamos o CQRS em dois bancos distintos para equilibrar esses volumes de dados, um SQL para escrita e um NOSQL para leitura, esses dois bancos a cada atualização de escrita, fornecesse uma atualização para o banco de leitura, tornando dinâmico e ágil.

Ao aplicarmos diretamente os ensinamentos de (YOUNG, 2010) no manual do CQRS, o “CQRS Documents”, junto com o que foi ensinado por (EVANS, 2014), em “Domain-Driven Design: Tackling Complexity in the Heart of Software” teremos uma aplicação robusta com o domínio altamente estruturado e diversos modelos para consulta e comandos, tornando o sistema mais flexível para cenários variados e complexos, diferentemente de um sistema que contém apenas um DTO para as operações CRUD (FOWLER, 2011). Para exemplificarmos em nosso estudo de caso sobre esses dois padrões Arquiteturais utilizarei o .NET Core 3.1 para o sistema.

Em nosso sistema, desenvolveremos com foco em uma Loja de cursos que o modelo de venda dessa loja é por assinatura mensal, semestral ou anual e existem alguns tipos de pagamentos, por boleto, cartão de crédito ou Paypal. O maior desafio dessa loja é o grande fluxo de leitura que recebe diariamente e o baixo volume de escrita, para isso, adaptamos o CQRS em dois bancos distintos para equilibrar esses volumes de dados, um SQL para escrita e um NOSQL para leitura, esses dois bancos a cada atualização de escrita, fornecesse uma atualização para o banco de leitura, tornando dinâmico e ágil.

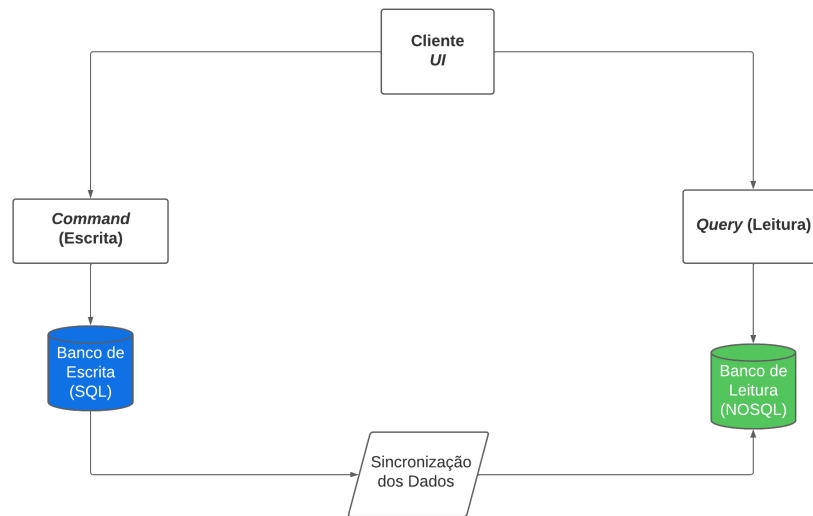


Figura 1 – Ciclo de Leitura e Escrita simplificada - Elaborado pelo autor

Na figura 1 descreve o Ciclo teórico de leitura e escrita utilizando dois bancos de dados e a relação entre o Command e Query e o caminho percorrido.

3.0.1 Modelagem do domínio

Na modelagem dos domínios, criaremos um domínio rico em nossa aplicação e teremos as seguintes entidades:

- Estudantes (Students)
- Pagamentos (Payments)
- Pagamento por boleto
- Pagamento por Cartão de Crédito
- Pagamento por Paypal
- Assinatura (Subscriptions)

