Seminário Programação orientado a objeto

Tema: "Validação de Regras de Negócio e Uso de Propriedades na Programação Orientada a Objetos com C#"

Regras de negócio

São condições, restrições ou diretrizes que definem ou restringem como um processo de negócio deve funcionar. Elas representam o conhecimento e as políticas da organização, sendo essenciais para garantir que o sistema siga os objetivos e limites do negócio.

> Exemplo geral: Os funcionários de uma loja só podem entrar na loja em horário comercial
>
> Exemplo :
>
> .public bool HorarioValido(DateTime horario)
>
> {
>
> return horario.Hour >= 8 && horario.Hour < 18;
>
> }

O encapsulamento consiste em proteger os dados internos da classe contra acessos diretos indevidos.

- As variáveis (atributos) são privadas.
- O acesso a elas é feito somente por métodos públicos.
- Isso permite controlar e validar o que entra e sai da classe.

Exemplo*:* A data e a lista de horários são privadas. Só podem ser modificadas usando métodos como RegistrarData

RegistrarData:

- Verifica se a data é válida (DateTime.TryParse).
- Garante que a data não seja no passado.
- Se a regra não for respeitada, lança um erro com

throw new ArgumentException(. . .)

Quando os dados não atendem às regras de negócio, usamos `throw` para interromper a execução e avisar que há um problema.

- A exceção pode ser capturada no código externo (`try/catch`) para mostrar uma mensagem clara ao usuário.

- Isso mantém o código mais seguro, organizado e fácil de entender

ValidarCompromisso ( )

Esse método serve para reunir todos os erros acumulados e, se houver algum:

- Cria uma mensagem com todos os erros.
- Lança uma exceção com throw new ArgumentException(...)

exemplo

```
public class Compromisso

{

    private DateTime data;



        public void RegistrarData(string textoData)

        {

        if (!DateTime.TryParse(textoData, out data))

        {

       throw new ArgumentException("Data inválida. Use o formato correto (ex: 05/05/2025).");

        }

        if (data < DateTime.Today)

        {

        throw new ArgumentException("A data não pode estar no passado.");

        }



        Console.WriteLine("Data registrada com sucesso: " + data.ToShortDateString());

    }

}
```

Conceito de Propriedades

Propriedades são membros de uma classe que fornece acesso controlado a campos privados.

Elas encapsulam o campo, permitindo que o programador controle a leitura e escrita dos dados de forma segura.

São declaradas usando **{ get; set; }.**

exemplo :public string Nome { get; set; }

Boas Práticas com Propriedades

- Use propriedades automáticas (`{ get; set; }`) sempre que possível para simplicidade.
- Use modificadores de acesso (`private set`) para proteger estados internos.
- Não exponha campos diretamente (`public string nome;` é má prática).
- Propriedades ajudam a manter o encapsulamento e a integridade dos dados.

```
public class Tarefa

{

    public string Nome { get; set; }

    public DateTime CriadaEm { get; }

    public string Status { get; private set; }

    public Tarefa(string nome)

    {

        Nome = nome;

        CriadaEm = DateTime.Now;

        Status = "Pendente";

    }



    public void Concluir()

    {

        Status = "Concluída";

    }

}
```

Object Initializers vs Construtores em C#

O que são Object Initializers?

São uma forma prática e rápida de criar objetos atribuindo valores diretamente às propriedades públicas com **`set`**

O que são Construtores?

Um construtor é um método especial de uma classe que é chamado automaticamente quando o objeto é criado.
Ele serve para inicializar o objeto e garantir que ele comece em um estado válido.

Requisitos para usar Object Initializers

- Só funcionam com propriedades públicas com set acessível.
- Não executam validações automaticamente — apenas atribuem valores diretamente.

Por que usar construtor é preferível em alguns casos?

- Quando há dados críticos, como **`Data`** e **`Hora`**, que precisam de validação:
- O construtor garante que os valores sejam verificados antes de o objeto ser usados
- Evita a criação de objetos inválidos.

Tabela Comparativa

Critério	Object Initializer	Construtor
Simplicidade	Mais simples para objetos leves	Mais detalhado
Validação de dados	Não faz validação automaticamente	Pode validar internamente
Segurança do estado	Pode criar objetos inválidos	Garante consistência
Reutilização de lógica	Repetição em cada uso	Lógica centralizada no construtor
Leitura de código	Leitura clara e objetiva	Levemente mais verboso
Recomendado para	DTOs, modelos simples	Objetos com lógica e regras de negócio

Boas Práticas Reforçadas

Validação centralizada na classe de domínio

- Toda regra de negócio deve estar encapsulada na própria entidade ou objeto de valor, evitando lógica espalhada pelo sistema.
- Isso garante coerência, reutilização e proteção contra estados inválidos

Uso de **`private set`** em propriedades críticas

- Protege o estado interno do objeto, permitindo alteração apenas por métodos controlados ou no construtor.

- Ajuda a manter o objeto imutável ou imutável por design, evitando efeitos colaterais inesperados.

Separação entre lógica de apresentação e lógica de negócio

- Interface (UI/API) não deve conter lógica de domínio.

- Favorece testabilidade, reusabilidade e manutenção.

Construtores com validação obrigatória → consistência do sistema

- Objetos só podem ser criados se estiverem em um estado válido.

- Impede que erros sejam tratados tardiamente no fluxo da aplicação.

Código deve ser autodocumentado por meio de nomes descritivos.

- .Código deve ser autodocumentado por meio de nomes descritivos.

- Comentários são úteis para explicar intenções ou casos complexos, mas não devem substituir bons nomes.

Conclusão e Recomendações

Usar propriedades com **private set**, ou propriedades somente leitura, garante que o estado do objeto só seja alterado de forma controlada, isso impede que o objeto entre em estados inválidos e mantém a coerência dos dados durante todo o ciclo de vida.

O modelo de domínio (ex: **Compromisso, Pedido, Reserva**) deve ser responsável por validar e aplicar as regras de negócio, isso evita duplicação de lógica em outras camadas e mantém a aplicação aderente às regras do domínio.

Use construtores com validação para entidades de domínio críticas, reserve object initializers para testes, DTOs ou modelos sem regras complexas

Lançar exceções com mensagens claras ajuda o desenvolvedor a entender rapidamente o que deu errado, isso facilita o debug, melhora os testes automatizados e fornece feedback imediato ao usuário ou à API.