






Notas sobre Sistemas

Type	Info
Topics	 Studies
Notebooks	 Programação Profissional com Clean Code
Related Notes	 Ementa do Curso

Arquitetura de Sistemas

- A separação de responsabilidades, modularidade e a escalabilidade são requisitos essenciais para o desenvolvimento de um sistema que possa crescer e se adaptar às mudanças ao longo do tempo
- A construção de um sistema exige planejamento, divisão de responsabilidades e serviços oferecidos (analogia de construir uma cidade)
- Exemplos
 - Quebrar cada um dos elementos e serviços de uma cidade

```
public class Cidade
{
    // Pense aqui numa cidade e como você deveria que
    brar
    // cada um dos elementos e servicos de uma cidade
}
```

- Separar construção e uso

```
// Exemplo Ruim: Misturando construção e uso
public class ExemploRuim
{
    public void Executar()
```

```

    {
        var servico = new Servico();
        servico.Configurar("Configuração Inicial");
        servico.ExecutarTarefa();
    }
}

// Exemplo Bom: Separando construção e uso
public class ExemploBom
{
    private readonly IServico _servico;

    // Injeção de dependência via construtor
    public ExemploBom(IServico servico)
    {
        _servico = servico;
    }

    public void Executar()
    {
        _servico.ExecutarTarefa();
    }
}

public interface IServico
{
    void Configurar(string configuracao);
    void ExecutarTarefa();
}

public class Servico : IServico
{
    private string _configuracao;

    public void Configurar(string configuracao)
    {
        _configuracao = configuracao;
    }
}

```

```

        public void ExecutarTarefa()
        {
            // Implementação da tarefa utilizando a configuração
        }
    }

    public static class Inicializador
    {
        public static ExemploBom Inicializar()
        {
            var servico = new Servico();
            servico.Configurar("Configuração Inicial");
            return new ExemploBom(servico);
        }
    }

    // Método Main isolado para inicialização
    public static class Programa
    {
        public static void Main()
        {
            var sistema = Inicializador.Inicializar();
            sistema.Executar();
        }
    }

```

- Isolar o código main

```

// Em cada plataforma temos um tipo de Main, no .NET
podemos considerar o program.cs um deles.

```

- Criar instâncias sem expor a lógica de criação (ex.: padrão Factory)

```

public interface IProduto { }

public class ProdutoA : IProduto { }

```

```

public class ProdutoB : IProduto { }

public static class FabricaProduto
{
    public static IProduto CriarProduto(string tipo)
    {
        switch (tipo)
        {
            case "A": return new ProdutoA();
            case "B": return new ProdutoB();
            default: throw new ArgumentException("Tip
o de produto desconhecido");
        }
    }
}

```

- Injetar as dependências

```

public interface IService { }
public class Service : IService { }

public class Consumidor
{
    private readonly IService _Service;

    // Injeção de dependência via construtor
    public Consumidor(IService Service)
    {
        _Service = Service;
    }
}

```

- Usar técnicas de escalabilidade

```

// Exemplo Ruim: Sistema não escalável
public class SistemaNaoEscalavel
{
    public void ProcessarDados()

```

```

    {
        List<string> dados = new List<string> { "Dado
1", "Dado2" };
        foreach (var dado in dados)
        {
            // Processamento ineficiente
            Console.WriteLine(dado);
        }
    }
}

```

```

// Exemplo Bom: Sistema escalável
public class SistemaEscalavel
{
    private readonly IProcessador _processador;

    // Injeção de dependência via construtor
    public SistemaEscalavel(IProcessador processador)
    {
        _processador = processador;
    }

    public void ProcessarDados()
    {
        IEnumerable<string> dados = ObterDadosDeFonte
Externa();
        _processador.Processar(dados);
    }

    private IEnumerable<string> ObterDadosDeFonteExte
rna()
    {
        // Simulação de obtenção de dados de uma font
e externa, por exemplo, banco de dados
        return new List<string> { "Dado1", "Dado2",
"Dado3" };
    }
}

