# Contratos de Nulidade com Atributos em C#

Trilho: continuidade do nosso conteúdo sobre **tipos nuláveis e não nuláveis**, **associações 0..1/1..1**, **exceções** e **boas práticas de projeto**. Este material integra o que já vimos e o aplica de forma objetiva com **atributos de nulidade** para tornar contratos explícitos no código e enriquecer a análise estática do compilador/analisadores.

# Fundamentação — O que são "Atributos" em C#?

**Atributos** são **anotações declarativas** escritas entre colchetes ( [ ... ] ) que você aplica a tipos, membros, parâmetros, propriedades, retornos e até ao assembly. Na compilação, viram **metadados** gravados no assembly e podem ser **lidos pelo compilador, analisadores (Roslyn), ferramentas e frameworks**, além de serem acessíveis por **reflection** em tempo de execução.

Eles **não trocam o tipo** nem substituem a lógica de validação; **documentam intenções e regras** de forma executável, permitindo que ferramentas emitam avisos/erros, ajustem geração de código ou mudem comportamentos controlados por convenção.

# Exemplos Rápidos de Atributos

```
[Obsolete("Use o método XyzAsync")]
public void Foo() { }

public static bool TryGet([NotNullWhen(true)] out string? value)
{ /* se retornar true, value não é nulo */ }
```

No nosso trilho, os **atributos de nulidade** (de System.Diagnostics.CodeAnalysis) **refinam o contrato** que o NRT já expressa com T / T?: eles informam ao analisador **condições de fluxo** (ex.: "se retornar true, o out é não nulo"), reduzindo falsos positivos/negativos e deixando **APIs mais claras**, com menos if defensivo e menos falhas de null em runtime.

# Por que usar atributos de nulidade?

### Contratos Explícitos

**Deixar explícito, no código, quando algo pode ser null e em que condições** (contratos claros).

### Análise Aprimorada

**Aumentar o poder da análise do compilador** (NRT + analisadores): menos falsos positivos/negativos e **alertas precisos** onde de fato há risco de NullReferenceException.

### APIs Autoexplicativas

**APIs mais autoexplicativas**: menos if defensivo espalhado, chamadas mais seguras, e documentação viva no próprio código.

> **Resumo didático:** tipos `T` vs `T?` modelam a *possibilidade* de ausência; atributos refinam as **regras contextuais** (ex.: "se retornar true, este `out` **não** é nulo").

# Panorama Rápido (Revisão do Trilho)

01

### NRT habilitado

(`enable`): o compilador passa a exigir inicialização e checagens coerentes.

02

### Operadores e padrões

`?.`, `??`, `??=`, `!` (com parcimônia), `is null` / `is not null`.

03

### Associações 0..1 e 1..1

0..1 → referência anulável e regras de criação/remoção encapsuladas; 1..1 → obrigatória, criar/validar no construtor/fábrica.

04

### Exceções e guards

lançar cedo, proteger invariantes e preferir pontos de captura na borda.

Vamos amarrar tudo usando **atributos de nulidade** para documentar e reforçar esses contratos.

# Atributos Essenciais e Quando Usar

Importe: `using System.Diagnostics.CodeAnalysis;`

| Atributo | Quando aplicar | Efeito na análise |
|---|---|---|
| [NotNull] | Parâmetro que não pode ser nulo ao retornar (ex.: guards). | Após o método, o analisador assume não nulo. |
| [MaybeNull] | Método/out que pode devolver null mesmo com tipo não nulável (ex.: caches). | O chamador precisa tratar null. |

| Atributo | Quando aplicar | Efeito na análise |
| --- | --- | --- |
| [AllowNull] | Permitir atribuir null a uma propriedade não nulável (o set aceita null, o get mantém contrato). | Flexibiliza o setter sem romper o getter. |
| [NotNullWhen(bool)] | Em métodos booleanos de validação/try-parse. | Se a condição passada for verdadeira, o destino é não nulo. |
| [MemberNotNull] / [MemberNotNullWhen] | Método garante que certos membros serão não nulos ao retornar (ex.: inicialização tardia). | O analisador considera os membros listados como não nulos depois da chamada. |
| [DisallowNull] | Bloquear atribuição de null a membro que nunca deve receber null. | Impede set nulo em compilação/analisadores. |
| [NotNullIfNotNull("param")] | Se o parâmetro param não for nulo, o retorno também não será. | Propaga a relação de nulidade. |