3.0.1.1 Estudantes (Students)

Na figura 2 temos a Entidade Estudantes com os Object Values nome, documento, e-mail, e endereço e a chamada da Entidade Assinatura checando se existe ou não uma assinatura ativa. Abaixo temos um método de criação de Assinatura e a notificação caso for criada uma nova assinatura ou se existe uma já existente.

```

public class Student : Entity
{
    private IList<Subscription> _subscriptions;

    5 referências
    public Student(Name name, Document document, Email email)
    {
        Name = name;
        Document = document;
        Email = email;
        _subscriptions = new List<Subscription>();

        AddNotifications(name, document, email);
    }

    4 referências
    public Name Name { get; private set; }
    1 referência
    public Document Document { get; private set; }
    4 referências
    public Email Email { get; private set; }
    0 referências
    public Address Address { get; private set; }
    0 referências
    public IReadOnlyCollection<Subscription> Subscriptions { get { return _subscriptions.ToArray(); } }

    7 referências
    public void AddSubscription(Subscription subscription)
    {
        var hasSubscriptionActive = false;
        foreach( var sub in _subscriptions)
        {
            if (sub.Active)
                hasSubscriptionActive = true;
        }

        AddNotifications(new Contract<Student>()
            .Requires()
            .IsFalse(hasSubscriptionActive, "Student.Subscriptions", "Você já tem uma assinatura ativa")
            .AreEquals(0, subscription.Payments.Count, "Student.Subscription.Payments", "Esta assinatura não contém pagamento")
            );
    }
}

```

Figura 2 – Entidade Estudantes no .NET Core 3.1

3.0.1.2 Pagamentos (Payments)

```

14 referências
public abstract class Payment : Entity
{
    3 referências
    protected Payment(DateTime paidDate, DateTime expiredDate, decimal total, decimal totalPaid, string payer, Document document, Address address, Email email)
    {
        Number = Guid.NewGuid().ToString().Replace("-", "").Substring(0, 10).ToUpper();
        PaidDate = paidDate;
        ExpiredDate = expiredDate;
        Total = total;
        TotalPaid = totalPaid;
        Payer = payer;
        Document = document;
        Address = address;
        Email = email;

        AddNotifications(new Contract<Payment>()
            .Requires()
            .IsLowerOrEqualThan(0, Total, "Payment.Total", "O total não pode ser zero")
            .IsGreaterOrEqualThan(Total, TotalPaid, "Payment.Total", "O valor pago é menor que o valor do pagamento")
            );
    }

    1 referência
    public string Number { get; private set; }
    2 referências
    public DateTime PaidDate { get; private set; }
    1 referência
    public DateTime ExpiredDate { get; private set; }
    3 referências
    public decimal Total { get; private set; }
    2 referências
    public decimal TotalPaid { get; private set; }
    1 referência
    public string Payer { get; private set; }
    1 referência
    public Document Document { get; private set; }
    1 referência
    public Address Address { get; private set; }
    1 referência
    public Email Email { get; private set; }
}

```

Figura 3 – Entidade Pagamento no .NET Core 3.1

Na figura 3 temos a Entidade Pagamento que só será referenciada ao escolher Entre um pagamento por Boleto, Cartão de Crédito ou Paypal, nessa entidade contém um número do cartão, data de Pagamento, data de expiração, total da assinatura, total pago, pagador, documento, endereço e e-mail. E um construtor passando todos os parâmetros citados acima.

3.0.1.3 Pagamento por Boleto

```

2 referência
public class BoletoPayment : Payment
{
    1 referência
    public BoletoPayment(
        string barCode,
        string boletoNumber
        , DateTime paidDate
        , DateTime expiredDate
        , decimal total
        , decimal totalPaid
        , string payer
        , Document document
        , Address address
        , Email email)
        : base
        (paidDate
        , expiredDate
        , total
        , totalPaid
        , payer
        , document
        , address
        , email)
    {
        BarCode = barCode;
        BoletoNumber = boletoNumber;
    }
    1 referência
    public string BarCode { get; private set; }
    1 referência
    public string BoletoNumber { get; private set; }
}

```

Figura 4 – Pagamento por boleto no .NET Core 3.1

Na figura 4 temos o pagamento por Boleto, ele tem apenas 2 variáveis, código de barras e o número do boleto, o restante vem de herança da entidade Pagamento.

3.0.1.4 Pagamento por Cartão de Crédito

Na figura 5 temos o pagamento por Cartão de Crédito, ele tem apenas 3 variáveis, nome do pagador escrito no cartão, número do cartão e código de transação, o restante vem de herança da entidade Pagamento.

