






# Notas sobre Classes

Type	Info
Topics	 <a href="#">Studies</a>
Notebooks	 <a href="#">Programação Profissional com Clean Code</a>
Related Notes	 <a href="#">Ementa do Curso</a>

## Organização e Encapsulamento

- Classes são as unidades fundamentais de estrutura e comportamento; organização influencia a clareza, manutenção e extensibilidade; devem ser organizadas em pacotes ou namespaces que reflitam a funcionalidade e propósito
- Encapsulamento esconde os detalhes de implementação, expõe apenas o necessário; protege a integridade do objeto e previne alterações indesejadas
- Exemplos
  - Estruturar classes de maneira lógica e ordenada

```
public class OrganizaClasse
{
    // Ruim: métodos e variáveis misturados
    public void MetodoPublico() { /* ... */ }
    private int variavelPrivada;
    public int VariavelPublica;

    // Bom: variáveis e constantes antes dos métodos
    public const int CONSTANTE = 10;
    private static int variavelEstaticaPrivada;
    private int variavelInstanciaPrivada;
```

```

    public void MetodoPublicoOrdenado() { /* ... */ }
    private void MetodoPrivadoUtilitario() { /* ... */ }
}

```

- Proteger as variáveis e métodos internos

```

public class Encapsulamento
{
    // Ruim: variáveis públicas expostas
    public int dadosPublicos;

    // Bom: uso de métodos para acesso controlado
    private int dadosPrivados;

    public int GetDados() { return dadosPrivados; }
    public void SetDados(int valor) { dadosPrivados = valor; }
}

```

## Responsabilidade e Coesão

- Exemplos
  - Classes devem ser pequenas, o que facilita manter e testar; única responsabilidade ou razão para mudar

```

public class ClasseGrandeRuim
{
    // Ruim: Classe com muitas responsabilidades
    public void ProcessarDados() { /* ... */ }
    public void ConectarBancoDeDados() { /* ... */ }
    public void ValidarEntrada() { /* ... */ }
}

public class ClassePequenaBoa
{
    // Bom: Classes pequenas com responsabilidades únicas
    public void ProcessarDados() { /* ... */ }
}

```

```

public class ConexaoBancoDeDados
{
    public void Conectar() { /* ... */ }
}

public class ValidacaoEntrada
{
    public void Validar() { /* ... */ }
}

```

- Single Responsibility Principle (SRP) ajuda a manter a coesão

```

public class RelatorioRuim
{
    // Ruim: Classe com múltiplas responsabilidades
    public void GerarRelatorio() { /* ... */ }
    public void EnviarEmail() { /* ... */ }
}

public class RelatorioBom
{
    // Bom: Classe com uma única responsabilidade
    public void Gerar() { /* ... */ }
}

public class EmailService
{
    public void Enviar() { /* ... */ }
}

```

- Relacionar métodos na mesma classe (elementos com alta coesão)

```

public class PedidoRuim
{
    // Ruim: Métodos não relacionados na mesma classe
    public void Pagamento() { /* ... */ }
    public void Reclamacao() { /* ... */ }
    public void ConsultaCep() { /* ... */ }
}

```

```

}

public class PedidoBom
{
    // Bom: Métodos relacionados na mesma classe
    public void Processar() { /* ... */ }
    public void Cancelar() { /* ... */ }
}

```

- Manter a coesão resulta em dividir as responsabilidades em diversas classes pequenas

```

public class Pedido
{
    public void AdicionarItem() { /* ... */ }
}

public class Pagamento
{
    public void ProcessarPagamento() { /* ... */ }
}

```

- Estruturar para facilitar futuras mudanças; princípio Aberto/Fechado (OCP) que permite abrir para extensão, mas fecha para modificação

```

public abstract class OrdemServico
{
    public abstract void CriarOrdem();
    public abstract void AtualizarOrdem();
}

public class OrdemServicoSistemaAtual : OrdemServico
{
    public override void CriarOrdem()
    {
        // Lógica para criar ordem no sistema atual
    }
}

```

```

        public override void AtualizarOrdem()
        {
            // Lógica para atualizar ordem no sistema atual
        }
    }

    public class OrdemServicoNovoSistema : OrdemServico
    {
        public override void CriarOrdem()
        {
            // Nova lógica para criar ordem no novo sistema
        }

        public override void AtualizarOrdem()
        {
            // Nova lógica para atualizar ordem no novo sis
        }
    }

```

- Separar/isolar partes do código que mudam frequentemente

```

    public class ConfiguracaoSistema
    {
        // Ruim: Configuração diretamente no código
        public string ObterConfiguracao() { return "Config"
    }

    public class ConfiguracaoSistemaIsolada
    {
        private readonly IConfiguracaoRepositorio _configuracaoRepositorio;

        public ConfiguracaoSistemaIsolada(IConfiguracaoRepositorio configuracaoRepositorio)
        {
            _configuracaoRepositorio = configuracaoRepositorio;
        }

        public string ObterConfiguracao()
        {

```

```
        return _configuracaoRepositorio.BuscarConfiguracao();
    }
}

public interface IConfiguracaoRepositorio
{
    string BuscarConfiguracao();
}
```