# Exemplo 1 — Guard padrão com [NotNull]

```
public static class Guard
{
   public static void AgainstNull<[DynamicallyAccessedMembers(0)] T>(
      [NotNull] ref T? value, string paramName)
   {
      if (value is null)
         throw new ArgumentNullException(paramName);
   }
}
```

**Didática**: após `Guard.AgainstNull(ref x, ...)`, o compilador sabe que `x` **não** é nulo.

# Exemplo 2 — TryParse didático com [NotNullWhen(true)]

```
public static bool TryParseNonEmpty(string? s,
    [NotNullWhen(true)] out string? result)
{
    if (!string.IsNullOrWhiteSpace(s)) { result = s; return true; }
    result = null; return false;
}
```

> **Contrato**: se retornar true, result é **não nulo** (o chamador não precisa checar novamente).

# Exemplo 3 — Inicialização tardia com [MemberNotNull]

```
public class ReportService
{
  private FileInfo? _template;
  public string Generate()
  {
    EnsureTemplateLoaded();
    // a partir daqui, o analisador sabe que _template != null
    return _template!.FullName;
  }
  [MemberNotNull(nameof(_template))]
  private void EnsureTemplateLoaded()
  {
    _template ??= new FileInfo("template.docx");
  }
}
```

# Exemplo 4 — Setter permissivo com [AllowNull] e DisallowNull

```
public class Produto
{
   [DisallowNull]
   public string Nome { get; private set; } =
     string.Empty; // nunca retorna null
   // setter aceita null; semântica de "ausente"
   [AllowNull]
   public string? Descricao { get; set; }
}
```

Este exemplo mostra como controlar a nulidade em propriedades de forma granular, permitindo flexibilidade no setter enquanto mantém garantias no getter.

# Integração com o nosso trilho (associações, invariantes e exceções)

### a) Associações 0..1 (opcional)

Propriedade **anulável** (ex.: Passport?), com **setter privado** e métodos de domínio (EmitirPassaporte) que usam *guards* + atributos para documentar o contrato.

Valores inválidos → **exceções** no ponto de criação (fail-fast).

### b) Associações 1..1 (obrigatória)

Propriedade **não anulável**, inicializada **no construtor**/fábrica, e imutável para preservar o invariante "sempre um".

### c) Exceções & guards

Atributos **não substituem** validação; eles **documentam** para o compilador. Use ArgumentNullException, ArgumentException e exceções de domínio quando a regra for violada.

### d) Métodos estáticos utilitários (puros)

Coloque *guards*/validadores estáticos em classes focadas (SRP), facilitando reuso e reforçando os contratos em um só lugar.

# Padrões de uso (receitas rápidas)

01

### Bordas/DTOs → Domínio

Em mapeadores, trate opcionais (?) com TryParse... anotados. Se falhar, lance exceção ou retorne erro estruturado.

02

### Factories ricas

Ao construir entidades, combine `required`/construtor com *guards* + atributos para cristalizar o contrato de nulidade.

03

### Coleções

Prefira **coleção vazia** a null (reduz if defensivo).

04

### ! null-forgiving

evite como "silenciador"; prefira expressar a certeza com **atributos** (ex.: [MemberNotNull]).

# Anti-padrões e como detectar

### Tornar "tudo ?" por conveniência

perde-se semântica de ausência vs. obrigatoriedade.

### Setters públicos em vínculos 0..1/1..1

violam multiplicidade; o analisador **não** salvará um design frágil.

### Esquecer as bordas

sem validação de entrada, os atributos documentam o impossível.

## Checklist de revisão:

- ○ Tipos corretos (T vs T?) representam a regra do domínio.
- ○ Atributos de nulidade refinam o contrato onde o tipo não basta.
- ○ Guards/Exceções protegem invariantes **no ponto de entrada**.
- ○ Associações encapsuladas (setter privado + métodos claros).
- ○ Coleções nunca nulas.

# Laboratórios (aplicação imediata)

### Lab A — Guard + [NotNull]

Implemente `Guard.AgainstNull<T>([NotNull] ref T? value, string name)`. Demonstre que, após a chamada, o IDE não exige ? para `value`.