3.0.1.5 Pagamento por Paypal

Na figura 6 temos o pagamento por PayPal, ele tem apenas uma variável que é o Código de transação fornecido pelo Paypal, o restante vem de herança da entidade Pagamento

```

public class CreditCardPayment : Payment
{
    1 referência
    public CreditCardPayment(
        string cardHolderName
        , string cardNumber
        , string lastTransactionNumber
        , DateTime paidDate
        , DateTime expiredDate
        , decimal total
        , decimal totalPaid
        , string payer
        , Document document
        , Address address
        , Email email)
        : base(paidDate, expiredDate, total, totalPaid, payer, document, address, email)
    {
        CardHolderName = cardHolderName;
        CardNumber = cardNumber;
        LastTransactionNumber = lastTransactionNumber;
    }

    1 referência
    public string CardHolderName { get; private set; }
    1 referência
    public string CardNumber { get; private set; }
    1 referência
    public string LastTransactionNumber { get; private set; }
}

```

Figura 5 – Pagamento por Cartão de Crédito no .NET Core 3.1

```

public class PayPalPayment : Payment
{
    3 referências
    public PayPalPayment(
        string transactionCode
        , DateTime paidDate
        , DateTime expiredDate
        , decimal total
        , decimal totalPaid
        , string payer
        , Document document
        , Address address
        , Email email)
        : base ( paidDate, expiredDate, total, totalPaid, payer, document, address, email)
    {
        TransactionCode = transactionCode;
    }
    1 referência
    public string TransactionCode { get; private set; }
}

```

Figura 6 – Pagamento por PayPal no .NET Core 3.1

3.0.1.6 Assinatura (Subscriptions)

Na figura 7 temos a entidade Assinatura, de todas ela é a mais complexa, pois todas as entidades necessitam dela, seja o estudante que o sistema faz a checagem se a assinatura está ativa ou não e o pagamento que fará com que a assinatura esteja ativa e portanto ative o estudante dentro da loja. Temos dentro de assinatura 4 variáveis, quando foi criado, quando foi atualizado e quando foi expirado a assinatura e se está ativa ou não. Dentro dessa entidade temos também o método de criação de pagamento, ativação de assinatura e inativação da assinatura.

```

41 referencias
public class Subscription : Entity
{
    private IList<Payment> _payments;
    4 referencias
    public Subscription(DateTime? expiredOn)
    {
        CreatedOn = DateTime.Now;
        UpdatedOn = DateTime.Now;
        ExpiredOn = expiredOn;
        Active = true;
        _payments = new List<Payment>();
    }

    1 referencia
    public DateTime CreatedOn { get; private set; }
    3 referencias
    public DateTime UpdatedOn { get; private set; }
    1 referencia
    public DateTime? ExpiredOn { get; private set; }
    4 referencias
    public bool Active { get; private set; }
    1 referencia
    public IReadOnlyCollection<Payment> Payments { get { return _payments.ToArray(); } }

    5 referencias
    public void AddPayment(Payment payment)
    {
        AddNotifications(new Contract<Subscription>()
        {
            .Requires()
                .IsGreaterThan(DateTime.Now, payment.PaidDate, "Subscription.Payments", "A data do pagamento deve ser futura")
                );
        // if (Valid) // só adiciona se for valido
        _payments.Add(payment);
    }
    0 referencias
    public void Activate()
    {
        Active = true;
        UpdatedOn = DateTime.Now;
    }
    0 referencias
    public void Inactivate()
    {
        Active = false;
        UpdatedOn = DateTime.Now;
    }
}

```

Figura 7 – Entidade Assinatura no .NET Core 3.1

4 Value objects ou Objetos de valor

É perceptível nas entidades e nas sub-entidades de pagamento o uso de Value Objects ou objetos de valor, que nada mais são que objetos sem uma identidade conceitual clara e que dão característica a outro objeto. Como por exemplo, endereço:

```

public class Address : ValueObject
{
    4 referências
    public Address(string street, string number, string neighborhood, string city, string state, string country, string zipcode)
    {
        Street = street;
        Number = number;
        Neighborhood = neighborhood;
        City = city;
        State = state;
        Country = country;
        ZipCode = zipcode;

        AddNotifications(new Contract<Address>()
            .Requires()
            .IsLowerOrEqualsThan(Street, 3, "Address.Street", "A rua deve conter pelo menos 3 caracteres")
        );
    }

    2 referências
    public string Street { get; private set; }
    1 referência
    public string Number { get; private set; }
    1 referência
    public string Neighborhood { get; private set; }
    1 referência
    public string City { get; private set; }
    1 referência
    public string State { get; private set; }
    1 referência
    public string Country { get; private set; }
    1 referência
    public string ZipCode { get; private set; }
}

```

Figura 8 – Objeto de valor endereço

Endereço é um conjunto de muitas outras informações, como Endereço, número, bairro, cidade, estado, país e CEP e foi agrupada em uma só informação, Endereço. As maiores vantagens de usar os objetos de valor são imutabilidade, podendo ser usado em todo sistema sem alterações, evita uso desnecessário de tipos primitivos e várias regras em diversos pontos do sistema.

