

Diretrizes de design de estrutura

Artigo • 27/01/2024

Esta seção apresenta diretrizes para criar bibliotecas que se estendem e interagem com o .NET Framework. A meta é ajudar os designers de biblioteca a garantir a consistência e a facilidade de uso da API fornecendo um modelo de programação unificado independente da linguagem de programação usada para desenvolvimento. Recomendamos que você siga essas diretrizes de design ao desenvolver classes e componentes que estendem o .NET Framework. O design inconsistente da biblioteca afeta negativamente a produtividade do desenvolvedor e desencoraja a adoção.

As diretrizes são organizadas como recomendações simples prefixadas com os termos **Do**, **Consider**, **Avoid** e **Do not**. Estas diretrizes têm como objetivo ajudar os designers de biblioteca de classes a entender os prós e contras de soluções diferentes. Pode haver situações em que um bom design de biblioteca exige que você não siga estas diretrizes de design. Esses casos devem ser raros, e é importante que você tenha uma razão clara e convincente para sua decisão.

Essas diretrizes são extraídas do livro *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition*, de Krzysztof Cwalina e Brad Abrams.

Nesta seção

[Diretrizes de nomenclatura](#)

Fornece diretrizes para nomear assemblies, namespaces, tipos e membros em bibliotecas de classes.

[Diretrizes de Design de tipo](#)

Fornece diretrizes para usar classes estáticas e abstratas, interfaces, enumerações, estruturas e outros tipos.

[Diretrizes de design de membro](#)

Fornece diretrizes para projetar e usar propriedades, métodos, construtores, campos, eventos, operadores e parâmetros.

[Designer voltado para extensibilidade](#)

Discute mecanismos de extensibilidade, como subclasse, uso de eventos, membros virtuais e retornos de chamada, e explica como escolher os mecanismos que melhor atendem aos requisitos da estrutura.

Diretrizes de design para exceções

Descreve as diretrizes de design para criar, gerar e capturar exceções.

Diretrizes de uso

Descreve as diretrizes para usar tipos comuns, como matrizes, atributos e coleções, dar suporte à serialização e sobrecarregar operadores de igualdade.

Padrões comuns de Design

Apresenta diretrizes para escolher e implementar propriedades de dependência.

Portions © 2005, 2009 Microsoft Corporation. Todos os direitos reservados.

Reimpresso com permissão da Pearson Education, Inc. das [Diretrizes de Design do Framework: convenções, linguagens e padrões para bibliotecas do .NET reutilizável](#), 2ª edição [↗](#) por Krzysztof Cwalina e Brad Abrams, publicado em 22 de outubro de 2008 por Addison-Wesley Professional como parte da série de desenvolvimento do Microsoft Windows.

Confira também

- [Visão geral](#)
- [Guia de desenvolvimento](#)

Diretrizes de nomenclatura

Artigo • 10/05/2023

Seguir um conjunto consistente de convenções de nomenclatura no desenvolvimento de um framework pode ser uma grande contribuição para sua usabilidade. Isso permite que a estrutura seja usada por muitos desenvolvedores em projetos amplamente separados. Além da consistência da forma, os nomes dos elementos da estrutura devem ser facilmente compreendidos e transmitir a função de cada elemento.

O objetivo deste capítulo é fornecer um conjunto consistente de convenções de nomenclatura que resulte em nomes que fazem sentido imediato para os desenvolvedores.

A adoção dessas convenções de nomenclatura como diretrizes gerais de desenvolvimento de código resulta em nomenclatura mais consistente em todo o código. No entanto, você só precisa aplicá-los a APIs expostas publicamente (tipos e membros públicos ou protegidos e interfaces explicitamente implementadas).

Nesta seção

[Convenções de maiúsculas e minúsculas](#)

[Convenções de nomenclatura gerais](#)

[Nomes de Assemblies e DLLs](#)

[Nomes de namespaces](#)

[Nomes de classes, structs e interfaces](#)

[Nomes de membros de tipo](#)

[Parâmetros de nomeação](#)

[Recursos de nomenclatura](#)

Portions © 2005, 2009 Microsoft Corporation. Todos os direitos reservados.

Reimpresso com permissão da Pearson Education, Inc. das [Diretrizes de Design do Framework: convenções, linguagens e padrões para bibliotecas do .NET reutilizável](#), 2ª edição [↗](#) por Krzysztof Cwalina e Brad Abrams, publicado em 22 de outubro de 2008 por Addison-Wesley Professional como parte da série de desenvolvimento do Microsoft Windows.

Confira também

- [Diretrizes de design do Framework](#)

Convenções de maiúsculas e minúsculas

Artigo • 05/03/2024

❗ Observação

Este conteúdo é reimpresso com permissão da Pearson Education, Inc. de *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition*. Essa edição foi publicada em 2008 e, desde então, o livro foi totalmente revisado [na terceira edição](#) [↗]. Algumas das informações nesta página podem estar desatualizadas.

As diretrizes neste capítulo estabelecem um método simples para usar maiúsculas e minúsculas que, quando aplicados de forma consistente, facilitam a leitura de identificadores para tipos, membros e parâmetros.

Regras de uso de maiúsculas e minúsculas para identificadores

Para diferenciar palavras em um identificador, coloque em maiúscula a primeira letra de cada palavra no identificador. Não use sublinhados para diferenciar palavras ou, nesse caso, em qualquer lugar nos identificadores. Há duas maneiras apropriadas de utilizar maiúsculas nos identificadores, dependendo do uso do identificador:

- PascalCasing
- camelCasing

A convenção PascalCasing, usada para todos os identificadores, exceto nomes de parâmetro, coloca em maiúscula o primeiro caractere de cada palavra (incluindo acrônimos com mais de duas letras de comprimento), conforme mostrado nos seguintes exemplos:

`PropertyDescriptor` `HtmlTag`

Uma exceção é feita para acrônimos de duas letras apenas, em que ambas as letras serão colocadas em maiúsculas, conforme mostrado no identificador a seguir:

`IOStream`

A convenção camelCasing, usada apenas para nomes de parâmetro, coloca em maiúsculas o primeiro caractere de cada palavra, exceto a primeira palavra, conforme

mostrado nos exemplos a seguir. Como o exemplo também mostra, os acrônimos de duas letras que iniciam um identificador ficam em minúsculas.

`PropertyDescriptor` `ioStream` `htmlTag`

✓ USE PascalCasing para todos os nomes de membros, tipos e namespaces públicos composto de várias palavras.

✓ USE camelCasing para nomes de parâmetro.

A tabela a seguir descreve as regras de uso de maiúsculas e minúsculas para diferentes tipos de identificadores.

[Expandir a tabela](#)

Identificador	Capitalização	Exemplo
Namespace	Pascal	<code>namespace System.Security { ... }</code>
Tipo	Pascal	<code>public class StreamReader { ... }</code>
Interface	Pascal	<code>public interface IEnumerable { ... }</code>
Método	Pascal	<code>public class Object { public virtual string ToString(); }</code>
Propriedade	Pascal	<code>public class String { public int Length { get; } }</code>
Evento	Pascal	<code>public class Process { public event EventHandler Exited; }</code>
Campo	Pascal	<code>public class MessageQueue { public static readonly TimeSpan InfiniteTimeout; } public struct UInt32 { public const Min = 0; }</code>
Valor de enumeração	Pascal	<code>public enum FileMode { Append, ... }</code>


Identificador	Capitalização	Exemplo
Parâmetro	Camel	<pre>public class Convert { public static int ToInt32(string value); }</pre>

Maiúsculas em palavras compostas e termos comuns

A maioria dos termos compostos é tratada como palavras simples para fins de uso de maiúsculas.

✗ NÃO USE maiúsculas em cada palavra nas chamadas palavras formadas por aglutinação e justaposição unidas sem o hífen.

Elas são formadas por mais de uma palavra escritas como uma única palavra sem separação ou uso de hífen, como “passatempo”. Para fins de diretrizes quanto ao uso de maiúsculas e minúsculas, trate essas palavras como uma única palavra. Use um dicionário atual para determinar se uma palavra é escrita por aglutinação/justaposição unidas e sem hífen.

 Expandir a tabela

Pascal	Camel	Not
BitFlag	bitFlag	Bitflag
Callback	callback	CallBack
Canceled	canceled	Cancelled
DoNot	doNot	Don't
Email	email	EMail
Endpoint	endpoint	EndPoint
FileName	fileName	Filename
Gridline	gridline	GridLine
Hashtable	hashtable	HashTable
Id	id	ID
Indexes	indexes	Indices

Pascal	Camel	Not
LogOff	logOff	LogOut
LogOn	logOn	LogIn
Metadata	metadata	MetaData, metaData
Multipanel	multipanel	MultiPanel
Multiview	multiview	MultiView
Namespace	namespace	NameSpace
Ok	ok	OK
Pi	pi	PI
Placeholder	placeholder	PlaceHolder
SignIn	signIn	SignOn
SignOut	signOut	SignOff
UserName	userName	Username
WhiteSpace	whiteSpace	Whitespace
Writable	writable	Writeable

Diferenciação de maiúsculas e minúsculas

Idiomas que podem ser executados em CLR não precisam oferecer suporte à diferenciação de maiúsculas e minúsculas, embora alguns ofereçam. Mesmo que seu idioma dê esse suporte, outros idiomas que podem acessar sua estrutura não oferecem. Todas as APIs que são acessíveis externamente, portanto, não podem depender apenas da diferenciação de maiúsculas e minúsculas para distinguir entre dois nomes no mesmo contexto.

✗ NÃO presume que todas as linguagens de programação façam diferenciação entre maiúsculas e minúsculas. Eles não são. Os nomes não podem se diferenciar apenas com base no uso de maiúsculas e minúsculas.

Portions © 2005, 2009 Microsoft Corporation. Todos os direitos reservados.

Reimpresso com permissão da Pearson Education, Inc. das [Diretrizes de Design do Framework: convenções, linguagens e padrões para bibliotecas do .NET reutilizável](#), 2ª

[edição](#) por Krzysztof Cwalina e Brad Abrams, publicado em 22 de outubro de 2008 por Addison-Wesley Professional como parte da série de desenvolvimento do Microsoft Windows.

Confira também

- [Diretrizes de design do Framework](#)
- [Diretrizes de nomenclatura](#)

Convenções de nomenclatura gerais

Artigo • 07/10/2023

🚨 Observação

Este conteúdo é reimpresso com permissão da Pearson Education, Inc., de *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition*. Essa edição foi publicada em 2008 e, desde então, o livro foi totalmente revisado na **terceira edição** [↗](#). Algumas das informações nesta página podem estar desatualizadas.

Esta seção descreve convenções de nomenclatura gerais relacionadas à escolha de palavras, diretrizes sobre o uso de abreviações e acrônimos e recomendações sobre como evitar o uso de nomes específicos da linguagem.

Escolha das palavras

✓ ESCOLHA nomes de identificador facilmente legíveis.

Por exemplo, uma propriedade nomeada `HorizontalAlignment` é mais legível em inglês do que `AlignmentHorizontal`.

✓ FAVOREÇA a legibilidade em vez da brevidade.

O nome de propriedade `CanScrollHorizontally` é melhor do que `ScrollableX` (uma referência obscura ao eixo X).

✗ NÃO use sublinhados, hifens ou outros caracteres não alfanuméricos.

✗ NÃO use notação húngara.

✗ EVITE o uso de identificadores que entram em conflito com palavras-chave de linguagens de programação amplamente usadas.

De acordo com a Regra 4 da CLS (Common Language Specification), todas as linguagens compatíveis devem fornecer um mecanismo que permita o acesso a itens nomeados que usam uma palavra-chave dessa linguagem como identificador. O C#, por exemplo, usa o sinal @ como um mecanismo de escape nesse caso. No entanto, ainda é uma boa ideia evitar palavras-chave comuns, porque é muito mais difícil usar um método com a sequência de escape do que um sem ela.

Usar abreviações e acrônimos

✗ NÃO use abreviações ou contrações como parte de nomes de identificador.

Por exemplo, use `GetWindow` em vez de `GetWin`.

✗ NÃO use acrônimos que não sejam amplamente aceitos e, mesmo que o sejam, somente quando necessário.

Evitar nomes específicos de linguagem

✓ USE nomes semanticamente interessantes em vez de palavras-chave específicas da linguagem para nomes de tipo.

Por exemplo, `GetLength` é um nome melhor do que `GetInt`.

✓ USE um nome de tipo CLR genérico, em vez de um nome específico da linguagem, nos casos raros em que um identificador não tem nenhum significado semântico além de seu tipo.

Por exemplo, um método de conversão para `Int64` deve ser nomeado `ToInt64`, não `ToLong` (porque `Int64` é um nome CLR para o alias `long` específico de C#). A tabela a seguir apresenta vários tipos de dados base usando os nomes de tipo CLR (bem como os nomes de tipo correspondentes para C#, Visual Basic e C++).

C#	Visual Basic	C++	CLR
sbyte	SByte	char	SByte
byte	Byte	unsigned char	Byte
short	Short	short	Int16
ushort	UInt16	unsigned short	UInt16
int	Inteiro	int	Int32
uint	UInt32	unsigned int	UInt32
longo	Long	__int64	Int64
ulong	UInt64	unsigned __int64	UInt64
float	Single	float	Single
double	Double	double	Double

C#	Visual Basic	C++	CLR
bool	Booleano	bool	Booleano
char	Char	wchar_t	Char
cadeia de caracteres	Cadeia de caracteres	Cadeia de caracteres	Cadeia de caracteres
object	Objeto	Objeto	Objeto

✓ USE um nome comum, como `value` ou `item`, em vez de repetir o nome do tipo, nos casos raros em que um identificador não tiver significado semântico e o tipo do parâmetro não for importante.

Nomear novas versões de APIs existentes

✓ USE um nome semelhante à API antiga ao criar novas versões de uma API existente.

Isso ajuda a destacar a relação entre as APIs.

✓ PREFIRA adicionar um sufixo em vez de um prefixo para indicar uma nova versão de uma API existente.

Isso ajudará a descoberta ao navegar na documentação ou a usar o IntelliSense. A versão antiga da API será organizada perto das novas APIs, pois a maioria dos navegadores e o IntelliSense mostram identificadores em ordem alfabética.

✓ CONSIDERE o uso de um identificador novo, mas significativo, em vez de adicionar um sufixo ou um prefixo.

✓ USE um sufixo numérico para indicar uma nova versão de uma API existente, especialmente se o nome existente da API for o único nome que faça sentido (ou seja, se for um padrão do setor) e se adicionar um sufixo significativo (ou alterar o nome) não for uma opção apropriada.

✗ NÃO use o sufixo "Ex" (ou similar) em um identificador para distingui-lo de uma versão anterior da mesma API.

✓ USE o sufixo "64" ao introduzir versões de APIs que operam em um inteiro de 64 bits (um inteiro longo) em vez de um inteiro de 32 bits. Você só precisa adotar essa abordagem quando houver a API de 32 bits; não faça isso com APIs novas com apenas a versão de 64 bits.

Reimpresso com permissão da Pearson Education, Inc. das *Diretrizes de Design do Framework: convenções, linguagens e padrões para bibliotecas do .NET reutilizável*, 2ª edição [↗](#) por Krzysztof Cwalina e Brad Abrams, publicado em 22 de outubro de 2008 por Addison-Wesley Professional como parte da série de desenvolvimento do Microsoft Windows.

Confira também

- [Diretrizes de design de estrutura](#)
- [Diretrizes de nomenclatura](#)
- [Convenções de nomenclatura .NET para EditorConfig](#)

Nomes de Assemblies e DLLs

Artigo • 12/03/2024

ⓘ Observação

Este conteúdo é reimpresso com permissão da Pearson Education, Inc. de *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition*. Essa edição foi publicada em 2008 e, desde então, o livro foi totalmente revisado [na terceira edição](#) [↗]. Algumas das informações nesta página podem estar desatualizadas.

Um assembly é a unidade de implantação e identidade para programas de código gerenciado. Embora os assemblies possam abranger um ou mais arquivos, normalmente um assembly mapeia um para um com uma DLL. Portanto, esta seção descreve apenas convenções de nomenclatura de DLL, que podem ser mapeadas para convenções de nomenclatura de assembly.

✓ ESCOLHA nomes para suas DLLs de assembly que sugiram grandes quantidades de funcionalidade, como System.Data.

Nomes de assembly e DLL não precisam corresponder a nomes de namespace, mas é razoável seguir o nome do namespace ao nomear assemblies. Uma boa regra geral é nomear a DLL com base no prefixo comum dos namespaces contidos no assembly. Por exemplo, um assembly com dois namespaces `MyCompany.MyTechnology.FirstFeature` e `MyCompany.MyTechnology.SecondFeature`, poderia ser chamado `MyCompany.MyTechnology.dll`.

✓ CONSIDERE nomear DLLs de acordo com o seguinte padrão:

```
<Company>.<Component>.dll
```

em que `<Component>` contém uma ou mais cláusulas separadas por pontos. Por exemplo:

```
Litware.Controls.dll
```

Portions © 2005, 2009 Microsoft Corporation. Todos os direitos reservados.

Reimpresso com permissão da Pearson Education, Inc. das [Diretrizes de Design do Framework: convenções, linguagens e padrões para bibliotecas do .NET reutilizável, 2ª edição](#) [↗] por Krzysztof Cwalina e Brad Abrams, publicado em 22 de outubro de 2008 por

Addison-Wesley Professional como parte da série de desenvolvimento do Microsoft Windows.

Confira também

- [Diretrizes de design do Framework](#)
- [Diretrizes de nomenclatura](#)

Nomes de namespaces

Artigo • 05/03/2024

ⓘ Observação

Este conteúdo é reimpresso com permissão da Pearson Education, Inc. de *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition*. Essa edição foi publicada em 2008 e, desde então, o livro foi totalmente revisado [na terceira edição](#) [↗]. Algumas das informações nesta página podem estar desatualizadas.

Assim como acontece com outras diretrizes de nomenclatura, a meta ao nomear namespaces está criando clareza suficiente para o programador usar a estrutura para saber imediatamente qual será o conteúdo do namespace. O modelo a seguir especifica a regra geral para nomear namespaces:

```
<Company>.( <Product> | <Technology> ) [ . <Feature> ] [ . <Subnamespace> ]
```

Os exemplos são os seguintes:

```
Fabrikam.Math   Litware.Security
```

✓ CRIE nomes de namespace de prefixo com um nome de empresa para evitar que namespaces de empresas diferentes tenham o mesmo nome.

✓ USE um nome de produto estável e independente de versão no segundo nível de um nome de namespace.

✗ NÃO use hierarquias organizacionais como base para nomes em hierarquias de namespace, pois os nomes de grupo dentro das corporações tendem a ser de curta duração. Organize a hierarquia de namespaces em torno de grupos de tecnologias relacionadas.

✓ USE PascalCasing e separe componentes de namespace com períodos (por exemplo, `Microsoft.Office.PowerPoint`). Se sua marca emprega maiúsculas e minúsculas não tradicionais, você deve seguir o que foi definido por sua marca, mesmo que isso se desvie do uso normal do namespace.

✓ CONSIDERE o uso de nomes de namespace no plural, quando apropriado.

Por exemplo, use `System.Collections` ao invés de `System.Collection`. No entanto, nomes de marcas e acrônimos são exceções a essa regra. Por exemplo, use `System.IO`

ao invés de `System.IOs`.