### Lab B — TryParse com [NotNullWhen(true)]

Crie `TryParseEmail(string? s, [NotNullWhen(true)] out string? email)`. Mostre uso sem ! no chamador quando o retorno for true.

### Lab C — Associação 0..1 com contrato explícito

`Person.Passport?` com método `IssuePassport(...)` que valida e, se atribuir, **garante** não nulo. Documente com atributos os efeitos (ex.: métodos auxiliares [MemberNotNull]).

# Lab D — Inicialização tardia

Classe com campo privado nulável e método anotado com [MemberNotNull] que assegura a não nulidade antes do uso.

```
public class DatabaseService
{
  private IDbConnection? _connection;

  [MemberNotNull(nameof(_connection))]
  private void EnsureConnection()
  {
    _connection ??= CreateConnection();
  }

  public void ExecuteQuery(string sql)
  {
    EnsureConnection();
    // _connection é garantidamente não nulo aqui
    _connection.Execute(sql);
  }
}
```

# Lab E — Revisão de API

Pegue um método público e **adicione atributos** para deixar explícitos: "se X, então Y não é nulo"; valide com testes/IDE.

```
// Antes
public bool TryGetUser(int id, out User user) { ... }

// Depois
public bool TryGetUser(int id, [NotNullWhen(true)] out User? user) { ... }
```

Este padrão elimina a necessidade de verificações adicionais de null quando o método retorna `true`, tornando o código mais limpo e seguro.

# Dicas de migração (código legado)

### Habilite NRT por módulo

Ative gradualmente para evitar sobrecarga de warnings.

### Ataque warnings das bordas para dentro

Comece pelas APIs públicas e vá refinando internamente.

### Modele o domínio sem null onde não faz sentido

Identifique conceitos que nunca deveriam ser nulos.

### Introduza atributos onde o compilador ainda não enxerga

Use atributos para documentar garantias de não nulidade.

### Substitua retornos null por valores neutros

Use `string.Empty`, `Enumerable.Empty()` quando apropriado.

# Exemplo Prático: Sistema de Usuários

```
public class UserService
{
  private readonly IUserRepository _repository;

  public UserService([NotNull] IUserRepository repository)
  {
    _repository = repository ??
      throw new ArgumentNullException(nameof(repository));
  }

  public bool TryGetUserByEmail(string? email,
    [NotNullWhen(true)] out User? user)
  {
    if (string.IsNullOrWhiteSpace(email))
    {
      user = null;
      return false;
    }

    user = _repository.FindByEmail(email);
    return user != null;
  }

  [return: NotNullIfNotNull("email")]
  public User? GetUserByEmail(string? email)
  {
    return string.IsNullOrWhiteSpace(email) ? null :
      _repository.FindByEmail(email);
  }
}
```

# Benefícios em Tempo de Desenvolvimento

### Análise Estática Aprimorada

- Menos falsos positivos nos warnings
- Detecção precisa de possíveis NullReferenceException
- IntelliSense mais inteligente
- Refatoração mais segura

### Qualidade do Código

- Redução de código defensivo desnecessário
- Maior confiança nas chamadas de método
- Menos bugs relacionados a null
- Manutenção mais fácil

### Documentação Viva

- Contratos explícitos no código
- Menos necessidade de comentários
- APIs autoexplicativas

### Experiência do Desenvolvedor

- Menos tempo debugando NullReferenceException
- Feedback imediato no IDE
- Maior produtividade

# Encerramento

Atributos de nulidade **não são "enfeites de docstring"**: eles **mudam o entendimento do compilador** sobre o seu código. Quando combinados com NRT, associações bem modeladas, *guards* e exceções, transformam a API em um **contrato executável**: claro para humanos, preciso para ferramentas e **seguro em tempo de execução**.

**Contratos Executáveis**

Documentação que o compilador entende e valida

**Análise Precisa**

Menos falsos positivos, mais detecção real de problemas

**Código Mais Seguro**

Redução significativa de NullReferenceException

> A combinação de **tipos nuláveis**, **atributos de nulidade** e **boas práticas de design** cria um ecossistema robusto onde a segurança de null é garantida tanto em tempo de compilação quanto em tempo de execução.