5 Commands ou Escrita

Ao chegarmos na parte de escrita, criamos separadamente uma assinatura para aquele usuário, neste exemplo usaremos o pagamento por boleto dentro do Command Handler da aplicação:

```
public ICommandResult Handle(CreateBoletoSubscriptionCommand command)
{
    // Fail Fast Validation
    command.Validate();
    if (!command.IsValid)
    {
        AddNotifications(command);
        return new CommandResult(false, "Não foi possível realizar seu cadastro");
    }

    // Verificar se documento já está cadastrado
    if (!_repository.DocumentExists(command.Document))
    {
        AddNotification("Document", "Este CPF já está em uso");
    }

    // Verificar se E-mail já está cadastrado
    if (!_repository.EmailExists(command.Email))
    {
        AddNotification("Email", "Este Email já está em uso");
    }

    // Gerar os VOs
    var name = new Name(command.FirstName, command.LastName);
    var document = new Document(command.Document, EDocumentType.CPF);
    var email = new Email(command.Email);
    var address = new Address(command.AddressStreet
        , command.AddressNumber
        , command.AddressNeighborhood
        , command.AddressCity, command.AddressState, command.AddressCountry, command.AddressZipCode);
}
```

Figura 9 – Command Handler de criação de Assinatura - Parte 1

```
// Gerar as Entidades
var student = new Student(name, document, email);
var subscription = new Subscription(DateTime.Now.AddDays(30));
var payment = new BoletoPayment(command.BarCode
    , command.BoletoNumber
    , command.PaidDate
    , command.ExpiredDate
    , command.Total
    , command.TotalPaid,
    command.Payer
    , new Document(command.PayerDocument, command.PayerDocumentType), address, email);

// Relacionamentos
subscription.AddPayment(payment);
student.AddSubscription(subscription);
//Aplicar as validações
AddNotifications(name, document, address, student, subscription, payment);
// Checar as notificações
if (!IsValid)
{
    return new CommandResult(false, "Não foi possível realizar sua assinatura");
}

//Salvar as informações
_repository.CreateSubscription(student);
//Enviar Email de boas vindas
_emailService.SendEmail(student.Name.ToString(), student.Email.Address, "Bem vindo ao Shop", "Sua Assinatura foi criada");
//retornar informações
return new CommandResult(true, "Assinatura realizada com sucesso");
}
```

Figura 10 – Command Handler de criação de Assinatura - Parte 2

Nas imagens 9 e 10 temos o Handler de criação de assinatura por boleto, onde o pagamento será por boleto. Primeiro ele faz uma validação no Command, verifica se o documento já está cadastrado, verifica se o e-mail está cadastrado, gera os objetos de valor, gera as entidades e faz as relações entre si, cria o pagamento, adiciona a assinatura e valida, se tudo for valido ele salva essas informações e envia um e-mail de boas-vindas e o sistema retorna um Command de Result.

Na imagem 11 temos as variáveis de criação do Command, que nada mais é que todas as variáveis das entidades Estudante, Pagamento, Pagamento por Boleto e Assinatura.

```

5 referências
public class CreateBoletoSubscriptionCommand : Notifiable<Notification>, ICommand
{
    5 referências
    public string FirstName { get; set; }
    3 referências
    public string LastName { get; set; }
    3 referências
    public string Email { get; set; }
    3 referências
    public string Document { get; set; }
    2 referências
    public string BarCode { get; set; }
    2 referências
    public string BoletoNumber { get; set; }
    1 referência
    public string PaymentNumber { get; set; }
    2 referências
    public DateTime PaidDate { get; set; }
    2 referências
    public DateTime ExpiredDate { get; set; }
    2 referências
    public decimal Total { get; set; }
    2 referências
    public decimal TotalPaid { get; set; }
    2 referências
    public string Payer { get; set; }
    2 referências
    public string PayerDocument { get; set; }
    2 referências
    public EDocumentType PayerDocumentType { get; set; }
    1 referência
    public string PayerEmail { get; set; }
    2 referências
    public string AddressStreet { get; set; }
    2 referências
    public string AddressNumber { get; set; }
    2 referências
    public string AddressNeighborhood { get; set; }
    2 referências
    public string AddressCity { get; set; }
    2 referências
    public string AddressState { get; set; }
    2 referências
    public string AddressCountry { get; set; }
    2 referências
    public string AddressZipCode { get; set; }
}