❌ NÃO use o mesmo nome para um namespace e um tipo nesse namespace.

Por exemplo, não use `Debug` como um nome de namespace e forneça também uma classe nomeada `Debug` no mesmo namespace. Vários compiladores exigem que esses tipos sejam totalmente qualificados.

Namespaces e conflitos de nome de tipo

❌ NÃO introduza nomes de tipo genéricos, como `Element`, `Node`, `Log` e `Message`.

Há uma probabilidade muito alta de que isso leve a conflitos de nome de tipo em cenários comuns. Você deve qualificar os nomes de tipo genérico (`FormElement`, `XmlNode`, `EventLog`, `SoapMessage`).

Há diretrizes específicas para evitar conflitos de nome de tipo para diferentes categorias de namespaces.

- **Namespaces do modelo de aplicativo**

Namespaces pertencentes a um único modelo de aplicativo geralmente são usados juntos, mas quase nunca são usados com namespaces de outros modelos de aplicativo. Por exemplo, o namespace `System.Windows.Forms` raramente é usado junto com o namespace `System.Web.UI`. Veja a seguir uma lista de grupos de namespace conhecidos do modelo de aplicativo:

`System.Windows*` `System.Web.UI*`

❌ NÃO dê o mesmo nome a tipos em namespaces em um único modelo de aplicativo.

Por exemplo, não adicione um tipo nomeado `Page` ao namespace `System.Web.UI.Adapters`, pois o namespace `System.Web.UI` já contém um tipo chamado `Page`.

- **Namespaces de infraestrutura**

Esse grupo contém namespaces que raramente são importados durante o desenvolvimento de aplicativos comuns. Por exemplo, namespaces `.Design` são usados principalmente ao desenvolver ferramentas de programação. Evitar conflitos com tipos nesses namespaces não é crítico.

- **Namespaces principais**

Os namespaces principais incluem todos os namespaces `System`, namespaces dos modelos de aplicativo e os namespaces de infraestrutura. Os namespaces principais incluem, entre outros, `System`, `System.IO` e `System.Xml` `System.Net`.

✗ NÃO forneça nomes de tipos que entrariam em conflito com qualquer tipo nos namespaces principais.

Por exemplo, nunca use `Stream` como um nome de tipo. Ele entraria em conflito com o `System.IO.Stream`, um tipo usado com muita frequência.

- **Grupos de namespaces de tecnologia**

Essa categoria inclui todos os namespaces com os mesmos dois primeiros nós de namespace (`<Company>.<Technology>*`), como `Microsoft.Build.Utilities` e `Microsoft.Build.Tasks`. É importante que os tipos pertencentes a uma única tecnologia não entrem em conflito entre si.

✗ NÃO atribua nomes de tipo que entrariam em conflito com outros tipos dentro de uma única tecnologia.

✗ NÃO introduza conflitos de nome de tipo entre tipos em namespaces de tecnologia e um namespace de modelo de aplicativo (a menos que a tecnologia não se destine a ser usada com o modelo de aplicativo).

Portions © 2005, 2009 Microsoft Corporation. Todos os direitos reservados.

Reimpresso com permissão da Pearson Education, Inc. das [Diretrizes de Design do Framework: convenções, linguagens e padrões para bibliotecas do .NET reutilizável](#), 2ª edição [↗](#) por Krzysztof Cwalina e Brad Abrams, publicado em 22 de outubro de 2008 por Addison-Wesley Professional como parte da série de desenvolvimento do Microsoft Windows.

Confira também

- [Diretrizes de design do Framework](#)
- [Diretrizes de nomenclatura](#)

Nomes de classes, structs e interfaces

Artigo • 07/10/2023

❗ Observação

Este conteúdo é reimpresso com permissão da Pearson Education, Inc., de *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition*. Essa edição foi publicada em 2008 e, desde então, o livro foi totalmente revisado na **terceira edição** [↗](#). Algumas das informações nesta página podem estar desatualizadas.

As diretrizes de nomenclatura a seguir se aplicam à nomenclatura de tipo geral.

✓ NOMEIE classes e structs com substantivos ou frases nominais, usando PascalCasing.

Isso distingue nomes de tipo de métodos, que são nomeados com frases verbais.

✓ NOMEIE interfaces com frases adjetivas ou, ocasionalmente, com substantivos ou frases nominais.

Substantivos e frases nominais devem ser usados raramente e podem indicar que o tipo deve ser uma classe abstrata e não uma interface.

✗ NÃO dê a nomes de classe um prefixo (por exemplo, "C").

✓ CONSIDERE encerrar o nome de classes derivadas com o nome da classe base.

Isso é muito legível e explica claramente a relação. Alguns exemplos disso no código são: `ArgumentOutOfRangeException`, que é uma espécie de `Exception`; e

`SerializableAttribute`, que é uma espécie de `Attribute`. No entanto, é importante adotar um julgamento razoável para aplicar essa diretriz: por exemplo, a classe `Button` é uma espécie de evento `Control`, embora `Control` não apareça em seu nome.

✓ FAÇA nomeações de interface de prefixo com a letra I, para indicar que o tipo é uma interface.

Por exemplo, `IComponent` (substantivo descritivo), `ICustomAttributeProvider` (frase nominal) e `IPersistable` (adjetivo) são nomes de interface apropriados. Da mesma forma que com outros nomes de tipo, evite abreviações.

✓ VERIFIQUE se os nomes diferem-se apenas pelo prefixo "I" no nome da interface quando você estiver definindo um par classe-interface em que a classe é uma

implementação padrão da interface.

Nomes dos parâmetros de tipo genérico

Genéricos foram adicionados ao .NET Framework 2.0. O recurso introduziu um novo tipo de identificador nomeado como *parâmetro de tipo*.

✓ NOMEIE parâmetros de tipo genérico com nomes descritivos, a menos que um nome de uma única letra seja autoexplicativo e um nome descritivo não agregue valor.

✓ CONSIDERE usar `T` como o nome do parâmetro de tipo em tipos com parâmetro de tipo de uma letra.

C#

```
public int IComparer<T> { ... }  
public delegate bool Predicate<T>(T item);  
public struct Nullable<T> where T:struct { ... }
```

✓ USE nomes de prefixo descritivos de parâmetro de tipo com `T`.

C#

```
public interface ISessionChannel<TSession> where TSession : ISession {  
    TSession Session { get; }  
}
```

✓ CONSIDERE indicar as restrições colocadas em um parâmetro de tipo no nome do parâmetro.

Por exemplo, um parâmetro restrito a `ISession` pode ser nomeado como `TSession`.

Nomes de tipos comuns

✓ SIGA as diretrizes descritas na tabela a seguir ao nomear tipos derivados ou implementar determinados tipos do .NET Framework.

Tipo base	Diretriz de tipo derivada/implementação
<code>System.Attribute</code>	✓ ADICIONE o sufixo "Attribute" aos nomes de classes de atributo personalizadas.
<code>System.Delegate</code>	✓ ADICIONE o sufixo "EventHandler" aos nomes de delegados que são usados em eventos.

Tipo base	Diretriz de tipo derivada/implementação
	<p>✓ ADICIONE o sufixo "Callback" a nomes de delegados diferentes daqueles usados como manipuladores de eventos.</p> <p>✗ NÃO adicione o sufixo "Delegado" a um delegado.</p>
<code>System.EventArgs</code>	✓ ADICIONE o sufixo "EventArgs".
<code>System.Enum</code>	<p>✗ NÃO derive dessa classe; em vez disso, use a palavra-chave com suporte em seu idioma; por exemplo, em C#, use a palavra-chave <code>enum</code>.</p> <p>✗ NÃO adicione o sufixo "Enum" ou "Flag".</p>
<code>System.Exception</code>	✓ ADICIONE o sufixo "Exception".
<code>IDictionary</code> <code>IDictionary<TKey, TValue></code>	✓ Adicione o sufixo "Dictionary". Observe que <code>IDictionary</code> é um tipo específico de coleção, mas essa diretriz tem precedência sobre as diretrizes de coleções mais gerais a seguir.
<code>IEnumerable</code> <code>ICollection</code> <code> IList</code> <code> IEnumerable<T></code> <code> ICollection<T></code> <code> IList<T></code>	✓ ADICIONE o sufixo "Collection".
<code>System.IO.Stream</code>	✓ ADICIONE o sufixo "Stream".
<code>CodeAccessPermission</code> <code>IPermission</code>	✓ ADICIONE o sufixo "Permission".

Nomenclatura de enumerações

Os nomes de tipos enumerados (também chamados de enumeração) em geral devem seguir as regras de nomenclatura de tipo padrão (PascalCasing, etc.). No entanto, há diretrizes adicionais que se aplicam especificamente a enumerações.

✓ USE um nome de tipo singular para uma enumeração, a menos que seus valores sejam campos de bits.

✓ USE um nome de tipo plural para uma enumeração com campos de bits como valores, também chamados de enumeração de sinalizadores.

✗ NÃO use um sufixo "Enum" em nomes de tipo enumerado.

✗ NÃO use sufixos "Flag" ou "Flags" em nomes de tipo enumerado.

✗ NÃO use um prefixo em nomes de valor de enumeração (por exemplo, "ad" para enumerações ADO, "rtf" para enumerações de rich text etc.).

Portions © 2005, 2009 Microsoft Corporation. Todos os direitos reservados.

Reimpresso com permissão da Pearson Education, Inc. das [Diretrizes de Design do Framework: convenções, linguagens e padrões para bibliotecas do .NET reutilizável](#), 2ª edição [↗](#) por Krzysztof Cwalina e Brad Abrams, publicado em 22 de outubro de 2008 por Addison-Wesley Professional como parte da série de desenvolvimento do Microsoft Windows.

Confira também

- [Diretrizes de design do Framework](#)
- [Diretrizes de nomenclatura](#)

Nomes de membros de tipo

Artigo • 12/03/2024

ⓘ Observação

Este conteúdo é reimpresso com permissão da Pearson Education, Inc. de *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition*. Essa edição foi publicada em 2008 e, desde então, o livro foi totalmente revisado [na terceira edição](#) [↗]. Algumas das informações nesta página podem estar desatualizadas.

Tipos são compostos de membros: métodos, propriedades, eventos, construtores e campos. As seções a seguir descrevem as diretrizes de nomenclatura de membros de tipo.

Nomes de métodos

Como os métodos são os meios para executar uma ação, as diretrizes de design exigem que os nomes de métodos sejam verbos ou frases verbais. Seguir essa diretriz também serve para distinguir os nomes de métodos de nomes de propriedades e tipos, que são substantivos ou frases adjetivas.

✓ NOMEIE os métodos que sejam verbos ou frases verbais.

C#

```
public class String {  
    public int CompareTo(...);  
    public string[] Split(...);  
    public string Trim();  
}
```

Nomes de propriedades

Diferentemente de outros membros, as propriedades devem ser nomeadas com uma frase substantivada ou um adjetivo. Isso porque uma propriedade se refere a dados, e o nome da propriedade reflete isso. PascalCasing sempre é usado para nomes de propriedade.

✓ NOMEIE as propriedades usando um substantivo, uma frase nominal ou um adjetivo.

✗ NÃO tenha propriedades que correspondam ao nome dos métodos "Get", como no exemplo a seguir:

```
public string TextWriter { get {...} set {...} } public string GetTextWriter(int value) { ... }
```

Esse padrão geralmente indica que a propriedade deveria realmente ser um método.

✓ NOMEIE as propriedades de coleção com uma frase no plural que descreva os itens na coleção em vez de usar uma frase no singular seguida por "List" ou "Collection".

✓ NOMEIE as propriedades booleanas com uma frase afirmativa (`CanSeek` em vez de `CantSeek`). Opcionalmente, você também pode prefixar as propriedades booleanas com "Is", "Can" ou "Has", mas somente quando isso adicionar valor.

✓ CONSIDERE nomear uma propriedade com o mesmo nome de seu tipo.

Por exemplo, a seguinte propriedade obtém e define corretamente um valor de enumeração denominado `Color`, portanto, a propriedade é chamada `Color`:

C#

```
public enum Color {...}
public class Control {
    public Color Color { get {...} set {...} }
}
```

Nomes de eventos

Eventos sempre fazem referência a uma ação, seja uma ação que está acontecendo ou que já ocorreu. Portanto, assim como acontece com os métodos, os eventos são nomeados com verbos e o tempo verbal é usado para indicar o horário em que o evento é acionado.

✓ NOMEIE os eventos com um verbo ou uma frase verbal.

Os exemplos incluem `Clicked`, `Painting`, `DroppedDown`, etc.

✓ NOMEIE os eventos com um conceito de antes e depois, usando os tempos verbais presente e pretérito.

Por exemplo, um evento de fechamento gerado antes de uma janela ser fechada seria chamado de `Closing`, e um gerado após a janela ser fechada seria chamado de `Closed`.

✗ NÃO use os prefixos ou sufixos "Before" ou "After" para indicar eventos anteriores e posteriores. Use os tempos verbais Pretérito e Presente conforme descrito.

✓ NOMEIE os manipuladores de eventos (delegados usados como tipos de eventos) com o sufixo "EventHandler", conforme mostrado no exemplo a seguir:

```
public delegate void ClickedEventHandler(object sender, ClickedEventArgs e);
```

✓ USE dois parâmetros nomeados como `sender` e `e` nos manipuladores de eventos.

O parâmetro do remetente representa o objeto que acionou o evento. O parâmetro do remetente normalmente é do tipo `object`, mesmo se for possível empregar um tipo mais específico.

✓ NOMEIE as classes de argumento de evento com o sufixo "EventArgs".

Nomes de campos

As diretrizes de nomenclatura de campo se aplicam a campos públicos e protegidos estáticos. Campos particulares e internos não são cobertos pelas diretrizes, e campos de instância pública ou protegida não são permitidos pelas [diretrizes de design de membro](#).

✓ USE PascalCasing nos nomes de campos.

✓ NOMEIE os campos usando um substantivo, uma frase nominal ou um adjetivo.

✗ NÃO use um prefixo para nomes de campos.

Por exemplo, não use "g_" ou "s_" para indicar campos estáticos.

Portions © 2005, 2009 Microsoft Corporation. Todos os direitos reservados.

Reimpresso com permissão da Pearson Education, Inc. das [Diretrizes de Design do Framework: convenções, linguagens e padrões para bibliotecas do .NET reutilizável](#), 2ª edição ↗ por Krzysztof Cwalina e Brad Abrams, publicado em 22 de outubro de 2008 por Addison-Wesley Professional como parte da série de desenvolvimento do Microsoft Windows.

Confira também

- Diretrizes de design do Framework
- Diretrizes de nomenclatura

Parâmetros de nomeação

Artigo • 06/03/2024

ⓘ Observação

Este conteúdo é reimpresso com permissão da Pearson Education, Inc. de *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition*. Essa edição foi publicada em 2008 e, desde então, o livro foi totalmente revisado [na terceira edição](#) [↗]. Algumas das informações nesta página podem estar desatualizadas.

Além do motivo óbvio da legibilidade, é importante seguir as diretrizes para nomes de parâmetros, pois eles são exibidos na documentação e no designer quando as ferramentas de design visual fornecem Intellisense e funcionalidade de navegação de classe.

- ✓ USE camelCasing em nomes de parâmetros.
- ✓ USE nomes de parâmetro descritivos.
- ✓ CONSIDERE o uso de nomes com base no significado de um parâmetro em vez do seu tipo.

Parâmetros de Sobrecarga de Operador de Nomenclatura

- ✓ USE `left` e `right` para nomes de parâmetro de sobrecarga de operador binário se não houver significado para os parâmetros.
- ✓ USE `value` para nomes de parâmetro de sobrecarga de operador unário se não houver significado para os parâmetros.
- ✓ CONSIDERE nomes significativos para parâmetros de sobrecarga de operador se isso acrescentar um valor significativo.
- ✗ NÃO use abreviações ou índices numéricos para nomes de parâmetro de sobrecarga de operador.

Portions © 2005, 2009 Microsoft Corporation. Todos os direitos reservados.

Reimpresso com permissão da Pearson Education, Inc. das [Diretrizes de Design do Framework: convenções, linguagens e padrões para bibliotecas do .NET reutilizável, 2ª edição](#) [↗] por Krzysztof Cwalina e Brad Abrams, publicado em 22 de outubro de 2008 por

Addison-Wesley Professional como parte da série de desenvolvimento do Microsoft Windows.

Confira também

- [Diretrizes de design do Framework](#)
- [Diretrizes de nomenclatura](#)

Recursos de nomenclatura

Artigo • 07/10/2023

ⓘ Observação

Este conteúdo é reimpresso com permissão da Pearson Education, Inc., de *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition*. Essa edição foi publicada em 2008 e, desde então, o livro foi totalmente revisado na **terceira edição** [↗](#). Algumas das informações nesta página podem estar desatualizadas.

Como os recursos localizáveis podem ser referenciados por meio de determinados objetos como se fossem propriedades, as diretrizes de nomenclatura para recursos são semelhantes às diretrizes de propriedade.

- ✓ Use PascalCasing em chaves de recurso.
- ✓ Fornece identificadores descritivos em vez de curtos.
- ✗ NÃO use palavras-chave específicas do idioma das principais linguagens CLR.
- ✓ Use apenas caracteres alfanuméricos e sublinhados em recursos de nomenclatura.
- ✓ Use a convenção de nomenclatura a seguir para recursos de mensagem de exceção.

O identificador de recurso deve ser o nome do tipo de exceção mais um identificador curto da exceção:

```
ArgumentExceptionIllegalCharacters    ArgumentExceptionInvalidName
```

```
ArgumentExceptionFileNameIsMalformed
```

Portions © 2005, 2009 Microsoft Corporation. Todos os direitos reservados.

Reimpresso com permissão da Pearson Education, Inc. das [Diretrizes de Design do Framework: convenções, linguagens e padrões para bibliotecas do .NET reutilizável, 2ª edição](#) [↗](#) por Krzysztof Cwalina e Brad Abrams, publicado em 22 de outubro de 2008 por Addison-Wesley Professional como parte da série de desenvolvimento do Microsoft Windows.

Confira também

- [Diretrizes de design do Framework](#)

- Diretrizes de nomenclatura

Diretrizes de design de tipo

Artigo • 10/05/2023

Da perspectiva do CLR, há apenas duas categorias de tipos: tipos de referência e tipos de valor. Porém, para fins de uma discussão sobre design de estrutura, dividimos tipos em grupos mais lógicos, cada um com as próprias regras de design específicas.

Classes são o caso geral de tipos de referência. Elas compõem a maior parte dos tipos na maioria das estruturas. As classes devem sua popularidade ao conjunto avançado de recursos orientados a objetos que dão suporte e à sua aplicabilidade geral. Classes base e classes abstratas são grupos lógicos especiais relacionados à extensibilidade.

Interfaces são tipos que podem ser implementados por tipos de referência e tipos de valor. Portanto, elas podem servir como raízes de hierarquias polimórficas de tipos de referência e tipos de valor. Além disso, as interfaces podem ser usadas para simular várias heranças, que não têm suporte nativo no CLR.

Structs são o caso geral de tipos de valor e devem ser reservados para tipos pequenos e simples, semelhantes aos primitivos de linguagem.