```

```

public interface IProcessador
{
    void Processar(IEnumerable<string> dados);
}

public class Processador : IProcessador
{
    public void Processar(IEnumerable<string> dados)
    {
        // Processamento eficiente
        Parallel.ForEach(dados, dado =>
        {
            Console.WriteLine(dado);
        });
    }
}

// Método Main isolado para inicialização
public static class Program
{
    public static void Main()
    {
        IProcessador processador = new Processador();
        var sistema = new SistemaEscalavel(processador);

        sistema.ProcessarDados();
    }
}

```

Preocupações Transversais e Testes de Arquitetura

- Exemplos
 - Cross-Cutting Concerns, que afetam múltiplos módulos, como logging, segurança, autenticação e transações; evita duplicação de código e facilita a manutenção

```

public class Logger
{
    public void LogInfo(string mensagem) { /* ... */
}
    public void LogWarning(string mensagem) { /* ...
*/ }
    public void LogError(string mensagem) { /* ... */
}
}

public class Seguranca
{
    public void ValidarToken() { /* ... */ }
    public void InvalidarToken() { /* ... */ }
}

public class Transacao
{
    public void Begin() { /* ... */ }
    public void Commit() { /* ... */ }
    public void Rollback() { /* ... */ }
}

```

- Testar a arquitetura do sistema; garante que o design atende aos requisitos de desempenho, escalabilidade e segurança; não houve violação da responsabilidade das camadas

```

public class TestesDeArquitetura
{
    [Fact]
    public void Dominio_Nao_Deve_Dependere_De_Infraest
rutura()
    {
        // Arrange & Act
        var resultado = Types.InAssembly(typeof(Pedid
o).Assembly)
                                .That()
                                .ResideInNamespace("MeuP

```

```

        projeto.Dominio")
            .ShouldNot()
            .HaveDependencyOn("MeuPr
ojecto.Infraestrutura")
            .GetResult();

        // Assert
        Assert.True(resultado.IsSuccessful, "A camada
de domínio não deve depender da camada de infraestrut
ura.");
    }

    [Fact]
    public void Dominio_Nao_Deve_Dependder_De_Aplicaca
o()
    {
        // Arrange & Act
        var resultado = Types.InAssembly(typeof(Pedid
o).Assembly)
            .That()
            .ResideInNamespace("MeuP
rojecto.Dominio")
            .ShouldNot()
            .HaveDependencyOn("MeuPr
ojecto.Aplicacao")
            .GetResult();

        // Assert
        Assert.True(resultado.IsSuccessful, "A camada
de domínio não deve depender da camada de aplicaca
o.");
    }

    [Fact]
    public void Infraestrutura_Deve_Dependder_De_Domin
io()
    {
        // Arrange & Act
        var resultado = Types.InAssembly(typeof(Repos

```



```

torioEmMemoriaDePedido).Assembly)
        .That()
        .ResideInNamespace("MeuP
rojeta.Infraestrutura")
        .Should()
        .HaveDependencyOn("MeuPr
ojeto.Dominio")
        .GetResult();

        // Assert
        Assert.True(resultado.IsSuccessful, "A camada
de infraestrutura deve depender da camada de domíni
o.");
    }
}

```

- Otimizar a tomada de decisão; escolher a melhor abordagem para resolver um problema de design, considerando requisitos e restrições; análise de trade-offs entre simplicidade, flexibilidade, desempenho e custo
- Usar padrões de design (design patterns) com sabedoria; pode levar a complexidade desnecessária

```

public class PadraoSingleton
{
    private static PadraoSingleton instancia;

    private PadraoSingleton() { }

    public static PadraoSingleton Instancia
    {
        get
        {
            if (instancia == null)
            {
                instancia = new PadraoSingleton();
            }
            return instancia;
        }
    }
}

```

```
}  
  }  
}
```

- Usar corretamente as linguagens específicas de domínio (Ubiquitous Language) que descrevam soluções de maneira próxima à linguagem do domínio