```

Figura 11 – Command de criação de Assinatura por boleto

```

11 referências
public class CommandResult : ICommandResult
{
    0 referências
    public CommandResult()
    {
    }
    9 referências
    public CommandResult(bool success, string message)
    {
        Success = success;
        Message = message;
    }
    1 referência
    public bool Success { get; set; }
    1 referência
    public string Message { get; set; }
}

```

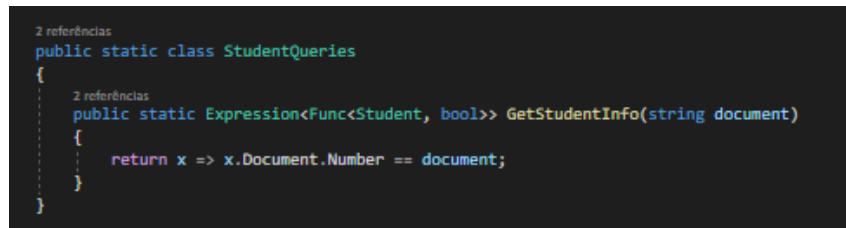
Figura 12 – Command de Resultado

Na imagem 12 temos o Command de resultado que será chamado após todas as validações da Assinatura por Boleto, se foi Sucesso ou não (True or False) e a mensagem configurada.

6 Query ou leitura

Existem muitas Querys para se fazer com a entidade estudante, de checar a quantidade de dias de assinatura, como checar seus dados cadastrais (dados da entidade estudante) ou saber qual foi o tipo de pagamento escolhido por ele.

Para exemplificar isso a imagem 13 traz a query onde o parâmetro é o Documento do estudante e retorna todas as informações do estudante.



```
2 referências
public static class StudentQueries
{
    2 referências
    public static Expression
```

Figura 13 – Query que busca informações do estudante pelo parâmetro Documento

7 Resultados e Discussões

No contexto das lojas, seja de qual tipo for, a balança dos dados sempre pesa mais para o lado da leitura, são feitas muitas consultas de preço, produto etc. dentro desse sistema, então para o nosso estudo de caso, observamos que usando CQRS e DDD, conseguimos otimizar essa balança, fazendo com que o sistema não ficasse pesado somente para a leitura, segregando em dois bancos de dados distintos (SQL para escrita e NOSQL para leitura). Essa separação de bancos possibilitou que o sistema reagisse melhor as alterações de dimensionamento, possibilitando o crescimento desse sistema e tornando o altamente escalável, do ponto de vista de domínio, esse sistema ficou altamente esclarecido, com suas entidades e objetos de valor definidos e estudados através de uma tradução da linguagem ubíqua para o domínio do sistema possibilitando os especialistas do domínio terem uma ampla compreensão de cada parte do sistema.

8 Considerações Finais

No projeto elaborado para este artigo foi elaborado uma loja sem os gargalos que a maioria dos sistemas de loja possuem e foi possível chegar no objetivo primário da pesquisa, fazendo com que o CQRS e o DDD sejam entendidos de forma sucinta e clara. É pretendido posteriormente adicionar a essa pesquisa, novas funcionalidades como a integração com o banco de dados não relacional e relacional para provar na prática que, segregando consultas da escrita, sistemas mais robustos conseguem sim trabalhar de forma efetiva sem pesar para um lado da balança. Esse trabalho poderá contribuir ativamente para engenheiros de Software que necessitam entender mais sobre Arquitetura de Software voltada para Serviços e sistemas distribuídos de alta performance.

Referências¹

- EVANS, E. *Domain-driven design : tackling complexity in the heart of software*. 2014. Citado 3 vezes nas páginas 13, 14 e 15.
- FOWLER, M. *CQRS*. 2011. Citado 2 vezes nas páginas 13 e 15.
- MEYER, B. *Eiffel : a language for software engineering*. 1986. Citado na página 13.
- YOUNG, G. *CQRS Documents by Greg Young*. 2010. Citado 2 vezes nas páginas 13 e 15.

¹ De acordo com a Associação Brasileira de Normas Técnicas. NBR 6023.