Enumerações são um caso especial de tipos de valor usados para definir conjuntos curtos de valores, como dias da semana, cores do console etc.

Classes estáticas são tipos destinados a serem contêineres para membros estáticos. Eles são comumente usados para fornecer atalhos para outras operações.

Delegados, exceções, atributos, matrizes e coleções são todos casos especiais de tipos de referência destinados a usos específicos e diretrizes para seu design e uso são discutidos em outros pontos neste livro.

✓ VERIFIQUE se cada tipo é um conjunto bem definido de membros relacionados, não apenas uma coleção aleatória de funcionalidades não relacionadas.

Nesta seção

[Escolher entre Classe e Struct](#)

[Design de classe abstrata](#)

[Design de classe estática](#)

[Design de interface](#)

[Design de Struct](#)

[Design de enumeração](#)

Tipos aninhados

Portions © 2005, 2009 Microsoft Corporation. Todos os direitos reservados.

Reimpresso com permissão da Pearson Education, Inc. das *Diretrizes de Design do Framework: convenções, linguagens e padrões para bibliotecas do .NET reutilizável*, 2ª edição [↗](#) por Krzysztof Cwalina e Brad Abrams, publicado em 22 de outubro de 2008 por Addison-Wesley Professional como parte da série de desenvolvimento do Microsoft Windows.

Confira também

- [Diretrizes de design do Framework](#)

Escolher entre Classe e Struct

Artigo • 07/10/2023

ⓘ Observação

Este conteúdo é reimpresso com permissão da Pearson Education, Inc. de *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition*. Essa edição foi publicada em 2008, e o livro desde então foi totalmente revisado na **terceira edição** [↗](#). Algumas das informações nesta página podem estar desatualizadas.

Uma das decisões básicas de design que cada designer de estrutura enfrenta é se vai criar um tipo como uma classe (um tipo de referência) ou como um struct (um tipo de valor). Uma boa compreensão das diferenças no comportamento dos tipos de referência e tipos de valor é crucial para fazer essa escolha.

A primeira diferença entre tipos de referência e tipos de valor que consideraremos é que os tipos de referência são alocados no heap e coletados como lixo, enquanto que os tipos de valor são alocados na pilha ou em linha na contenção de tipos e desalocados quando a pilha desenrola ou quando seu tipo de contenção é desalocado. Portanto, alocações e desalocações de tipos de valor são, em geral, mais baratas do que alocações e desalocações de tipos de referência.

Em seguida, as matrizes de tipos de referência são alocadas fora de linha, significando que os elementos da matriz são apenas referências a instâncias do tipo de referência que residem no heap. As matrizes de tipos de valor são alocadas em linha, significando que os elementos da matriz são as instâncias reais do tipo de valor. Portanto, alocações e desalocações de matrizes de tipos de valor são muito mais baratas do que alocações e desalocações de matrizes de tipos de referência. Além disso, na maioria dos casos, as matrizes de tipos de valor apresentam uma localidade de referência muito melhor.

A próxima diferença está relacionada ao uso de memória. Os tipos de valor são encaixotados quando convertidos em um tipo de referência ou em uma das interfaces que implementam. Eles são desencaixotados quando convertidos de volta para o tipo de valor. Como as caixas são objetos alocados no heap e são coletados como lixo, o excesso de conversão boxing e conversão unboxing pode ter um impacto negativo no heap, no coletor de lixo e, basicamente, no desempenho do aplicativo. Por outro lado, nenhuma conversão boxing como tal ocorre à medida que os tipos de referência são convertidos. (Para obter mais informações, consulte [Conversão boxing e Conversão unboxing](#).)

Em seguida, as atribuições de tipo de referência copiam a referência, enquanto as atribuições de tipo de valor copiam o valor inteiro. Portanto, as atribuições de tipos de referência grandes são mais baratas do que as atribuições de tipos de valor grandes.

Por fim, os tipos de referência são passados por referência, enquanto os tipos de valor são passados por valor. As alterações em uma instância de um tipo de referência afetam todas as referências que apontam para a instância. As instâncias de tipo de valor são copiadas quando são passadas por valor. Quando uma instância de um tipo de valor é alterada, é claro que isso não afeta nenhuma de suas cópias. Como as cópias não são criadas explicitamente pelo usuário, mas são criadas implicitamente quando os argumentos são passados ou os valores retornados são retornados, os tipos de valor que podem ser alterados podem ser confusos para muitos usuários. Portanto, os tipos de valor devem ser imutáveis.

Como regra geral, a maioria dos tipos em uma estrutura deve ser classes. Há, no entanto, algumas situações em que as características de um tipo de valor tornam mais apropriado usar structs.

✓ CONSIDERE definir um struct em vez de uma classe se as instâncias do tipo forem pequenas e normalmente de curta duração ou se forem comumente incorporadas em outros objetos.

✗ EVITE definir um struct, a menos que o tipo tenha todas as características a seguir:

- Representa logicamente um único valor, semelhante aos tipos primitivos (`int`, `double` etc.).
- Tem um tamanho de instância inferior a 16 bytes.
- É imutável.
- Não precisará ser encaixotado com frequência.

Em todos os outros casos, você deve definir seus tipos como classes.

Portions © 2005, 2009 Microsoft Corporation. Todos os direitos reservados.

Reimpresso com permissão da Pearson Education, Inc. das [Diretrizes de Design do Framework: convenções, linguagens e padrões para bibliotecas do .NET reutilizável](#), 2ª edição [↗](#) por Krzysztof Cwalina e Brad Abrams, publicado em 22 de outubro de 2008 por Addison-Wesley Professional como parte da série de desenvolvimento do Microsoft Windows.

Confira também

- Diretrizes de Design de tipo
- Diretrizes de design do Framework

Design de classe abstrata

Artigo • 05/03/2024

ⓘ Observação

Este conteúdo é reimpresso com permissão da Pearson Education, Inc. de *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition*. Essa edição foi publicada em 2008 e, desde então, o livro foi totalmente revisado [na terceira edição](#) [↗]. Algumas das informações nesta página podem estar desatualizadas.

❌ NÃO defina construtores internos públicos ou protegidos em tipos abstratos.

Os construtores devem ser públicos somente se os usuários precisarem criar instâncias do tipo. Como não é possível criar instâncias de um tipo abstrato, um tipo abstrato com um construtor público é projetado incorretamente e pode ser enganoso para os usuários.

✅ Defina um construtor protegido ou interno em classes abstratas.

Um construtor protegido é mais comum e simplesmente permite que a classe base faça a própria inicialização quando subtipos forem criados.

Um construtor interno pode ser usado para limitar implementações concretas da classe abstrata ao assembly que define a classe.

✅ Forneça pelo menos um tipo concreto herdado de cada classe abstrata enviada.

Isso ajuda a validar o design da classe abstrata. Por exemplo, [System.IO.FileStream](#) é uma implementação da classe abstrata [System.IO.Stream](#).

Portions © 2005, 2009 Microsoft Corporation. Todos os direitos reservados.

Reimpresso com permissão da Pearson Education, Inc. das [Diretrizes de Design do Framework: convenções, linguagens e padrões para bibliotecas do .NET reutilizável, 2ª edição](#) [↗] por Krzysztof Cwalina e Brad Abrams, publicado em 22 de outubro de 2008 por Addison-Wesley Professional como parte da série de desenvolvimento do Microsoft Windows.

Confira também

- [Diretrizes de Design de tipo](#)

- Diretrizes de design do Framework

Design de classe estática

Artigo • 12/03/2024

ⓘ Observação

Este conteúdo é reimpresso com permissão da Pearson Education, Inc. de *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition*. Essa edição foi publicada em 2008 e, desde então, o livro foi totalmente revisado [na terceira edição](#) [↗]. Algumas das informações nesta página podem estar desatualizadas.

Uma classe estática é definida como uma classe que contém apenas membros estáticos (é claro, além dos membros da instância herdados de [System.Object](#) e possivelmente de um construtor privado). Algumas linguagens dão suporte interno para classes estáticas. No C# 2.0 e posteriores, quando uma classe é declarada estática, ela é selada, abstrata e nenhum membro da instância pode ser substituído ou declarado.

As classes estáticas são um equilíbrio entre o design orientado a objetos puro e a simplicidade. Elas costumam ser usadas para fornecer atalhos para outras operações (como [System.IO.File](#)), detentores de métodos de extensão ou funcionalidade para a qual um wrapper completo orientado a objeto não se justifica (como [System.Environment](#)).

✓ USE classes estáticas com moderação.

Classes estáticas devem ser usadas apenas como classes de suporte para o núcleo orientado a objeto da estrutura.

✗ NÃO trate classes estáticas como um bucket diverso.

✗ NÃO declare nem substitua membros da instância em classes estáticas.

✓ Declare classes estáticas como seladas, abstratas e adicione um construtor de instância privada se a linguagem de programação não tiver suporte interno para classes estáticas.

Portions © 2005, 2009 Microsoft Corporation. Todos os direitos reservados.

Reimpresso com permissão da Pearson Education, Inc. das [Diretrizes de Design do Framework: convenções, linguagens e padrões para bibliotecas do .NET reutilizável, 2ª edição](#) [↗] por Krzysztof Cwalina e Brad Abrams, publicado em 22 de outubro de 2008 por

Addison-Wesley Professional como parte da série de desenvolvimento do Microsoft Windows.

Confira também

- [Diretrizes de Design de tipo](#)
- [Diretrizes de design do Framework](#)

Design de interface

Artigo • 12/03/2024

ⓘ Observação

Este conteúdo é reimpresso com permissão da Pearson Education, Inc. de *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition*. Essa edição foi publicada em 2008 e, desde então, o livro foi totalmente revisado [na terceira edição](#) [↗]. Algumas das informações nesta página podem estar desatualizadas.

Embora a maioria das APIs seja melhor modelada usando classes e structs, há casos em que as interfaces são mais apropriadas ou são a única opção.

O CLR não dá suporte a várias heranças (ou seja, as classes de CLR não podem herdar de mais de uma classe base), mas permite que os tipos implementem uma ou mais interfaces, além de herdar de uma classe base. Portanto, as interfaces geralmente são usadas para obter o efeito de várias heranças. Por exemplo, `IDisposable` é uma interface que permite que os tipos ofereçam suporte à eliminação, independentemente de qualquer outra hierarquia de herança na qual desejam participar.

A outra situação em que a definição de uma interface é apropriada está na criação de uma interface comum que pode ser suportada por vários tipos, incluindo alguns tipos de valor. Os tipos de valor não podem herdar de tipos diferentes de `ValueType`, mas podem implementar interfaces, portanto, usar uma interface é a única opção para fornecer um tipo base comum.

✓ DEFINA uma interface se você precisar que alguma API comum tenha suporte de um conjunto de tipos que incluem tipos de valor.

✓ CONSIDERE definir uma interface se você precisar dar suporte à sua funcionalidade em tipos que já herdam de algum outro tipo.

✗ EVITE o uso de interfaces de marcador (interfaces sem membros).

Se você precisar marcar uma classe como tendo uma característica específica (marcador), no geral, use um atributo personalizado em vez de uma interface.

✓ FORNEÇA pelo menos um tipo que é uma implementação de uma interface.

Fazer isso ajuda a validar o design da interface. Por exemplo, `List<T>` é uma implementação da interface `ICollection<T>`.

✓ FORNEÇA pelo menos uma API que consome cada interface que você define (um método que usa a interface como um parâmetro ou uma propriedade tipada como a interface).

Fazer isso ajuda a validar o design da interface. Por exemplo, `List<T>.Sort` consome a interface `System.Collections.Generic.IComparer<T>`.

✗ NÃO adicione membros a uma interface que tenha sido enviada anteriormente.

Isso interromperia as implementações da interface. Você deve criar uma nova interface para evitar problemas de controle de versão.

Exceto pelas situações descritas nessas diretrizes, você deve, no geral, escolher classes em vez de interfaces na criação de bibliotecas reutilizáveis de código gerenciado.

Portions © 2005, 2009 Microsoft Corporation. Todos os direitos reservados.

Reimpresso com permissão da Pearson Education, Inc. das [Diretrizes de Design do Framework: convenções, linguagens e padrões para bibliotecas do .NET reutilizável](#), 2ª edição [↗](#) por Krzysztof Cwalina e Brad Abrams, publicado em 22 de outubro de 2008 por Addison-Wesley Professional como parte da série de desenvolvimento do Microsoft Windows.

Confira também

- [Diretrizes de Design de tipo](#)
- [Diretrizes de design do Framework](#)

Design de Struct

Artigo • 06/10/2023

ⓘ Observação

Este conteúdo é reimpresso com permissão da Pearson Education, Inc., de *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition*. Essa edição foi publicada em 2008 e, desde então, o livro foi totalmente revisado na **terceira edição** [↗](#). Algumas das informações nesta página podem estar desatualizadas.

O tipo de valor de uso geral é geralmente chamado de struct, sua palavra-chave em C#. Esta seção fornece diretrizes para design geral de struct.

✗ NÃO forneça um construtor sem parâmetros para um struct.

Seguir essa diretriz permite que matrizes de structs sejam criadas sem precisar executar o construtor em cada item da matriz. Observe que o C# não permite que structs tenham construtores sem parâmetros.

✗ NÃO defina tipos mutáveis de valor.

Tipos de valor mutáveis têm vários problemas. Por exemplo, quando um getter de propriedade retorna um tipo de valor, o chamador recebe uma cópia. Como a cópia é criada implicitamente, os desenvolvedores podem não estar cientes de que estão alterando a cópia e não o valor original. Além disso, algumas linguagens (linguagens dinâmicas, em particular) têm problemas ao usar tipos de valor mutáveis porque até mesmo variáveis locais, quando desreferenciadas, fazem com que uma cópia seja feita.

✓ VERIFIQUE se um estado em que todos os dados da instância estão definidos como zero, falsos ou nulos (conforme apropriado) é válido.

Isso impede a criação acidental de instâncias inválidas quando uma matriz dos structs é criada.

✓ Implemente `IEquatable<T>` nos tipos de valor.

O método `Object.Equals` em tipos de valor causa conversão boxing e sua implementação padrão não é muito eficiente, pois usa reflexão. `Equals` pode ter um desempenho muito melhor e pode ser implementado para que ele não cause conversão boxing.

✗ NÃO estenda [ValueType](#) explicitamente. Na verdade, a maioria das linguagens impede isso.

Em geral, os structs podem ser muito úteis, mas só devem ser usados para valores pequenos, únicos e imutáveis que não serão usados com frequência.

Portions © 2005, 2009 Microsoft Corporation. Todos os direitos reservados.

Reimpresso com permissão da Pearson Education, Inc. das [Diretrizes de Design do Framework: convenções, linguagens e padrões para bibliotecas do .NET reutilizável](#), 2ª edição [↗](#) por Krzysztof Cwalina e Brad Abrams, publicado em 22 de outubro de 2008 por Addison-Wesley Professional como parte da série de desenvolvimento do Microsoft Windows.

Confira também

- [Diretrizes de Design de tipo](#)
- [Diretrizes de design do Framework](#)
- [Escolher entre Classe e Struct](#)

Design de enumeração

Artigo • 12/03/2024

ⓘ Observação

Este conteúdo é reimpresso com permissão da Pearson Education, Inc. de *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition*. Essa edição foi publicada em 2008 e, desde então, o livro foi totalmente revisado [na terceira edição](#) [↗]. Algumas das informações nesta página podem estar desatualizadas.

Enumerações são um tipo especial de valor. Há dois tipos de enumerações: enumerações simples e enumerações de sinalizador.

Enumerações simples representam pequenos conjuntos fechados de opções. Um exemplo comum da enumeração simples é um conjunto de cores.

As enumerações de sinalizador foram projetadas para dar suporte a operações bit a bit nos valores de enumeração. Um exemplo comum da enumeração de sinalizadores é uma lista de opções.

✓ USE uma enumeração para parâmetros, propriedades e valores retornados fortemente tipados que representam conjuntos de valores.

✓ PREFIRA usar uma enumeração, em vez de constantes estáticas.

✗ NÃO use uma enumeração para conjuntos abertos (como a versão do sistema operacional, nomes de amigos etc.).

✗ NÃO forneça valores de enumeração reservados destinados a uso futuro.

Você sempre pode simplesmente adicionar valores à enumeração em uma fase posterior. Confira [Como adicionar valores a enumerações](#) para mais detalhes sobre como adicionar valores a enumerações. Os valores reservados apenas poluem o conjunto de valores reais e tendem a levar a erros do usuário.

✗ EVITE expor publicamente enumerações com apenas um valor.

Uma prática comum para garantir a extensibilidade futura das APIs C é adicionar parâmetros reservados às assinaturas de método. Esses parâmetros reservados podem ser expressos como enumerações com um só valor padrão. Isso não deve ser feito em

APIs gerenciadas. A sobrecarga de método permite adicionar parâmetros em versões futuras.

✗ NÃO inclua valores sentinela em enumerações.

Embora às vezes sejam úteis para desenvolvedores de estrutura, os valores sentinela são confusos para os usuários da estrutura. Eles são usados para acompanhar o estado da enumeração, em vez de serem um dos valores do conjunto representado pela enumeração.

✓ FORNEÇA um valor de zero em enumerações simples.

Considere chamar o valor como "Nenhum". Se esse valor não for apropriado para essa enumeração específica, o valor padrão mais comum para a enumeração deverá receber o valor subjacente de zero.

✓ CONSIDERE usar `Int32` (o padrão na maioria das linguagens de programação) como o tipo subjacente de uma enumeração, a menos que qualquer um dos seguintes seja verdadeiro:

- A enumeração é uma enumeração de sinalizadores e você tem mais de 32 sinalizadores ou espera ter mais no futuro.
- O tipo subjacente precisa ser diferente de `Int32` para facilitar a interoperabilidade com código não gerenciado esperando enumerações de tamanhos diferentes.
- Um tipo subjacente menor resultaria em uma economia substancial de espaço. Se você espera que a enumeração seja usada principalmente como um argumento para o fluxo de controle, o tamanho fará pouca diferença. A economia de tamanho poderá ser significativa se:
 - Você espera que a enumeração seja usada como um campo em uma estrutura ou classe instanciada com muita frequência.
 - Você espera que os usuários criem grandes matrizes ou coleções das instâncias de enumeração.
 - Você espera que um grande número de instâncias da enumeração seja serializado.

Para uso na memória, lembre-se de que os objetos gerenciados são sempre alinhados por `DWORD`, portanto, você precisa efetivamente de várias enumerações ou outras estruturas pequenas em uma instância para empacotar uma enumeração menor para fazer a diferença, pois o tamanho total da instância sempre será arredondado para um `DWORD`.

✓ NOMEIE enumerações de sinalizador com substantivos plurais ou frases substantivas e enumerações simples com substantivos singulares ou frases substantivas.

✗ NÃO se estenda `System.Enum` diretamente.

`System.Enum` é um tipo especial usado pelo CLR para criar enumerações definidas pelo usuário. A maioria das linguagens de programação oferece um elemento de programação que dá acesso a essa funcionalidade. Por exemplo, em C#, a palavra-chave `enum` é usada para definir uma enumeração.

Como projetar enumerações de sinalizador

✓ APLIQUE `System.FlagsAttribute` para enumerações de sinalizador. Não aplique esse atributo a enumerações simples.

✓ USE poderes de dois para os valores de enumeração de sinalizador para que possam ser combinados livremente usando a operação OR bit a bit.

✓ CONSIDERE fornecer valores de enumeração especiais para combinações de sinalizadores comumente usadas.

As operações bit a bit são um conceito avançado e não devem ser necessárias para tarefas simples. `ReadWrite` é um exemplo desse valor especial.

✗ EVITE criar enumerações de sinalizador em que determinadas combinações de valores são inválidas.

✗ EVITE usar valores de enumeração de sinalizador de zero, a menos que o valor represente "todos os sinalizadores são limpos" e seja nomeado adequadamente, conforme prescrito pela próxima diretriz.

✓ NOMEIE o valor zero de enumerações de sinalizador `None`. Para uma enumeração de sinalizador, o valor deve sempre significar "todos os sinalizadores estão limpos".

Como adicionar valor a enumerações

É muito comum descobrir que você precisa adicionar valores a uma enumeração depois de já tê-la enviado. Há um possível problema de compatibilidade do aplicativo quando o valor que acaba de ser adicionado é retornado de uma API, pois aplicativos mal escritos podem não lidar corretamente com o novo valor.

✓ CONSIDERE adicionar valores a enumerações, apesar de um pequeno risco de compatibilidade.

Se você tiver dados reais sobre incompatibilidades de aplicativo causadas por adições a uma enumeração, considere adicionar uma API que retorne os valores novos e antigos e prefira a API antiga, que deve continuar retornando apenas os valores antigos. Isso garantirá que seus aplicativos continuem sendo compatíveis.

Portions © 2005, 2009 Microsoft Corporation. Todos os direitos reservados.

Reimpresso com permissão da Pearson Education, Inc. das [Diretrizes de Design do Framework: convenções, linguagens e padrões para bibliotecas do .NET reutilizável](#), 2ª edição [↗](#) por Krzysztof Cwalina e Brad Abrams, publicado em 22 de outubro de 2008 por Addison-Wesley Professional como parte da série de desenvolvimento do Microsoft Windows.

Confira também

- [Diretrizes de Design de tipo](#)
- [Diretrizes de design do Framework](#)

Tipos aninhados

Artigo • 06/10/2023

ⓘ Observação

Este conteúdo é reimpresso com permissão da Pearson Education, Inc., de *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition*. Essa edição foi publicada em 2008 e, desde então, o livro foi totalmente revisado na **terceira edição** [↗](#). Algumas das informações nesta página podem estar desatualizadas.

Um tipo aninhado é um tipo definido dentro do escopo de outro tipo, que é chamado de tipo de delimitador. Um tipo aninhado tem acesso a todos os membros do respectivo tipo de delimitador. Por exemplo, ele tem acesso a campos privados definidos no tipo de delimitador e a campos protegidos definidos em todos os ascendentes do tipo de delimitador.

Em geral, os tipos aninhados devem ser usados com moderação. Há vários motivos para isso. Alguns desenvolvedores não estão totalmente familiarizados com o conceito. Esses desenvolvedores podem, por exemplo, ter problemas com a sintaxe de declarar variáveis de tipos aninhados. Os tipos aninhados também são muito firmemente associados aos seus tipos de delimitador e, como tal, não são adequados como tipos de uso geral.

Os tipos aninhados são mais adequados para modelar detalhes de implementação de seus tipos de delimitador. O usuário final raramente tem que declarar variáveis de um tipo aninhado, e quase nunca deve ter que instanciar explicitamente tipos aninhados. Por exemplo, o enumerador de uma coleção pode ser um tipo aninhado dessa coleção. Os enumeradores geralmente são instanciados pelo tipo de conexão e, como muitas linguagens dão suporte à instrução `foreach`, as variáveis de enumerador raramente precisam ser declaradas pelo usuário final.

✓ USE tipos aninhados quando a relação entre o tipo aninhado e seu tipo externo for tal que a semântica de acessibilidade de membros seja desejável.

✗ NÃO use tipos aninhados públicos como um constructo de agrupamento lógico; use namespaces para isso.

✗ EVITE tipos aninhados expostos publicamente. A única exceção a isso é se variáveis do tipo aninhado precisarem ser declaradas apenas em cenários raros, como subclasse ou outros cenários avançados de personalização.

✗ NÃO use tipos aninhados se o tipo provavelmente for referenciado fora do tipo que o contém.

Por exemplo, uma enumeração passada para um método definido em uma classe não deve ser definida como um tipo aninhado na classe.

✗ NÃO use tipos aninhados se eles precisarem ser instanciados pelo código do cliente. Se um tipo tiver um construtor público, ele provavelmente não deve ser aninhado.

Se um tipo puder ser instanciado, isso parece indicar que o tipo tem um lugar na estrutura por conta própria (você pode criá-lo, trabalhar com ele e destruí-lo sem nunca usar o tipo externo) e, portanto, não deve ser aninhado. Os tipos internos não devem ser amplamente reutilizados fora do tipo externo sem qualquer relação com o tipo externo.

✗ NÃO defina um tipo aninhado como membro de uma interface. Há muitas linguagens que não dão suporte a esse constructo.

Portions © 2005, 2009 Microsoft Corporation. Todos os direitos reservados.

Reimpresso com permissão da Pearson Education, Inc. das [Diretrizes de Design do Framework: convenções, linguagens e padrões para bibliotecas do .NET reutilizável](#), 2ª edição [↗](#) por Krzysztof Cwalina e Brad Abrams, publicado em 22 de outubro de 2008 por Addison-Wesley Professional como parte da série de desenvolvimento do Microsoft Windows.

Confira também

- [Diretrizes de Design de tipo](#)
- [Diretrizes de design do Framework](#)

Diretrizes de design de membro

Artigo • 10/05/2023

Métodos, propriedades, eventos, construtores e campos são chamados coletivamente de membros. Os membros são, em última análise, os meios pelos quais a funcionalidade da estrutura é exposta aos usuários finais de uma estrutura.

Os membros podem ser virtuais ou não virtuais, concretos ou abstratos, estáticos ou de instância e podem ter vários escopos diferentes de acessibilidade. Toda essa variedade proporciona uma expressividade incrível, porém, ao mesmo tempo requer cuidado por parte do designer de estrutura.

Este capítulo apresenta as diretrizes básicas que devem ser seguidas ao criar membros de qualquer tipo.

Nesta seção

[Sobrecarga de membro](#)

[Design de propriedade](#)

[Design do construtor](#)

[Design de eventos](#)

[Design de campo](#)

[Métodos de Extensão](#)

[Sobrecargas de operador](#)

[Design de parâmetro](#)

Portions © 2005, 2009 Microsoft Corporation. Todos os direitos reservados.

Reimpresso com permissão da Pearson Education, Inc. das [Diretrizes de Design do Framework: convenções, linguagens e padrões para bibliotecas do .NET reutilizável](#), 2ª edição [↗](#) por Krzysztof Cwalina e Brad Abrams, publicado em 22 de outubro de 2008 por Addison-Wesley Professional como parte da série de desenvolvimento do Microsoft Windows.

Confira também

- [Diretrizes de design do Framework](#)

Sobrecarga de membro

Artigo • 07/10/2023

ⓘ Observação

Este conteúdo é reimpresso com permissão da Pearson Education, Inc. de *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition*. Essa edição foi publicada em 2008, e o livro desde então foi totalmente revisado na **terceira edição** [↗](#). Algumas das informações nesta página podem estar desatualizadas.

Sobrecarga de membro significa criar dois ou mais membros no mesmo tipo que diferem apenas no número ou no tipo de parâmetros, mas têm o mesmo nome. Por exemplo, a seguir, o método `WriteLine` é sobrecarregado:

C#

```
public static class Console {  
    public void WriteLine();  
    public void WriteLine(string value);  
    public void WriteLine(bool value);  
    ...  
}
```

Como somente métodos, construtores e propriedades indexadas podem ter parâmetros, somente esses membros podem ser sobrecarregados.

A sobrecarga é uma das técnicas mais importantes para melhorar a usabilidade, a produtividade e a legibilidade das bibliotecas reutilizáveis. A sobrecarga no número de parâmetros possibilita fornecer versões mais simples de construtores e métodos. A sobrecarga no tipo de parâmetro possibilita o uso do mesmo nome de membro para membros que executam operações idênticas em um conjunto selecionado de tipos diferentes.

✓ TENTE usar nomes de parâmetro descritivos para indicar o padrão usado por sobrecargas mais curtas.

✗ EVITE variar arbitrariamente nomes de parâmetro em sobrecargas. Se um parâmetro em uma sobrecarga representar a mesma entrada que um parâmetro em outra sobrecarga, os parâmetros deverão ter o mesmo nome.

❌ EVITE ser inconsistente na ordenação dos parâmetros em membros sobrecarregados. Parâmetros com o mesmo nome devem aparecer na mesma posição em todas as sobrecargas.

✅ FAÇA apenas a sobrecarga mais longa virtual (se a extensibilidade for necessária). Sobrecargas mais curtas devem simplesmente chamar uma sobrecarga mais longa.

❌ NÃO use os modificadores `ref` ou `out` para sobrecarregar membros.

Algumas linguagens não podem resolver chamadas para sobrecargas como esta. Além disso, essas sobrecargas geralmente têm semântica completamente diferente e provavelmente não devem ser sobrecargas, mas dois métodos separados.

❌ NÃO faça sobrecargas com parâmetros na mesma posição e tipos semelhantes ainda que com semântica diferente.

✅ PERMITA que `null` seja passado para argumentos opcionais.

✅ USE sobrecarga de membro em vez de definir membros com argumentos padrão.

Os argumentos padrão não são compatíveis com CLS.

Portions © 2005, 2009 Microsoft Corporation. Todos os direitos reservados.

Reimpresso com permissão da Pearson Education, Inc. das [Diretrizes de Design do Framework: convenções, linguagens e padrões para bibliotecas do .NET reutilizável](#), 2ª edição [↗](#) por Krzysztof Cwalina e Brad Abrams, publicado em 22 de outubro de 2008 por Addison-Wesley Professional como parte da série de desenvolvimento do Microsoft Windows.

Confira também

- [Diretrizes de design de membro](#)
- [Diretrizes de design do Framework](#)

Design de propriedade

Artigo • 07/10/2023

ⓘ Observação

Esse conteúdo é reimpresso por permissão da Pearson Education, Inc. das *Diretrizes de Design da Estrutura: Convenções, Idiomas e Padrões para Bibliotecas .NET Reutilizáveis*, 2ª Edição. Essa edição foi publicada em 2008, e desde então o livro foi totalmente revisado na **terceira edição** [↗](#). Algumas das informações nesta página podem estar desatualizadas.

Embora as propriedades sejam tecnicamente muito semelhantes aos métodos, elas são bem diferentes em termos de seus cenários de uso. Elas devem ser vistas como campos inteligentes. Elas têm a sintaxe de chamada dos campos e a flexibilidade dos métodos.

✓ CRIE propriedades somente get se o chamador não puder alterar o valor da propriedade.

Saiba que, se o tipo da propriedade for um tipo de referência mutável, o valor da propriedade poderá ser alterado mesmo que a propriedade seja somente get.

✗ NÃO forneça propriedades somente set ou propriedades com um setter que tenha acessibilidade mais ampla do que o getter.

Por exemplo, não use propriedades com um setter público e um getter protegido.

Se o getter da propriedade não puder ser fornecido, implemente a funcionalidade como um método. Considere iniciar o nome do método com `Set` e seguir com o que você teria usado como nome da propriedade. Por exemplo, `AppDomain` tem um método chamado `SetCachePath`, em vez de ter uma propriedade somente set chamada `CachePath`.

✓ FORNEÇA valores padrão concretos para todas as propriedades, garantindo que os padrões não resultem em uma brecha de segurança ou código extremamente ineficiente.

✓ PERMITA que as propriedades sejam definidas em qualquer ordem, mesmo que isso resulte em um estado temporário inválido do objeto.

É comum que duas ou mais propriedades estejam inter-relacionadas com um ponto em que alguns valores de uma propriedade podem ser inválidos, considerando os valores de outras propriedades no mesmo objeto. Nesses casos, exceções resultantes do estado

inválido devem ser adiadas até que as propriedades inter-relacionadas sejam de fato usadas juntas pelo objeto.

✓ PRESERVE o valor anterior se um setter de propriedade gerar uma exceção.

✗ EVITE gerar exceções de getters de propriedade.

Os getters de propriedade devem ser operações simples e não devem ter pré-condições. Se um getter puder gerar uma exceção, ele provavelmente deverá ser reprojetoado para ser um método. Observe que essa regra não se aplica aos indexadores, em que esperamos exceções como resultado da validação dos argumentos.

Design de propriedade indexada

Uma propriedade indexada é uma propriedade especial que pode ter parâmetros e ser chamada com sintaxe especial semelhante à indexação de matriz.

As propriedades indexadas são comumente conhecidas como indexadores. Os indexadores devem ser usados apenas em APIs que dão acesso a itens em uma coleção lógica. Por exemplo, uma cadeia de caracteres é uma coleção de caracteres e o indexador `System.String` foi adicionado para acessar seus caracteres.

✓ CONSIDERE usar indexadores para dar acesso aos dados armazenados em uma matriz interna.

✓ CONSIDERE fornecer indexadores em tipos que representam coleções de itens.

✗ EVITE usar propriedades indexadas com mais de um parâmetro.

Se o design exigir vários parâmetros, reconsidere se a propriedade realmente representa um acessador para uma coleção lógica. Caso não represente, use métodos. Considere iniciar o nome do método com `Get` ou `Set`.

✗ EVITE indexadores com tipos de parâmetro que não `System.Int32`, `System.Int64`, `System.String`, `System.Object` ou uma enumeração.

Se o design exigir outros tipos de parâmetros, avalie fortemente se a API realmente representa um acessador para uma coleção lógica. Se isso não acontecer, use um método. Considere iniciar o nome do método com `Get` ou `Set`.

✓ Use o nome `Item` para propriedades indexadas, a menos que haja um nome obviamente melhor (por exemplo, confira a propriedade `Chars` em `System.String`).

Em C#, os indexadores são, por padrão, chamados `Item`. O `IndexerNameAttribute` pode ser usado para personalizar esse nome.

✗ NÃO forneça um indexador e métodos semanticamente equivalentes.

✗ NÃO forneça mais de uma família de indexadores sobrecarregados em um tipo.

Isso é imposto pelo compilador C#.

✗ NÃO use propriedades indexadas não padrão.

Isso é imposto pelo compilador C#.

Eventos de notificação de alteração de propriedade

Às vezes, é útil fornecer um evento notificando o usuário de alterações em um valor da propriedade. Por exemplo, `System.Windows.Forms.Control` gera um evento `TextChanged` depois que o valor de sua propriedade `Text` é alterado.

✓ CONSIDERE gerar eventos de notificação de alteração quando os valores de propriedade em APIs de alto nível (geralmente componentes de designer) são modificados.

Se houver um bom cenário para um usuário saber quando uma propriedade de um objeto está mudando, o objeto deverá gerar um evento de notificação de alteração para a propriedade.

No entanto, é improvável que valha a pena a sobrecarga para gerar esses eventos para APIs de baixo nível, como tipos de base ou coleções. Por exemplo, `List<T>` não geraria esses eventos quando um novo item é adicionado à lista e a propriedade `Count` é alterada.

✓ CONSIDERE gerar eventos de notificação de alteração quando o valor de uma propriedade é alterado por forças externas.

Se um valor da propriedade for alterado por alguma força externa (de uma forma diferente de chamar métodos no objeto), gerar eventos indicará ao desenvolvedor que o valor está mudando e foi alterado. Um bom exemplo é a propriedade `Text` de um controle de caixa de texto. Quando o usuário digita texto em um `TextBox`, o valor da propriedade muda automaticamente.

Portions © 2005, 2009 Microsoft Corporation. Todos os direitos reservados.

Reimpresso com permissão da Pearson Education, Inc. das [Diretrizes de Design do Framework: convenções, linguagens e padrões para bibliotecas do .NET reutilizável](#), 2ª edição ↗ por Krzysztof Cwalina e Brad Abrams, publicado em 22 de outubro de 2008 por

Addison-Wesley Professional como parte da série de desenvolvimento do Microsoft Windows.

Confira também

- [Diretrizes de design de membro](#)
- [Diretrizes de design do Framework](#)

Design do construtor

Artigo • 07/10/2023

ⓘ Observação

Este conteúdo é reimpresso com permissão da Pearson Education, Inc. de *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition*. Essa edição foi publicada em 2008, e o livro desde então foi totalmente revisado na **terceira edição** [↗](#). Algumas das informações nesta página podem estar desatualizadas.

Há dois tipos de construtores: construtores de tipo e construtores de instância.

Os construtores de tipo são estáticos e são executados pelo CLR antes que o tipo seja usado. Os construtores de instância são executados quando uma instância de um tipo é criada.

Construtores de tipo não podem usar parâmetros. Os construtores de instância podem. Construtores de instância que não tomam parâmetros geralmente são chamados de construtores sem parâmetros.

Construtores são a maneira mais natural de criar instâncias de um tipo. A maioria dos desenvolvedores pesquisará e tentará usar um construtor antes de considerar formas alternativas de criar instâncias (como métodos de fábrica).

✓ CONSIDERE fornecer construtores simples, idealmente padrão.

Um construtor simples tem um número muito pequeno de parâmetros e todos os parâmetros são primitivos ou enumerações. Esses construtores simples aumentam a usabilidade da estrutura.

✓ CONSIDERE o uso de um método de fábrica estático em vez de um construtor se a semântica da operação desejada não for mapeada diretamente para a construção de uma nova instância ou se seguir as diretrizes de design do construtor não parecer natural.

✓ USE parâmetros de construtor como atalhos para definir as propriedades principais.

Não deve haver diferença na semântica entre o uso do construtor vazio seguido por alguns conjuntos de propriedades e o uso de um construtor com vários argumentos.

✓ USE o mesmo nome para parâmetros de construtor e uma propriedade se os parâmetros do construtor forem usados para simplesmente definir a propriedade.

A única diferença entre esses parâmetros e as propriedades deve ser o uso de maiúsculas e minúsculas.

✓ FAÇA um trabalho mínimo no construtor.

Os construtores não devem trabalhar muito além de capturar os parâmetros do construtor. O custo de qualquer outro processamento deve ser adiado até que seja necessário.

✓ GERE exceções de construtores de instância, se apropriado.

✓ DECLARE explicitamente o construtor público sem parâmetros em classes, se esse construtor for necessário.

Se você não declarar explicitamente nenhum construtor em um tipo, muitos idiomas (como C#) adicionarão automaticamente um construtor público sem parâmetros. (Classes abstratas obtêm um construtor protegido.)

Adicionar um construtor parametrizado a uma classe impede que o compilador adicione o construtor sem parâmetros. Isso geralmente causa alterações interruptivas acidentais.

✗ EVITE definir explicitamente construtores sem parâmetros em structs.

Isso torna a criação de matriz mais rápida, pois se o construtor sem parâmetros não for definido, ele não precisará ser executado em todos os slots da matriz. Observe que muitos compiladores, incluindo C#, não permitem que structs tenham construtores sem parâmetros por esse motivo.

✗ EVITE chamar membros virtuais em um objeto dentro de seu construtor.

Chamar um membro virtual fará com que a substituição mais derivada seja chamada, mesmo que o construtor do tipo mais derivado ainda não tenha sido totalmente executado.

Diretrizes do construtor de tipo

✓ TORNE os construtores estáticos privados.

Um construtor estático, também chamado de construtor de classe, é usado para inicializar um tipo. O CLR chama o construtor estático antes que a primeira instância do tipo seja criada ou que outros membros estáticos nesse tipo sejam chamados. O usuário não tem controle sobre quando o construtor estático é chamado. Se não for privado,

um construtor estático poderá ser chamado por um código diferente do CLR. Dependendo das operações realizadas no construtor, isso pode causar um comportamento inesperado. O compilador C# força os construtores estáticos a serem privados.

✗ NÃO lance exceções de construtores estáticos.

Se uma exceção for gerada de um construtor de tipo, o tipo não poderá ser usado no domínio do aplicativo atual.

✓ CONSIDERE inicializar campos estáticos embutidos em vez de usar explicitamente construtores estáticos, pois o runtime é capaz de otimizar o desempenho de tipos que não têm um construtor estático definido explicitamente.

Portions © 2005, 2009 Microsoft Corporation. Todos os direitos reservados.

Reimpresso com permissão da Pearson Education, Inc. das [Diretrizes de Design do Framework: convenções, linguagens e padrões para bibliotecas do .NET reutilizável](#), 2ª edição [↗](#) por Krzysztof Cwalina e Brad Abrams, publicado em 22 de outubro de 2008 por Addison-Wesley Professional como parte da série de desenvolvimento do Microsoft Windows.

Confira também

- [Diretrizes de design de membro](#)
- [Diretrizes de design do Framework](#)

Design de eventos

Artigo • 05/03/2024

ⓘ Observação

Este conteúdo é reimpresso com permissão da Pearson Education, Inc. de *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition*. Essa edição foi publicada em 2008 e, desde então, o livro foi totalmente revisado [na terceira edição](#) [↗]. Algumas das informações nesta página podem estar desatualizadas.

Os eventos são a forma mais usada de retornos de chamada (constructos que permitem que a estrutura chame o código do usuário). Outros mecanismos de retorno de chamada incluem membros que recebem representantes, membros virtuais e plug-ins baseados em interface. Dados de estudos de usabilidade indicam que a maioria dos desenvolvedores se sente mais confortável usando eventos do que outros mecanismos de retorno de chamada. Os eventos são bem integrados ao Visual Studio e a muitas linguagens.

É importante notar que existem dois grupos de eventos: eventos gerados antes de um estado do sistema mudar, chamados pré-eventos, e eventos gerados após uma alteração de estado, chamados pós-eventos. Um exemplo de pré-evento seria `Form.Closing`, que é gerado antes de um formulário ser encerrado. Um exemplo de pós-evento seria `Form.Closed`, que é gerado depois de um formulário ser encerrado.

- ✔ Use o termo "gerar" para eventos em vez de "lançar" ou "disparar".
- ✔ Use `System.EventHandler<TEventArgs>` em vez de criar manualmente novos representantes para serem usados como manipuladores de eventos.
- ✔ CONSIDERE usar uma subclasse de `EventArgs` como argumento do evento, a menos que você tenha certeza absoluta de que o evento nunca precisará carregar nenhum dado para o método de manipulação de eventos. Nesse caso, use diretamente o tipo `EventArgs`.

Ao enviar uma API usando `EventArgs` diretamente, não é possível adicionar dados que serão transportados com o evento sem interromper a compatibilidade. No caso de uma subclasse, mesmo que completamente vazia de início, é possível adicionar propriedades quando necessário.

✔ Use um método virtual protegido para gerar cada evento. Isso é aplicável somente a eventos não estáticos em classes não seladas, não a structs, classes seladas ou eventos estáticos.

O objetivo do método é fornecer uma maneira de uma classe derivada manipular o evento usando uma substituição. A substituição é uma maneira mais flexível, rápida e natural de lidar com eventos de classe base em classes derivadas. Por convenção, o nome do método deve começar com "On" e ser seguido pelo nome do evento.

A classe derivada pode optar por não chamar a implementação base do método na substituição. Para preparar-se para isso, não inclua nenhum processamento no método necessário a fim de que a classe base funcione corretamente.

✔ Defina um parâmetro para o método protegido que gera um evento.

O parâmetro deve ser nomeado `e` e escrito como a classe de argumento de evento.

✗ Não transmita nulo como remetente ao gerar um evento não estático.

✔ Transmita nulo como remetente ao gerar um evento estático.

✗ Não transmita nulo como o parâmetro de dados do evento ao gerar um evento.

Transmita `EventArgs.Empty` quando não quiser transmitir nenhum dado para o método de manipulação de eventos. Os desenvolvedores esperam que esse parâmetro não seja nulo.

✔ Considere gerar eventos que o usuário final possa cancelar. Isso só se aplica a pré-eventos.

Use [System.ComponentModel.CancelEventArgs](#) ou a subclasse respectiva como argumento de evento para permitir que o usuário final cancele eventos.

Design de manipulador de eventos personalizado

Há casos em que `EventHandler<T>` não pode ser usado, como quando a estrutura precisa trabalhar com versões anteriores do CLR, que não fornecem suporte a genéricos. Nesses casos, pode ser necessário projetar e desenvolver um representante de manipulador de eventos personalizado.

✔ Use um tipo de retorno nulo para manipuladores de eventos.

Um manipulador de eventos pode invocar diversos métodos de manipulação de eventos, possivelmente em diversos objetos. Se os métodos de manipulação de eventos

pudessem retornar um valor, haveria diversos valores de retorno para cada chamada de evento.

✓ Use `object` como o tipo do primeiro parâmetro do manipulador de eventos e chame-o de `sender`.

✓ Use `System.EventArgs` ou a respectiva subclasse como o tipo do segundo parâmetro do manipulador de eventos e chame-o de `e`.

✗ Não tenha mais de dois parâmetros em manipuladores de eventos.

Portions © 2005, 2009 Microsoft Corporation. Todos os direitos reservados.

Reimpresso com permissão da Pearson Education, Inc. das [Diretrizes de Design do Framework: convenções, linguagens e padrões para bibliotecas do .NET reutilizável](#), 2ª edição [↗](#) por Krzysztof Cwalina e Brad Abrams, publicado em 22 de outubro de 2008 por Addison-Wesley Professional como parte da série de desenvolvimento do Microsoft Windows.

Confira também

- [Diretrizes de design de membro](#)
- [Diretrizes de design do Framework](#)

Design de campo

Artigo • 05/03/2024

ⓘ Observação

Este conteúdo é reimpresso com permissão da Pearson Education, Inc. de *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition*. Essa edição foi publicada em 2008 e, desde então, o livro foi totalmente revisado [na terceira edição](#) [↗]. Algumas das informações nesta página podem estar desatualizadas.

O princípio do encapsulamento é uma das noções mais importantes no design orientado a objeto. Esse princípio afirma que os dados armazenados dentro de um objeto devem ser acessíveis somente para esse objeto.

Uma maneira útil de interpretar o princípio é dizer que um tipo deve ser criado para que as alterações em campos desse tipo (alterações de nome ou tipo) possam ser feitas sem quebrar o código que não seja para os membros do tipo. Essa interpretação imediatamente implica que todos os campos devem ser privados.

Excluimos campos constantes e estáticos somente leitura dessa restrição estrita, pois esses campos, praticamente por definição, nunca são necessários para serem alterados.

✗ NÃO forneça campos de instância públicos ou protegidos.

Você deve fornecer propriedades para acessar campos em vez de torná-los públicos ou protegidos.

✓ USE campos constantes para constantes que nunca serão alteradas.

O compilador queima os valores dos campos const diretamente no código de chamada. Portanto, os valores const nunca podem ser alterados sem o risco de quebrar a compatibilidade.

✓ USE campos estáticos públicos `readonly` para instâncias de objeto predefinidas.

Se houver instâncias predefinidas do tipo, declare-as como campos estáticos públicos somente leitura do próprio tipo.

✗ NÃO atribua instâncias de tipos mutáveis a campos `readonly`.

Um tipo mutável é um tipo com instâncias que podem ser modificadas depois de instanciadas. Por exemplo, as matrizes, a maioria das coleções e os fluxos são tipos

mutáveis, mas [System.Int32](#), [System.Uri](#) e [System.String](#) são todos imutáveis. O modificador somente leitura em um campo de tipo de referência impede que a instância armazenada no campo seja substituída, mas não impede que os dados da instância do campo sejam modificados chamando membros que alteram a instância.

Portions © 2005, 2009 Microsoft Corporation. Todos os direitos reservados.

Reimpresso com permissão da Pearson Education, Inc. das [Diretrizes de Design do Framework: convenções, linguagens e padrões para bibliotecas do .NET reutilizável](#), 2ª edição [↗](#) por Krzysztof Cwalina e Brad Abrams, publicado em 22 de outubro de 2008 por Addison-Wesley Professional como parte da série de desenvolvimento do Microsoft Windows.

Confira também

- [Diretrizes de design de membro](#)
- [Diretrizes de design do Framework](#)

Métodos de Extensão

Artigo • 07/10/2023

ⓘ Observação

Esse conteúdo é reimpresso por permissão da Pearson Education, Inc. das *Diretrizes de Design da Estrutura: Convenções, Idiomas e Padrões para Bibliotecas .NET Reutilizáveis, 2ª Edição*. Essa edição foi publicada em 2008, e desde então o livro foi totalmente revisado na **terceira edição** [↗](#). Algumas das informações nesta página podem estar desatualizadas.

Os métodos de extensão são um recurso de linguagem que permite chamar métodos estáticos usando a sintaxe de chamada de método de instância. Esses métodos devem usar pelo menos um parâmetro, que representa a instância em que o método deve operar.

A classe que define esses métodos de extensão é conhecida como a "patrocinadora" e deve ser declarada como estática. Para usar métodos de extensão, é necessário importar o namespace que define a classe patrocinadora.

✗ EVITE definir métodos de extensão de maneira frívola, especialmente em tipos que não são seus.

Se você detiver um código-fonte de um tipo, considere usar métodos de instância regulares. Se você não for o proprietário e quiser adicionar um método, tenha muito cuidado. O uso liberal de métodos de extensão pode desordenar APIs de tipos que não foram projetadas para ter esses métodos.

✓ CONSIDERE usar métodos de extensão em qualquer um dos seguintes cenários:

- Para fornecer funcionalidade auxiliar relevante para cada implementação de uma interface, se essa funcionalidade puder ser gravada nos termos da interface principal. Isso ocorre porque implementações concretas não podem ser atribuídas a interfaces. Por exemplo, os operadores `LINQ to Objects` são implementados como métodos de extensão para todos os tipos `IEnumerable<T>`. Assim, qualquer implementação `IEnumerable<>` é habilitada automaticamente para LINQ.
- Quando um método de instância introduziria uma dependência em algum tipo, mas essa dependência quebraria as regras de gerenciamento de dependência. Por exemplo, uma dependência de `String` a `System.Uri` provavelmente não é desejável, portanto, o método de instância `String.ToUri()` retornando `System.Uri` seria o

design errado de uma perspectiva de gerenciamento de dependências. Um método `Uri.ToUri(this string str)` de extensão estático retornando `System.Uri` seria um design muito melhor.

✗ EVITE definir métodos de extensão em `System.Object`.

Os usuários do VB não poderão chamar esses métodos em referências de objeto usando a sintaxe do método de extensão. O VB não dá suporte à chamada desses métodos porque, no VB, declarar uma referência como Objeto força que todas as invocações de método nele sejam associadas tardiamente (o membro real chamado é determinado em tempo de execução), enquanto as associações aos métodos de extensão são determinadas em tempo de compilação (limite antecipado).

Observe que a diretriz se aplica a outras linguagens em que o mesmo comportamento de associação está presente ou em que métodos de extensão não têm suporte.

✗ NÃO coloque métodos de extensão no mesmo namespace que o tipo estendido, a menos que seja para adicionar métodos a interfaces ou para gerenciamento de dependências.

✗ EVITE definir dois ou mais métodos de extensão com a mesma assinatura, mesmo que eles residam em namespaces diferentes.

✓ CONSIDERE definir métodos de extensão no mesmo namespace que o tipo estendido se o tipo for uma interface e se os métodos de extensão forem destinados a serem usados na maioria ou em todos os casos.

✗ NÃO defina métodos de extensão que implementam um recurso em namespaces normalmente associados a outros recursos. Em vez disso, defina-os no namespace associado ao recurso ao qual pertencem.

✗ EVITE a nomenclatura genérica de namespaces dedicados aos métodos de extensão (por exemplo, "Extensões"). Use um nome descritivo (por exemplo, "Roteamento") em vez disso.

Portions © 2005, 2009 Microsoft Corporation. Todos os direitos reservados.

Reimpresso com permissão da Pearson Education, Inc. das [Diretrizes de Design do Framework: convenções, linguagens e padrões para bibliotecas do .NET reutilizável](#), 2ª edição [↗](#) por Krzysztof Cwalina e Brad Abrams, publicado em 22 de outubro de 2008 por Addison-Wesley Professional como parte da série de desenvolvimento do Microsoft Windows.

Confira também

- [Diretrizes de design de membro](#)
- [Diretrizes de design do Framework](#)

Sobrecargas de operador

Artigo • 05/03/2024

ⓘ Observação

Este conteúdo é reimpresso com permissão da Pearson Education, Inc. de *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition*. Essa edição foi publicada em 2008 e, desde então, o livro foi totalmente revisado [na terceira edição](#) [↗]. Algumas das informações nesta página podem estar desatualizadas.

As sobrecargas de operador permitem que os tipos de estrutura apareçam como se fossem primitivos de linguagem internos.

Embora permitido e útil em algumas situações, as sobrecargas do operador devem ser usadas com cautela. Em muitos casos, houve abuso da sobrecarga de operador, como quando designers de estrutura começaram a usar operadores para operações que deveriam ser métodos simples. As diretrizes a seguir devem ajudá-lo a decidir quando e como usar a sobrecarga de operador.

✗ EVITE definir sobrecargas de operador, exceto em tipos que devem parecer tipos primitivos (internos).

✓ CONSIDERE definir sobrecargas de operador em um tipo que deve parecer um tipo primitivo.

Por exemplo, `System.String` tem `operator==` e `operator!=` definidos.

✓ DEFINA sobrecargas de operador em structs que representam números (como `System.Decimal`).

✗ NÃO seja engraçadinho ao definir sobrecargas de operador.

A sobrecarga de operador é útil nos casos em que é imediatamente óbvio qual será o resultado da operação. Por exemplo, faz sentido conseguir subtrair um `DateTime` de outro `DateTime` e obter um `TimeSpan`. No entanto, não é apropriado usar o operador de união lógica para unir duas consultas de banco de dados ou usar o operador shift para gravar em um fluxo.

✗ NÃO forneça sobrecargas de operador, a não ser que pelo menos um dos operandos seja do tipo que define a sobrecarga.

✓ SOBRECARREGUE operadores de maneira simétrica.

Por exemplo, se você sobrecarregar o `operator==`, também deverá sobrecarregar o `operator!=`. Da mesma forma, se você sobrecarregar o `operator<`, também deverá sobrecarregar o `operator>` e assim por diante.

✓ CONSIDERE fornecer métodos com nomes amigáveis que correspondem a cada operador sobrecarregado.

Muitas linguagens não dão suporte à sobrecarga de operador. Por esse motivo, é recomendável que os tipos que sobrecarregam os operadores incluam um método secundário com um nome específico do domínio apropriado que fornece funcionalidade equivalente.

A tabela a seguir contém uma lista de operadores e os nomes de método amigáveis correspondentes.

 Expandir a tabela

Símbolo do operador C#	Nome dos metadados	Nome amigável
N/A	<code>op_Implicit</code>	<code>To<TypeName>/From<TypeName></code>
N/A	<code>op_Explicit</code>	<code>To<TypeName>/From<TypeName></code>
<code>+ (binary)</code>	<code>op_Addition</code>	<code>Add</code>
<code>- (binary)</code>	<code>op_Subtraction</code>	<code>Subtract</code>
<code>* (binary)</code>	<code>op_Multiply</code>	<code>Multiply</code>
<code>/</code>	<code>op_Division</code>	<code>Divide</code>
<code>%</code>	<code>op_Modulus</code>	<code>Mod or Remainder</code>
<code>^</code>	<code>op_ExclusiveOr</code>	<code>Xor</code>
<code>& (binary)</code>	<code>op_BitwiseAnd</code>	<code>BitwiseAnd</code>
<code> </code>	<code>op_BitwiseOr</code>	<code>BitwiseOr</code>
<code>&&</code>	<code>op_LogicalAnd</code>	<code>And</code>
<code> </code>	<code>op_LogicalOr</code>	<code>Or</code>
<code>=</code>	<code>op_Assign</code>	<code>Assign</code>
<code><<</code>	<code>op_LeftShift</code>	<code>LeftShift</code>

Símbolo do operador C#	Nome dos metadados	Nome amigável
>>	op_RightShift	RightShift
N/A	op_SignedRightShift	SignedRightShift
N/A	op_UnsignedRightShift	UnsignedRightShift
==	op_Equality	Equals
!=	op_Inequality	Equals
>	op_GreaterThan	CompareTo
<	op_LessThan	CompareTo
>=	op_GreaterThanOrEqual	CompareTo
<=	op_LessThanOrEqual	CompareTo
*=	op_MultiplicationAssignment	Multiply
-=	op_SubtractionAssignment	Subtract
^=	op_ExclusiveOrAssignment	Xor
<<=	op_LeftShiftAssignment	LeftShift
%=	op_ModulusAssignment	Mod
+=	op_AdditionAssignment	Add
&=	op_BitwiseAndAssignment	BitwiseAnd
=	op_BitwiseOrAssignment	BitwiseOr
,	op_Comma	Comma
/=	op_DivisionAssignment	Divide
--	op_Decrement	Decrement
++	op_Increment	Increment
- (unary)	op_UnaryNegation	Negate
+ (unary)	op_UnaryPlus	Plus
~	op_OnesComplement	OnesComplement

Operador de sobrecarga ==

Sobrecarregar `operator ==` é bastante complicado. A semântica do operador precisa ser compatível com vários outros membros, como [Object.Equals](#).

Operadores de conversão

Os operadores de conversão são operadores unários que permitem a conversão de um tipo em outro. Os operadores devem ser definidos como membros estáticos no operando ou no tipo de retorno. Há dois tipos de operadores de conversão: implícito e explícito.

✗ NÃO forneça um operador de conversão se essa conversão não for claramente esperada pelos usuários finais.

✗ NÃO defina operadores de conversão fora do domínio de um tipo.

Por exemplo, [Int32](#), [Double](#) e [Decimal](#) são todos os tipos numéricos, enquanto [DateTime](#), não. Portanto, não deve haver nenhum operador de conversão para converter um `Double(long)` em um `DateTime`. Um construtor é preferido nesse caso.

✗ NÃO forneça um operador de conversão implícita se a conversão potencialmente tiver perda.

Por exemplo, não deve haver uma conversão implícita de `Double` em `Int32` porque `Double` tem um intervalo maior do que `Int32`. Um operador de conversão explícita pode ser fornecido mesmo que a conversão potencialmente tenha perda.

✗ NÃO gere exceções de conversões implícitas.

É muito difícil para os usuários finais entenderem o que está acontecendo, pois eles podem não estar cientes de que uma conversão está ocorrendo.

✓ GERE [System.InvalidCastException](#) se uma chamada para um operador de conversão resultar em uma conversão de perda e o contrato do operador não permitir conversões com perda.

Portions © 2005, 2009 Microsoft Corporation. Todos os direitos reservados.

Reimpresso com permissão da Pearson Education, Inc. das [Diretrizes de Design do Framework: convenções, linguagens e padrões para bibliotecas do .NET reutilizável](#), 2ª edição ↗ por Krzysztof Cwalina e Brad Abrams, publicado em 22 de outubro de 2008 por Addison-Wesley Professional como parte da série de desenvolvimento do Microsoft Windows.

Confira também

- [Diretrizes de design de membro](#)
- [Diretrizes de design do Framework](#)

Design de parâmetro

Artigo • 05/03/2024

ⓘ Observação

Este conteúdo é reimpresso com permissão da Pearson Education, Inc. de *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition*. Essa edição foi publicada em 2008 e, desde então, o livro foi totalmente revisado [na terceira edição](#) [↗]. Algumas das informações nesta página podem estar desatualizadas.

Esta seção oferece diretrizes amplas sobre o design de parâmetros, incluindo seções com diretrizes para verificar argumentos. Além disso, você deve consultar as diretrizes descritas em [Parâmetros de nomenclatura](#).

✓ Use o tipo de parâmetro menos derivado que fornece a funcionalidade exigida pelo membro.

Por exemplo, suponha que você queira criar um método que enumere uma coleção e imprima cada item no console. Esse método deve usar [IEnumerable](#) como o parâmetro, não [ArrayList](#) ou [IList](#), por exemplo.

✗ NÃO use parâmetros reservados.

Se mais entradas para um membro forem necessárias em alguma versão futura, uma nova sobrecarga poderá ser adicionada.

✗ NÃO tenha métodos expostos publicamente que usem ponteiros, matrizes de ponteiros ou matrizes multidimensionais como parâmetros.

É relativamente difícil usar ponteiros e matrizes multidimensionais corretamente. Em quase todos os casos, as APIs podem ser reprojatadas para evitar o uso desses tipos como parâmetros.

✓ Coloque todos os parâmetros `out` seguindo todos os valores e parâmetros `ref` (excluindo matrizes de parâmetros), mesmo que isso resulte em uma inconsistência na ordenação de parâmetros entre sobrecargas (consulte [Sobrecarga de membro](#)).

Os parâmetros `out` podem ser vistos como valores retornados extra e o agrupamento deles facilita o entendimento da assinatura do método.

✓ Seja consistente na nomeação dos parâmetros ao substituir membros ou implementar membros da interface.

Isso comunica melhor a relação entre os métodos.

Escolhendo entre parâmetros Enum e Boolean

✓ Use enumerações se um membro tiver dois ou mais parâmetros boolianos.

✗ NÃO use boolianos, a menos que você tenha certeza absoluta de que nunca haverá necessidade de mais de dois valores.

As enumerações dão a você algum espaço para adição futura de valores, mas você deve estar ciente de todas as implicações da adição de valores a enumerações, que estão descritas em [Design de enumeração](#).

✓ Considere o uso de boolianos para parâmetros de construtor que são realmente valores de dois estados e são usados para inicializar propriedades booleanas.

Validando argumentos

✓ Valide argumentos passados para membros públicos, protegidos ou explicitamente implementados. Gere [System.ArgumentException](#) ou uma de suas subclasses, se a validação falhar.

Observe que a validação real não precisa necessariamente acontecer no próprio membro público ou protegido. Ela pode acontecer em um nível inferior em alguma rotina privada ou interna. O ponto principal é que toda a área de superfície exposta aos usuários finais verifique os argumentos.

✓ Gere [ArgumentNullException](#) se um argumento nulo for passado e o membro não oferecer suporte a argumentos nulos.

✓ Valide parâmetros de enumeração.

Não suponha que os argumentos de enumeração estarão no intervalo definido pela enumeração. O CLR permite a conversão de qualquer valor inteiro em um valor de enumeração, ainda que o valor não esteja definido na enumeração.

✗ NÃO use [Enum.IsDefined](#) para verificações de intervalo de enumeração.

✓ Lembre-se de que os argumentos mutáveis podem ter sido alterados após a validação.

Se o membro for sensível à segurança, você será incentivado a fazer uma cópia e, em seguida, validar e processar o argumento.

Passagem de parâmetro

Na perspectiva de um designer de estrutura, há três grupos principais de parâmetros: parâmetros por valor, parâmetros `ref` e parâmetros `out`.

Quando um argumento é passado por um parâmetro por valor, o membro recebe uma cópia do argumento real passado. Se o argumento for um tipo de valor, uma cópia do argumento será colocada na pilha. Se o argumento for um tipo de referência, uma cópia da referência será colocada na pilha. Por padrão, as linguagens CLR mais populares, como C#, VB.NET e C++, passam os parâmetros por valor.

Quando um argumento é passado por um parâmetro `ref`, o membro recebe uma referência do argumento real passado. Se o argumento for um tipo de valor, uma referência ao argumento será colocada na pilha. Se o argumento for um tipo de referência, uma cópia da referência será colocada na pilha. Os parâmetros `Ref` podem ser usados para permitir que o membro modifique os argumentos passados pelo chamador.

Os parâmetros `Out` são semelhantes aos parâmetros `ref`, com algumas pequenas diferenças. O parâmetro é inicialmente considerado não atribuído e não pode ser lido no corpo do membro antes de receber algum valor. Além disso, deve ser atribuído algum valor ao parâmetro antes que o membro retorne.

✗ Evite usar os parâmetros `out` ou `ref`.

O uso dos parâmetros `out` ou `ref` requer experiência com ponteiros, compreensão das diferenças entre tipos de valor e tipos de referência e os métodos de tratamento com vários valores de retorno. Além disso, a diferença entre parâmetros `out` e `ref` não é amplamente compreendida. Os arquitetos do Framework criando para um público geral não devem esperar que os usuários se tornem proficientes em trabalhar com parâmetros `out` ou `ref`.

✗ Não passe tipos de referência por referência.

Há algumas exceções limitadas à regra, como um método que pode ser usado para trocar referências.

Membros com número variável de parâmetros

Os membros que podem usar um número variável de argumentos são expressos fornecendo um parâmetro de matriz. Por exemplo, [String](#) fornece o seguinte método:

C#

```
public class String {  
    public static string Format(string format, object[] parameters);  
}
```

Em seguida, um usuário pode chamar o método [String.Format](#), da seguinte maneira:

```
String.Format("File {0} not found in {1}", new object[] { filename, directory });
```

A adição da palavra-chave `params` C# a um parâmetro de matriz altera o parâmetro para um parâmetro de matriz `params` e fornece um atalho para criar uma matriz temporária.

C#

```
public class String {  
    public static string Format(string format, params object[] parameters);  
}
```

Isso permite que o usuário chame o método passando os elementos da matriz diretamente na lista de argumentos.

```
String.Format("File {0} not found in {1}", filename, directory);
```

Observe que a palavra-chave `params` só pode ser adicionada ao último parâmetro na lista de parâmetros.

✓ Considere a adição da palavra-chave `params` aos parâmetros de matriz se você espera que os usuários finais passem matrizes com um pequeno número de elementos. Se é esperado que muitos elementos sejam passados em cenários comuns, os usuários provavelmente não passarão esses elementos embutidos de e, portanto, a palavra-chave `params` não é necessária.

✗ Evite usar matrizes de `params` se o chamador quase sempre tiver a entrada já em uma matriz.

Por exemplo, membros com parâmetros de matriz de bytes quase nunca seriam chamados passando bytes individuais. Por esse motivo, os parâmetros de matriz de bytes no .NET Framework não usam a palavra-chave `params`.

❌ NÃO use matrizes de params se a matriz for modificada pelo membro que usa o parâmetro de matriz params.

Como muitos compiladores transformam os argumentos para o membro em uma matriz temporária no site de chamada, a matriz pode ser um objeto temporário e, portanto, todas as modificações nela serão perdidas.

✅ Considere o uso da palavra-chave params em uma sobrecarga simples, mesmo que uma sobrecarga mais complexa não possa usá-la.

Pergunte a si mesmo se os usuários valorizariam ter a matriz params em uma sobrecarga, mesmo que não estivesse em todas as sobrecargas.

✅ Tente ordenar parâmetros para possibilitar o uso da palavra-chave params.

✅ Considere fornecer sobrecargas especiais e caminhos de código para chamadas com um pequeno número de argumentos em APIs extremamente sensíveis ao desempenho.

Isso possibilita evitar a criação de objetos de matriz quando a API é chamada com um pequeno número de argumentos. Forme os nomes dos parâmetros tomando uma forma singular do parâmetro de matriz e adicionando um sufixo numérico.

Você só deve fazer isso se quiser diferenciar todo o caminho do código, não apenas criar uma matriz e chamar o método mais geral.

✅ Lembre-se que nulo pode ser passado como um argumento de matriz params.

Você deve validar que a matriz não é nula antes do processamento.

❌ NÃO use os métodos `varargs`, também conhecidos como reticências.

Algumas linguagens CLR, como C++, dão suporte a uma convenção alternativa para passar listas de parâmetros variáveis chamadas métodos `varargs`. A convenção não deve ser usada em estruturas, pois não está em conformidade com CLS.

Parâmetros de Ponteiro

Em geral, os ponteiros não devem aparecer na área de superfície pública de uma estrutura de código gerenciado bem projetada. Na maioria das vezes, os ponteiros devem ser encapsulados. No entanto, em alguns casos, os ponteiros são necessários por motivos de interoperabilidade e o uso de ponteiros nesses casos é apropriado.

✅ Forneça uma alternativa para qualquer membro que use um argumento de ponteiro, pois os ponteiros não são compatíveis com CLS.

✗ EVITE fazer uma verificação cara de argumentos de ponteiro.

✓ Siga convenções comuns relacionadas ao ponteiro ao criar membros com ponteiros.

Por exemplo, não é necessário passar o índice inicial, pois a aritmética de ponteiro simples pode ser usada para obter o mesmo resultado.

Portions © 2005, 2009 Microsoft Corporation. Todos os direitos reservados.

Reimpresso com permissão da Pearson Education, Inc. das [Diretrizes de Design do Framework: convenções, linguagens e padrões para bibliotecas do .NET reutilizável](#), 2ª edição [↗](#) por Krzysztof Cwalina e Brad Abrams, publicado em 22 de outubro de 2008 por Addison-Wesley Professional como parte da série de desenvolvimento do Microsoft Windows.

Confira também

- [Diretrizes de design de membro](#)
- [Diretrizes de design do Framework](#)

Designer voltado para extensibilidade

Artigo • 10/05/2023

Um aspecto importante da criação de uma estrutura é garantir que a extensibilidade dela tenha sido cuidadosamente considerada. Isso requer que você entenda os custos e benefícios associados a vários mecanismos de extensibilidade. Este capítulo ajuda você a decidir quais dos mecanismos de extensibilidade, como subclasse, eventos, membros virtuais, retornos de chamada e assim por diante, podem atender melhor aos requisitos da sua estrutura.

Há muitas maneiras de permitir a extensibilidade em estruturas. As opções variam de menos poderosos, mas menos caros, a muito poderosos, mas caros. Para qualquer requisito de extensibilidade, você deve escolher o mecanismo de extensibilidade menos caro que atenda aos requisitos. Tenha em mente que muitas vezes é possível adicionar mais extensibilidade posteriormente, mas você nunca pode retirá-la sem a introdução de alterações interruptivas.

Nesta seção

[Classes não seladas](#)

[Membros protegidos](#)

[Eventos e retornos de chamada](#)

[Membros virtuais](#)

[Abstrações \(tipos e interfaces abstratos\)](#)

[Classes base para implementar abstrações](#)

[Selar](#)

Portions © 2005, 2009 Microsoft Corporation. Todos os direitos reservados.

Reimpresso com permissão da Pearson Education, Inc. das [Diretrizes de Design do Framework: convenções, linguagens e padrões para bibliotecas do .NET reutilizável](#), 2ª edição [↗](#) por Krzysztof Cwalina e Brad Abrams, publicado em 22 de outubro de 2008 por Addison-Wesley Professional como parte da série de desenvolvimento do Microsoft Windows.

Confira também

- [Diretrizes de design do Framework](#)

Classes não seladas

Artigo • 05/03/2024

ⓘ Observação

Este conteúdo é reimpresso com permissão da Pearson Education, Inc. de *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition*. Essa edição foi publicada em 2008 e, desde então, o livro foi totalmente revisado [na terceira edição](#) [↗]. Algumas das informações nesta página podem estar desatualizadas.

As classes seladas não podem ser herdadas e impedem a extensibilidade. Por outro lado, as classes que podem ser herdadas são chamadas de classes não seladas.

✓ CONSIDERE usar classes não seladas sem membros virtuais ou protegidos adicionais como uma ótima maneira de fornecer extensibilidade barata, mas muito valorizada, a uma estrutura.

Os desenvolvedores geralmente querem herdar de classes não seladas para adicionar membros de conveniência, como construtores personalizados, novos métodos ou sobrecargas de método. Por exemplo, `System.Messaging.MessageQueue` não está selado, portanto, permite que os usuários criem filas personalizadas padrão para um caminho de fila específico ou adicionem métodos personalizados que simplificam a API para cenários específicos.

As classes não são seladas por padrão na maioria das linguagens de programação, e esse também é o padrão recomendado para a maioria das classes em estruturas. A extensibilidade proporcionada por tipos não selados é muito apreciada pelos usuários da estrutura e é relativamente barata de fornecer devido aos custos de teste relativamente baixos associados a tipos não selados.

Portions © 2005, 2009 Microsoft Corporation. Todos os direitos reservados.

Reimpresso com permissão da Pearson Education, Inc. das [Diretrizes de Design do Framework: convenções, linguagens e padrões para bibliotecas do .NET reutilizável, 2ª edição](#) [↗] por Krzysztof Cwalina e Brad Abrams, publicado em 22 de outubro de 2008 por Addison-Wesley Professional como parte da série de desenvolvimento do Microsoft Windows.

Confira também

- Diretrizes de design do Framework
- Designer voltado para extensibilidade
- Selar

Membros protegidos

Artigo • 10/10/2023

ⓘ Observação

Este conteúdo é reimpresso com permissão da Pearson Education, Inc. de *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition*. Essa edição foi publicada em 2008 e, desde então, o livro é totalmente revisado na **terceira edição** [↗](#). Algumas das informações nesta página podem estar desatualizadas.

Os membros protegidos por si só não fornecem nenhuma extensibilidade, mas podem tornar a extensibilidade por meio da subclasse mais poderosa. Eles podem ser usados para expor opções avançadas de personalização sem complicar desnecessariamente a interface pública principal.

Os designers de estrutura precisam ter cuidado com os membros protegidos porque o nome "protegido" pode dar uma falsa sensação de segurança. Qualquer pessoa é capaz de subclassificar uma classe não lacrada e acessar membros protegidos e, portanto, todas as mesmas práticas de codificação defensiva usadas para membros públicos se aplicam a membros protegidos.

- ✓ CONSIDERE o uso de membros protegidos para personalização avançada.
- ✓ TRATE membros protegidos em classes não seladas como públicos para fins de segurança, documentação e análise de compatibilidade.

Qualquer pessoa pode herdar de uma classe e acessar os membros protegidos.

Portions © 2005, 2009 Microsoft Corporation. Todos os direitos reservados.

Reimpresso com permissão da Pearson Education, Inc. das [Diretrizes de Design do Framework: convenções, linguagens e padrões para bibliotecas do .NET reutilizável, 2ª edição](#) [↗](#) por Krzysztof Cwalina e Brad Abrams, publicado em 22 de outubro de 2008 por Addison-Wesley Professional como parte da série de desenvolvimento do Microsoft Windows.

Confira também

- [Diretrizes de design do Framework](#)
- [Designer voltado para extensibilidade](#)

Eventos e retornos de chamada

Artigo • 07/10/2023

🚨 Observação

Este conteúdo é reimpresso com permissão da Pearson Education, Inc. de *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition*. Essa edição foi publicada em 2008, e o livro desde então foi totalmente revisado na **terceira edição** [↗](#). Algumas das informações nesta página podem estar desatualizadas.

Retornos de chamada são pontos de extensibilidade que permitem que uma estrutura retorne ao código do usuário por meio de um delegado. Esses delegados geralmente são passados para a estrutura por meio de um parâmetro de um método.

Os eventos são um caso especial de retornos de chamada que dão suporte a sintaxe conveniente e consistente para fornecer o delegado (um manipulador de eventos). Além disso, os designers e o preenchimento de declaração do Visual Studio ajudam no uso de APIs baseadas em eventos. (Confira [Design de Evento](#).)

- ✓ CONSIDERE usar retornos de chamada para permitir que os usuários forneçam código personalizado a ser executado pela estrutura.
- ✓ CONSIDERE usar eventos para permitir que os usuários personalizem o comportamento de uma estrutura sem necessidade de entender o design orientado ao objeto.
- ✓ PREFIRA eventos a retornos de chamada simples, pois eles são mais familiares para uma gama maior de desenvolvedores e são integrados ao preenchimento de declaração do Visual Studio.
- ✗ EVITE usar retornos de chamada em APIs sensíveis ao desempenho.
- ✓ USE os novos tipos `Func<...>`, `Action<...>` ou `Expression<...>` em vez de delegados personalizados ao definir APIs com retornos de chamada.

`Func<...>` e `Action<...>` representam delegados genéricos. `Expression<...>` representa definições de função que podem ser compiladas e depois invocadas em tempo de execução, mas também podem ser serializadas e passadas para processos remotos.

✓ MEÇA e entender as implicações de desempenho de usar `Expression<...>`, em vez de usar delegados `Func<...>` e `Action<...>`.

Os tipos `Expression<...>` são, na maioria dos casos, logicamente equivalentes a delegados `Func<...>` e `Action<...>`. A principal diferença entre eles é que os delegados devem ser usados em cenários de processo local; expressões são destinadas a casos em que é benéfico e possível avaliar a expressão em um processo remoto ou computador.

✓ Entenda que, ao chamar um delegado, você está executando código arbitrário, o que pode ter repercussões de segurança, correção e compatibilidade.

Portions © 2005, 2009 Microsoft Corporation. Todos os direitos reservados.

Reimpresso com permissão da Pearson Education, Inc. das [Diretrizes de Design do Framework: convenções, linguagens e padrões para bibliotecas do .NET reutilizável](#), 2ª edição [↗](#) por Krzysztof Cwalina e Brad Abrams, publicado em 22 de outubro de 2008 por Addison-Wesley Professional como parte da série de desenvolvimento do Microsoft Windows.

Confira também

- [Designer voltado para extensibilidade](#)
- [Diretrizes de design do Framework](#)

Membros virtuais

Artigo • 05/03/2024

ⓘ Observação

Este conteúdo é reimpresso com permissão da Pearson Education, Inc. de *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition*. Essa edição foi publicada em 2008 e, desde então, o livro foi totalmente revisado [na terceira edição](#) [↗]. Algumas das informações nesta página podem estar desatualizadas.

Os membros virtuais podem ser substituídos, alterando o comportamento da subclasse. Eles são bastante semelhantes aos retornos de chamada em termos da extensibilidade que fornecem, mas são melhores em termos de desempenho de execução e consumo de memória. Além disso, os membros virtuais são mais naturais em cenários que exigem a criação de um tipo especial de um tipo existente (especialização).

Os membros virtuais têm um desempenho melhor do que retornos de chamada e eventos, mas não têm um desempenho melhor do que os métodos não virtuais.

A principal desvantagem dos membros virtuais é que o comportamento de um membro virtual só pode ser modificado no momento da compilação. O comportamento de um retorno de chamada pode ser modificado em tempo de execução.

Membros virtuais, como retornos de chamada (e talvez mais do que retornos de chamada), são caros de projetar, testar e manter porque qualquer chamada a um membro virtual pode ser substituída de maneiras imprevisíveis e pode executar código arbitrário. Além disso, geralmente é necessário muito mais esforço para definir claramente o contrato de membros virtuais, portanto, o custo de projetá-los e documentá-los é maior.

✗ NÃO crie membros virtuais, a menos que você tenha um bom motivo para fazer isso e esteja ciente de todos os custos relacionados a criação, teste e manutenção de membros virtuais.

Os membros virtuais são menos indulgentes em termos de alterações que podem ser feitas a eles sem interromper a compatibilidade. Além disso, são mais lentos do que membros não virtuais, principalmente porque as chamadas para membros virtuais não são embutidas.

✓ CONSIDERE limitar a extensibilidade a apenas o absolutamente necessário.

✓ PREFIRA a acessibilidade protegida à acessibilidade pública para membros virtuais. Os membros públicos devem fornecer extensibilidade (se necessário) chamando um membro virtual protegido.

Os membros públicos de uma classe devem fornecer o conjunto certo de funcionalidades para consumidores diretos dessa classe. Os membros virtuais foram projetados para serem substituídos em subclasses, e a acessibilidade protegida é uma ótima forma de definir o escopo de todos os pontos de extensibilidade virtual para onde eles podem ser usados.

Portions © 2005, 2009 Microsoft Corporation. Todos os direitos reservados.

Reimpresso com permissão da Pearson Education, Inc. das [Diretrizes de Design do Framework: convenções, linguagens e padrões para bibliotecas do .NET reutilizável](#), 2ª edição [↗](#) por Krzysztof Cwalina e Brad Abrams, publicado em 22 de outubro de 2008 por Addison-Wesley Professional como parte da série de desenvolvimento do Microsoft Windows.

Confira também

- [Diretrizes de design do Framework](#)
- [Designer voltado para extensibilidade](#)

Abstrações (tipos e interfaces abstratos)

Artigo • 05/03/2024

ⓘ Observação

Este conteúdo é reimpresso com permissão da Pearson Education, Inc. de *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition*. Essa edição foi publicada em 2008 e, desde então, o livro foi totalmente revisado [na terceira edição](#) [↗]. Algumas das informações nesta página podem estar desatualizadas.

Uma abstração é um tipo que descreve um contrato, mas não fornece uma implementação completa do contrato. As abstrações geralmente são implementadas como classes abstratas ou interfaces e vêm com um conjunto bem definido de documentação de referência que descreve a semântica necessária dos tipos que implementam o contrato. Algumas das abstrações mais importantes no .NET Framework incluem [Stream](#), [IEnumerable<T>](#) e [Object](#).

Você pode estender estruturas implementando um tipo concreto que dá suporte ao contrato de uma abstração e usando esse tipo concreto com APIs de estrutura consumindo (operando) a abstração.

É muito difícil projetar uma abstração significativa e útil que seja capaz de resistir ao teste do tempo. A principal dificuldade é obter o conjunto certo de membros, nem mais nem menos. Se uma abstração tiver muitos membros, será difícil ou até impossível implementar. Se ele tiver poucos membros para a funcionalidade prometida, ele se tornará inútil em muitos cenários interessantes.

Muitas abstrações em uma estrutura também afetam negativamente a usabilidade da estrutura. Por vezes, é muito difícil entender uma abstração sem entender como ela se encaixa no panorama geral das implementações concretas e das APIs que operam na abstração. Além disso, os nomes das abstrações e seus membros são necessariamente abstratos, o que muitas vezes os torna enigmáticos e inacessíveis sem antes entender o contexto mais amplo de seu uso.

No entanto, as abstrações fornecem uma extensibilidade extremamente poderosa que os outros mecanismos de extensibilidade geralmente podem não corresponder. Eles estão no centro de muitos padrões de arquitetura, como plug-ins, inversão de controle (IoC), pipelines e assim por diante. Eles também são extremamente importantes para testar a capacidade das estruturas. Boas abstrações possibilitam a eliminação de

dependências pesadas para fins de teste de unidade. Em resumo, as abstrações são responsáveis pela tão procurada riqueza das estruturas modernas orientadas a objetos.

✗ NÃO forneça abstrações, a menos que sejam testadas desenvolvendo várias implementações concretas e APIs que consumam as abstrações.

✓ ESCOLHA cuidadosamente entre uma classe abstrata e uma interface ao criar uma abstração.

✓ CONSIDERE fornecer testes de referência para implementações concretas de abstrações. Esses testes devem permitir que os usuários testem se suas implementações implementam corretamente o contrato.

Portions © 2005, 2009 Microsoft Corporation. Todos os direitos reservados.

Reimpresso com permissão da Pearson Education, Inc. das [Diretrizes de Design do Framework: convenções, linguagens e padrões para bibliotecas do .NET reutilizável](#), 2ª edição [↗](#) por Krzysztof Cwalina e Brad Abrams, publicado em 22 de outubro de 2008 por Addison-Wesley Professional como parte da série de desenvolvimento do Microsoft Windows.

Confira também

- [Diretrizes de design do Framework](#)
- [Designer voltado para extensibilidade](#)

Classes base para implementar abstrações

Artigo • 12/03/2024

ⓘ Observação

Este conteúdo é reimpresso com permissão da Pearson Education, Inc. de *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition*. Essa edição foi publicada em 2008 e, desde então, o livro foi totalmente revisado [na terceira edição](#) [↗]. Algumas das informações nesta página podem estar desatualizadas.

Estritamente falando, uma classe se torna uma classe base quando outra classe é derivada dela. Para a finalidade desta seção, no entanto, uma classe base é uma classe projetada principalmente para fornecer uma abstração comum ou para outras classes reutilizarem alguma implementação padrão por herança. As classes base geralmente ficam no meio de hierarquias de herança, entre uma abstração na raiz de uma hierarquia e várias implementações personalizadas na parte inferior.

Elas servem como auxiliares de implementação para implementar abstrações. Por exemplo, uma das abstrações da Framework para coleções ordenadas de itens é a interface `IList<T>`. A implementação `IList<T>` não é trivial e, portanto, a Estrutura fornece várias classes base, como `Collection<T>` e `KeyedCollection<TKey,TItem>`, que servem como auxiliares para implementar coleções personalizadas.

As classes base geralmente não são adequadas para servir como abstrações sozinhas, pois tendem a conter muita implementação. Por exemplo, a classe base `Collection<T>` contém muita implementação relacionada ao fato de implementar a interface não genérica `IList` (para integrar melhor com coleções não genéricas) e ao fato de ser uma coleção de itens armazenados na memória em um de seus campos.

Como já discutido, as classes base podem fornecer ajuda inestimável para os usuários que precisam implementar abstrações, mas, ao mesmo tempo, podem ser uma responsabilidade significativa. Elas adicionam área de superfície e aumentam a profundidade das hierarquias de herança, portanto, complicam conceitualmente a estrutura. Assim, as classes base só deverão ser usadas se fornecerem um valor significativo para os usuários da estrutura. Elas devem ser evitadas se fornecerem valor somente aos implementadores da estrutura, nesse caso, a delegação para uma

implementação interna, em vez de herança de uma classe base deve ser fortemente considerada.

✓ CONSIDERE tornar as classes base abstratas mesmo que não contenham membros abstratos. Isso comunica claramente aos usuários que a classe foi projetada somente para ser herdada.

✓ CONSIDERE colocar classes base em um namespace separado dos tipos de cenário principais. Por definição, as classes base se destinam a cenários avançados de extensibilidade, portanto, não são interessantes para a maioria dos usuários.

✗ EVITE nomear classes base com um sufixo "Base" se a classe for destinada a ser usada em APIs públicas.

Portions © 2005, 2009 Microsoft Corporation. Todos os direitos reservados.

Reimpresso com permissão da Pearson Education, Inc. das [Diretrizes de Design do Framework: convenções, linguagens e padrões para bibliotecas do .NET reutilizável](#), 2ª edição [↗](#) por Krzysztof Cwalina e Brad Abrams, publicado em 22 de outubro de 2008 por Addison-Wesley Professional como parte da série de desenvolvimento do Microsoft Windows.

Confira também

- [Diretrizes de design do Framework](#)
- [Designer voltado para extensibilidade](#)

ⓘ Observação

Este conteúdo é reimpresso com permissão da Pearson Education, Inc. de *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition*. Essa edição foi publicada em 2008 e, desde então, o livro foi totalmente revisado [na terceira edição](#) [↗]. Algumas das informações nesta página podem estar desatualizadas.

Um dos recursos das estruturas orientadas a objetos é que os desenvolvedores podem estendê-las e personalizá-las de maneiras inesperadas pelos designers de estrutura. Esse é o poder e o risco do design extensível. Assim, ao projetar sua estrutura, é muito importante ter o cuidado de projetá-la para a extensibilidade quando ela for desejada e limitar essa extensibilidade quando ela se tornar arriscada.

Um mecanismo poderoso que impede a extensibilidade é a selagem. Você pode selar a classe ou membros individuais. Selar uma classe impede que os usuários herdem da classe. Selar um membro impede que os usuários substituam um membro específico.

✗ NÃO faça a selagem de classes sem ter um bom motivo para fazer isso.

Selar uma classe porque você não pode pensar em um cenário de extensibilidade não é um bom motivo. Os usuários da estrutura gostam de herdar de classes por vários motivos não óbvios, como adicionar membros de conveniência. Consulte [Classes não seladas](#) para obter exemplos de motivos não óbvios que os usuários desejam herdar de um tipo.

Os bons motivos para selar uma classe incluem os seguintes:

- A classe é estática. Consulte [Design de classe estática](#).
- A classe armazena segredos confidenciais de segurança em membros protegidos herdados.
- A classe herda muitos membros virtuais e o custo de selá-los individualmente superaria os benefícios de deixar a classe não selada.
- A classe é um atributo que requer uma pesquisa de runtime muito rápida. Os atributos selados têm níveis de desempenho ligeiramente mais altos do que os não selados. Consulte [Atributos](#).

✗ NÃO declare membros protegidos ou virtuais em tipos selados.

Por definição, um tipo selado não pode ser herdado. Isso significa que os membros protegidos nos tipos selados não podem ser chamados e os métodos virtuais nos tipos selados não podem ser substituídos.

✓ CONSIDERE a selagem de membros que você substitui.

Problemas que podem resultar da introdução de membros virtuais (discutidos em [Membros Virtuais](#)) também se aplicam a substituições, embora em um grau ligeiramente menor. Selar uma substituição protege você desses problemas a partir desse ponto na hierarquia de herança.

Portions © 2005, 2009 Microsoft Corporation. Todos os direitos reservados.

Reimpresso com permissão da Pearson Education, Inc. das [Diretrizes de Design do Framework: convenções, linguagens e padrões para bibliotecas do .NET reutilizável](#), 2ª edição [↗](#) por Krzysztof Cwalina e Brad Abrams, publicado em 22 de outubro de 2008 por Addison-Wesley Professional como parte da série de desenvolvimento do Microsoft Windows.

Confira também

- [Diretrizes de design do Framework](#)
- [Designer voltado para extensibilidade](#)
- [Classes não seladas](#)

Diretrizes de design para exceções

Artigo • 10/05/2023

O tratamento de exceções tem muitas vantagens em relação ao relatório de erros baseado em valor retornado. Um bom design de estrutura ajuda o desenvolvedor de aplicativos a perceber os benefícios das exceções. Esta seção discute os benefícios das exceções e apresenta diretrizes para usá-las com eficiência.

Nesta seção

[Gerando exceções](#)

[Usar tipos de exceção padrão](#)

[Exceções e desempenho](#)

Portions © 2005, 2009 Microsoft Corporation. Todos os direitos reservados.

Reimpresso com permissão da Pearson Education, Inc. das [Diretrizes de Design do Framework: convenções, linguagens e padrões para bibliotecas do .NET reutilizável](#), 2ª edição [↗](#) por Krzysztof Cwalina e Brad Abrams, publicado em 22 de outubro de 2008 por Addison-Wesley Professional como parte da série de desenvolvimento do Microsoft Windows.

Confira também

- [Diretrizes de design do Framework](#)

Gerar exceções

Artigo • 07/10/2023

ⓘ Observação

Este conteúdo é reimpresso com permissão da Pearson Education, Inc., de *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition*. Essa edição foi publicada em 2008 e, desde então, o livro foi totalmente revisado na **terceira edição** [↗](#). Algumas das informações nesta página podem estar desatualizadas.

As diretrizes de geração de exceção descritas nesta seção exigem uma boa definição do significado da falha de execução. A falha de execução ocorre sempre que um membro não pode fazer o que foi projetado para fazer (o que o nome do membro implica). Por exemplo, se o método `OpenFile` não puder retornar um identificador de arquivo aberto ao chamador, ele será considerado uma falha de execução.

A maioria dos desenvolvedores se sente confortável em usar exceções para erros de uso, como divisão por zero ou referências nulas. No Framework, as exceções são usadas para todas as condições de erro, incluindo erros de execução.

✗ NÃO retorne códigos de erro.

Exceções são o principal meio de relatar erros em estruturas.

✓ RELATE relata falhas de execução gerando exceções.

✓ CONSIDERE encerrar o processo chamando `System.Environment.FailFast` (.NET Framework recurso 2.0), em vez de gerar uma exceção, se o código encontrar uma situação em que não seja seguro para execução adicional.

✗ NÃO use exceções para o fluxo normal de controle, se possível.

Exceto por falhas e operações do sistema com possíveis condições de corrida, os designers de estrutura devem criar APIs para que os usuários possam escrever código que não gere exceções. Por exemplo, você pode fornecer um modo de verificar pré-condições antes de chamar um membro para que os usuários possam escrever código que não gere exceções.

O membro usado para verificar pré-condições de outro membro é frequentemente chamado de testador; o membro que realmente faz o trabalho é chamado de executor.

Há casos em que o padrão Testador-Executor pode ter uma sobrecarga de desempenho inaceitável. Nesses casos, o chamado padrão de Try-Parse deve ser considerado (confira [Exceções e Desempenho](#) para mais informações).

✓ CONSIDERE as implicações de desempenho de gerar exceções. As taxas de geração acima de 100 por segundo provavelmente afetarão visivelmente o desempenho da maioria dos aplicativos.

✓ DOCUMENTE todas as exceções geradas por membros publicamente chamáveis devido a uma violação do contrato de membro (em vez de uma falha do sistema) e trate-as como parte de seu contrato.

As exceções que fazem parte do contrato não devem ser alteradas de uma versão para a próxima (ou seja, o tipo de exceção não deve ser alterado e novas exceções não devem ser adicionadas).

✗ NÃO tem membros públicos que possam gerar ou não com base em alguma opção.

✗ NÃO tenha membros públicos que retornem exceções como o valor retornado ou um parâmetro `out`.

Retornar exceções de APIs públicas em vez de gerá-las anula muitos dos benefícios do relatório de erros baseado em exceção.

✓ CONSIDERE usar métodos do construtor de exceções.

É comum gerar a mesma exceção de locais diferentes. Para evitar bloat de código, use métodos auxiliares que criam exceções e inicializam suas propriedades.

Além disso, os membros que geram exceções não estão sendo embutidos. Mover a instrução `throw` dentro do construtor pode permitir que o membro seja embutido.

✗ NÃO gere exceções de blocos de filtro de exceção.

Quando um filtro de exceção gera uma exceção, a exceção é capturada pelo CLR e o filtro retorna `false`. Esse comportamento é indistinguível do filtro executando e retornando `false` explicitamente, portanto, é muito difícil de depurar.

✗ EVITE gerar explicitamente exceções de blocos `finally`. Exceções geradas implicitamente resultantes de métodos de chamada que geram são aceitáveis.

Portions © 2005, 2009 Microsoft Corporation. Todos os direitos reservados.

Reimpresso com permissão da Pearson Education, Inc. das [Diretrizes de Design do Framework: convenções, linguagens e padrões para bibliotecas do .NET reutilizável](#), 2ª edição ↗ por Krzysztof Cwalina e Brad Abrams, publicado em 22 de outubro de 2008 por

Addison-Wesley Professional como parte da série de desenvolvimento do Microsoft Windows.

Confira também

- [Diretrizes de design do Framework](#)
- [Diretrizes de design para exceções](#)

Usar tipos de exceção padrão

Artigo • 07/10/2023

❗ Observação

Esse conteúdo é reimpresso por permissão da Pearson Education, Inc. das *Diretrizes de Design da Estrutura: Convenções, Idiomas e Padrões para Bibliotecas .NET Reutilizáveis*, 2ª Edição. Essa edição foi publicada em 2008, e desde então o livro foi totalmente revisado na **terceira edição** [↗](#). Algumas das informações nesta página podem estar desatualizadas.

Esta seção descreve as exceções padrão fornecidas pelo Framework e os detalhes de seu uso. A lista não está completa de forma alguma. Consulte a documentação de referência do .NET Framework para uso de outros tipos de exceção do Framework.

Exceção e SystemException

❌ NÃO lance `System.Exception` ou `System.SystemException`.

❌ NÃO capture `System.Exception` ou `System.SystemException` no código da estrutura, a menos que você pretenda relançar.

❌ EVITE capturar `System.Exception` ou `System.SystemException`, exceto em manipuladores de exceção de nível superior.

ApplicationException

❌ NÃO lance ou derive de `ApplicationException`.

InvalidOperationException

✅ LANCE um `InvalidOperationException` se o objeto estiver em um estado inadequado.

ArgumentException, ArgumentNullException e ArgumentOutOfRangeException

✅ LANCE `ArgumentException` ou um de seus subtipos se argumentos ruins forem passados para um membro. Prefira o tipo de exceção mais derivado, se aplicável.

✓ DEFINA a propriedade `ParamName` ao lançar uma das subclasses de `ArgumentException`.

Esta propriedade representa o nome do parâmetro que fez com que a exceção fosse lançada. Observe que a propriedade pode ser definida usando uma das sobrecargas do construtor.

✓ USE `value` para o nome do parâmetro de valor implícito dos setters de propriedade.

NullReferenceException, IndexOutOfRangeException e AccessViolationException

✗ NÃO permita que APIs de chamada pública lancem de forma explícita ou implícita [NullReferenceException](#), [AccessViolationException](#) ou [IndexOutOfRangeException](#). Essas exceções são reservadas e lançadas pelo mecanismo de execução e na maioria dos casos indicam um bug.

Faça a verificação de argumentos para evitar lançar essas exceções. Lançar essas exceções expõe detalhes de implementação do seu método que podem mudar com o tempo.

StackOverflowException

✗ NÃO lance [StackOverflowException](#) explicitamente. A exceção deve ser lançada explicitamente apenas pelo CLR.

✗ NÃO capture `StackOverflowException`.

É quase impossível escrever código gerenciado que permaneça consistente na presença de excedentes de pilha arbitrários. As partes não gerenciadas do CLR permanecem consistentes usando testes para mover excedentes de pilha para locais bem definidos, em vez de recuar de excedentes de pilha arbitrários.

OutOfMemoryException

✗ NÃO lance [OutOfMemoryException](#) explicitamente. Essa exceção deve ser lançada apenas pela infraestrutura CLR.

ComException, SEHException e ExecutionEngineException

✗ NÃO lance [COMException](#), [ExecutionEngineException](#) e [SEHException](#) explicitamente. Essas exceções devem ser lançadas apenas pela infraestrutura CLR.

Portions © 2005, 2009 Microsoft Corporation. Todos os direitos reservados.

Reimpresso com permissão da Pearson Education, Inc. das [Diretrizes de Design do Framework: convenções, linguagens e padrões para bibliotecas do .NET reutilizável](#), 2ª edição [↗](#) por Krzysztof Cwalina e Brad Abrams, publicado em 22 de outubro de 2008 por Addison-Wesley Professional como parte da série de desenvolvimento do Microsoft Windows.

Confira também

- [Diretrizes de design do Framework](#)
- [Diretrizes de design para exceções](#)

Exceções e desempenho

Artigo • 12/03/2024

ⓘ Observação

Este conteúdo é reimpresso com permissão da Pearson Education, Inc. de *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition*. Essa edição foi publicada em 2008 e, desde então, o livro foi totalmente revisado [na terceira edição](#) [↗]. Algumas das informações nesta página podem estar desatualizadas.

Uma preocupação comum relacionada a exceções é que, se exceções forem usadas para código que falha rotineiramente, o desempenho da implementação será inaceitável. A preocupação é válida. Quando um membro lança uma exceção, seu desempenho pode ser ordens de magnitude mais lentas. No entanto, é possível obter um bom desempenho, aderindo estritamente às diretrizes de exceção que não permitem o uso de códigos de erro. Dois padrões descritos nesta seção sugerem maneiras de fazer isso.

✗ NÃO use códigos de erro devido a preocupações de que exceções possam afetar negativamente o desempenho.

Para melhorar o desempenho, é possível usar o padrão Testador-Executor ou o padrão Tentar-Analisar, descrito nas próximas duas seções.

Padrão Testador-Executor

Às vezes, o desempenho de um membro gerador de exceções pode ser aprimorado ao dividir o membro em dois. Vamos examinar o método `Add` da interface `ICollection<T>`.

C#

```
ICollection<int> numbers = ...  
numbers.Add(1);
```

O método `Add` gera exceção se a coleção for somente leitura. Isso pode ser um problema de desempenho em cenários nos quais se espera que a chamada de método falhe com frequência. Uma das maneiras de atenuar o problema é testar se a coleção é gravável antes de tentar adicionar um valor.

C#

```
ICollection<int> numbers = ...  
...  
if (!numbers.IsReadOnly)  
{  
    numbers.Add(1);  
}
```

O membro usado para testar uma condição, que em nosso exemplo é a propriedade `IsReadOnly`, é conhecido como o testador. O membro usado para executar uma operação potencialmente geradora de exceção, o método `Add` em nosso exemplo, é chamado de executor.

✓ CONSIDERE o padrão Testador-Executor para membros que podem gerar exceções em cenários comuns para evitar problemas de desempenho relacionados a exceções.

Padrão Tentar-Analisar

Para APIs extremamente sensíveis ao desempenho, um padrão ainda mais rápido do que o padrão Testador-Executor descrito na seção anterior deve ser usado. O padrão exige o ajuste do nome do membro para tornar um caso de teste bem definido uma parte da semântica do membro. Por exemplo, `DateTime` define um método `Parse` que gera uma exceção se a análise de uma cadeia de caracteres falhar. Ele também define um método `TryParse` que tenta analisar, mas retorna false se a análise não for bem-sucedida e retornar o resultado de uma análise bem-sucedida usando um parâmetro `out`.

C#

```
public struct DateTime  
{  
    public static DateTime Parse(string dateTime)  
    {  
        ...  
    }  
    public static bool TryParse(string dateTime, out DateTime result)  
    {  
        ...  
    }  
}
```

Ao usar esse padrão, é importante definir a funcionalidade try em termos estritos. Se o membro falhar por qualquer motivo diferente da tentativa bem definida, o membro ainda deverá gerar uma exceção correspondente.

- ✓ CONSIDERE o padrão Tentar-Analisar para membros que podem gerar exceções em cenários comuns para evitar problemas de desempenho relacionados a exceções.
- ✓ USE o prefixo "Try" e o tipo de retorno booliano para métodos que implementam esse padrão.
- ✓ FORNEÇA um membro que gera exceções para cada membro usando o padrão Tentar-Analisar.

Portions © 2005, 2009 Microsoft Corporation. Todos os direitos reservados.

Reimpresso com permissão da Pearson Education, Inc. das [Diretrizes de Design do Framework: convenções, linguagens e padrões para bibliotecas do .NET reutilizável](#), 2ª edição [↗](#) por Krzysztof Cwalina e Brad Abrams, publicado em 22 de outubro de 2008 por Addison-Wesley Professional como parte da série de desenvolvimento do Microsoft Windows.

Confira também

- [Diretrizes de design do Framework](#)
- [Diretrizes de design para exceções](#)

Diretrizes de uso

Artigo • 10/05/2023

Esta seção contém diretrizes para usar tipos comuns em APIs acessíveis publicamente. Ela explica o uso direto de tipos internos do Framework (por exemplo, atributos de serialização) e os operadores comuns de sobrecarga.

A interface [System.IDisposable](#) não é abordada nesta seção, mas é discutida em [Padrão de descarte](#).

ⓘ Observação

Para obter diretrizes e informações adicionais sobre outros tipos internos comuns do .NET Framework, confira os tópicos de referência quanto ao seguinte: [System.DateTime](#), [System.DateTimeOffset](#), [System.ICloneable](#), [System.IComparable<T>](#), [System.IEquatable<T>](#), [System.Nullable<T>](#), [System.Object](#) e [System.Uri](#).

Nesta seção

[matrizes](#)

[Atributos](#)

[Coleções](#)

[Serialização](#)

[Uso de System.XML](#)

[Operadores de igualdade\](#)

Portions © 2005, 2009 Microsoft Corporation. Todos os direitos reservados.

Reimpresso com permissão da Pearson Education, Inc. das [Diretrizes de Design do Framework: convenções, linguagens e padrões para bibliotecas do .NET reutilizável](#), 2ª edição ↗ por Krzysztof Cwalina e Brad Abrams, publicado em 22 de outubro de 2008 por Addison-Wesley Professional como parte da série de desenvolvimento do Microsoft Windows.

Confira também

- [Diretrizes de design do Framework](#)

Matrizes (diretrizes de design do .NET Framework)

Artigo • 05/03/2024

ⓘ Observação

Este conteúdo é reimpresso com permissão da Pearson Education, Inc. de *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition*. Essa edição foi publicada em 2008 e, desde então, o livro foi totalmente revisado [na terceira edição](#) [↗]. Algumas das informações nesta página podem estar desatualizadas.

✓ PREFIRA usar coleções em vez de matrizes em APIs públicas. A seção [Coleções](#) fornece detalhes sobre como escolher entre coleções e matrizes.

✗ NÃO use campos de matriz somente leitura. O campo em si é somente leitura e não pode ser alterado, mas os elementos na matriz podem ser alterados.

✓ CONSIDERE usar matrizes denteadas em vez de matrizes multidimensionais.

Uma matriz denteada é uma matriz com elementos que também são matrizes. As matrizes que constituem os elementos podem ter tamanhos diferentes, resultando em menos espaço desperdiçado para alguns conjuntos de dados (por exemplo, matriz esparsa) em comparação com matrizes multidimensionais. Além disso, o CLR otimiza as operações de índice em matrizes denteadas e, portanto, elas podem apresentar um melhor desempenho de runtime em alguns cenários.

Portions © 2005, 2009 Microsoft Corporation. Todos os direitos reservados.

Reimpresso com permissão da Pearson Education, Inc. das [Diretrizes de Design do Framework: convenções, linguagens e padrões para bibliotecas do .NET reutilizável, 2ª edição](#) [↗] por Krzysztof Cwalina e Brad Abrams, publicado em 22 de outubro de 2008 por Addison-Wesley Professional como parte da série de desenvolvimento do Microsoft Windows.

Confira também

- [Array](#)
- [Diretrizes de design do Framework](#)
- [Diretrizes de uso](#)

Atributos (diretrizes de design do .NET Framework)

Artigo • 07/10/2023

ⓘ Observação

Este conteúdo é reimpresso com permissão da Pearson Education, Inc. de *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition*. Essa edição foi publicada em 2008, e o livro desde então foi totalmente revisado na **terceira edição** [↗](#). Algumas das informações nesta página podem estar desatualizadas.

[System.Attribute](#) é uma classe base usada para definir atributos personalizados.

Os atributos são anotações que podem ser adicionadas a elementos de programação, como assemblies, tipos, membros e parâmetros. Eles são armazenados nos metadados do assembly e podem ser acessados em tempo de execução usando as APIs de reflexão. Por exemplo, Framework define o [ObsoleteAttribute](#), que pode ser aplicado a um tipo ou membro para indicar que o tipo ou membro foi preterido.

Os atributos podem ter uma ou mais propriedades que carregam dados adicionais relacionados ao atributo. Por exemplo, `ObsoleteAttribute` poderia levar informações adicionais sobre a versão na qual um tipo ou um membro foi preterido e a descrição das novas APIs substituindo a API obsoleta.

Algumas propriedades de um atributo devem ser especificadas quando o atributo é aplicado. Elas são chamadas de propriedades obrigatórias ou argumentos obrigatórios, pois são representados como parâmetros de construtor posicional. Por exemplo, a propriedade [ConditionString](#) de [ConditionalAttribute](#) é uma propriedade obrigatória.

Propriedades que não precisam necessariamente ser especificadas quando o atributo é aplicado são chamadas de propriedades opcionais (ou argumentos opcionais). Elas são representadas por propriedades configuráveis. Os compiladores fornecem sintaxe especial para definir essas propriedades quando um atributo é aplicado. Por exemplo, a propriedade [AttributeUsageAttribute.Inherited](#) representa um argumento opcional.

- ✓ ATRIBUA um nome de classes de atributo personalizadas com o sufixo "Attribute".
- ✓ APLIQUE [AttributeUsageAttribute](#) a atributos personalizados.
- ✓ FORNEÇA propriedades configuráveis para argumentos opcionais.

✓ FORNEÇA propriedades somente get para argumentos necessários.

✓ FORNEÇA parâmetros de construtor para inicializar propriedades correspondentes aos argumentos necessários. Cada parâmetro deve ter o mesmo nome (embora com maiúsculas e minúsculas diferentes) que a propriedade correspondente.

✗ EVITE fornecer parâmetros de construtor para inicializar propriedades correspondentes aos argumentos opcionais.

Em outras palavras, não tenha propriedades que possam ser definidas com um construtor e um setter. Essa diretriz torna muito explícito quais argumentos são opcionais e quais são obrigatórios e evita ter duas maneiras de fazer a mesma coisa.

✗ EVITE sobrecarregar construtores de atributo personalizados.

Ter apenas um construtor comunica claramente ao usuário quais argumentos são obrigatórios e quais são opcionais.

✓ SELE classes de atributo personalizado, se possível. Isso torna a pesquisa do atributo mais rápida.

Portions © 2005, 2009 Microsoft Corporation. Todos os direitos reservados.

Reimpresso com permissão da Pearson Education, Inc. das [Diretrizes de Design do Framework: convenções, linguagens e padrões para bibliotecas do .NET reutilizável](#), 2ª edição [↗](#) por Krzysztof Cwalina e Brad Abrams, publicado em 22 de outubro de 2008 por Addison-Wesley Professional como parte da série de desenvolvimento do Microsoft Windows.

Confira também

- [Diretrizes de design do Framework](#)
- [Diretrizes de uso](#)

Diretrizes para coleções

Artigo • 09/05/2023

Qualquer tipo projetado especificamente para manipular um grupo de objetos com uma característica comum pode ser considerado uma coleção. É quase sempre apropriado que esses tipos implementem `IEnumerable` ou `IEnumerable<T>`, portanto, nesta seção, consideramos apenas os tipos que implementam uma ou ambas as interfaces como coleções.

❌ Não use coleções com tipos fracos em APIs públicas.

O tipo de todos os parâmetros e valores retornados que representam itens de coleção devem ser o tipo de item exato, não qualquer um de seus tipos base (isso se aplica somente a membros públicos da coleção).

❌ Não use `ArrayList` ou `List<T>` em APIs públicas.

Esses tipos são estruturas de dados projetadas para serem usadas na implementação interna, não em APIs públicas. `List<T>` é otimizado para desempenho e energia à custa da limpeza das APIs e flexibilidade. Por exemplo, se você retornar `List<T>`, nunca poderá receber notificações quando o código do cliente modificar a coleção. Além disso, `List<T>` expõe muitos membros, como `BinarySearch`, que não são úteis ou aplicáveis em muitos cenários. As duas seções a seguir descrevem tipos (abstrações) projetados especificamente para uso em APIs públicas.

❌ Não use `Hashtable` ou `Dictionary<TKey, TValue>` em APIs públicas.

Esses tipos são estruturas de dados projetadas para serem usadas na implementação interna. As APIs públicas devem usar `IDictionary`, `IDictionary<TKey, TValue>` ou um tipo personalizado que implemente uma ou ambas as interfaces.

❌ NÃO use `IEnumerator<T>`, `IEnumerator` ou qualquer outro tipo que implemente qualquer uma dessas interfaces, exceto como o tipo de retorno de um método `GetEnumerator`.

Tipos que retornam enumeradores de métodos diferentes de `GetEnumerator` não podem ser usados com a instrução `foreach`.

❌ Não implemente `IEnumerator<T>` e `IEnumerable<T>` no mesmo tipo. O mesmo se aplica às interfaces não genéricas `IEnumerator` e `IEnumerable`.

Parâmetros de coleção

✓ Use o tipo menos especializado possível como um tipo de parâmetro. A maioria dos membros que usam coleções como parâmetros usa a interface `IEnumerable<T>`.

✗ Evite usar `ICollection<T>` ou `ICollection` como um parâmetro apenas para acessar a propriedade `Count`.

Em vez disso, considere usar `IEnumerable<T>` ou `IEnumerable` e verificar dinamicamente se o objeto implementa `ICollection<T>` ou `ICollection`.

Propriedades da coleção e valores retornados

✗ Não forneça propriedades de coleção configuráveis.

Os usuários podem substituir o conteúdo da coleção limpando a coleção primeiro e, em seguida, adicionando o novo conteúdo. Se a substituição de toda a coleção for um cenário comum, considere fornecer o método `AddRange` na coleção.

✓ Use `Collection<T>` ou uma subclasse de `Collection<T>` para propriedades ou valores retornados que representam coleções de leitura/gravação.

Se `Collection<T>` não atender a alguns requisitos (por exemplo, a coleção não deve implementar `IList`), use uma coleção personalizada implementando `IEnumerable<T>`, `ICollection<T>` ou `IList<T>`.

✓ Use `ReadOnlyCollection<T>`, uma subclasse de `ReadOnlyCollection<T>`, ou em casos raros `IEnumerable<T>` para propriedades ou valores retornados que representam coleções somente leitura.

Em geral, prefira `ReadOnlyCollection<T>`. Se não atender a alguns requisitos (por exemplo, a coleção não deve implementar `IList`), use uma coleção personalizada implementando `IEnumerable<T>`, `ICollection<T>` ou `IList<T>`. Se você implementar uma coleção personalizada somente leitura, implemente `ICollection<T>.IsReadOnly` para retornar `true`.

Quando você tiver certeza de que o único cenário que deseja dar suporte é a iteração somente avanço, você pode usar `IEnumerable<T>`.

✓ Considere o uso de subclasses de coleções base genéricas em vez de usar as coleções diretamente.

Isso permite um nome melhor e a adição de membros auxiliares que não estão presentes nos tipos de coleção base. Isso é especialmente aplicável a APIs de alto nível.

✓ Considere retornar uma subclasse de `Collection<T>` ou `ReadOnlyCollection<T>` de métodos e propriedades comumente usados.

Isso permitirá que você adicione métodos auxiliares ou altere a implementação da coleção no futuro.

✓ Considere o uso de uma coleção com chave se os itens armazenados na coleção tiverem chaves exclusivas (nomes, IDs etc.). As coleções de chave podem ser indexadas por um inteiro e uma chave e geralmente são implementadas herdando de `KeyedCollection<TKey, TItem>`.

Coleções com chave geralmente têm volumes de memória maiores e não devem ser usadas se a sobrecarga de memória superar os benefícios de ter as chaves.

✗ Não retorne valores nulos de propriedades de coleção ou de métodos que retornam coleções. Em vez disso, retorne uma coleção vazia ou uma matriz vazia.

A regra geral é que coleções ou matrizes nulas e vazias (0 itens) devem ser tratadas da mesma forma.

Instantâneos versus coleções dinâmicas

As coleções que representam um estado em algum momento são chamadas de coleções de instantâneos. Por exemplo, uma coleção que contém linhas retornadas de uma consulta de banco de dados seria um instantâneo. As coleções que sempre representam o estado atual são chamadas de coleções dinâmicas. Por exemplo, uma coleção de `ComboBox` itens é uma coleção dinâmica.

✗ Não retorne coleções de instantâneos de propriedades. As propriedades devem retornar coleções dinâmicas.

Os getters de propriedade devem ser operações muito leves. O retorno de um instantâneo requer a criação de uma cópia de uma coleção interna em uma operação $O(n)$.

✓ Use uma coleção de instantâneos ou um `IEnumerable<T>` dinâmico (ou seu subtipo) para representar coleções voláteis (ou seja, que podem ser alteradas sem modificar explicitamente a coleção).

Em geral, todas as coleções que representam um recurso compartilhado (por exemplo, arquivos em um diretório) são voláteis. Essas coleções são muito difíceis ou impossíveis

de serem implementadas como coleções dinâmicas, a menos que a implementação seja simplesmente um enumerador somente avanço.

Escolhendo entre matrizes e coleções

✓ Prefira coleções em vez de matrizes.

As coleções fornecem mais controle sobre o conteúdo, podem evoluir ao longo do tempo e são mais utilizáveis. Além disso, o uso de matrizes para cenários somente leitura não é recomendado porque o custo de clonagem da matriz é muito elevado. Estudos de usabilidade mostraram que alguns desenvolvedores se sentem mais confortáveis usando APIs baseadas em coleção.

No entanto, se você estiver desenvolvendo APIs de baixo nível, talvez seja melhor usar matrizes para cenários de leitura/gravação. As matrizes têm um volume de memória menor, o que ajuda a reduzir o conjunto de trabalho e o acesso a elementos em uma matriz é mais rápido porque é otimizado pelo runtime.

✓ Considere o uso de matrizes em APIs de baixo nível para minimizar o consumo de memória e maximizar o desempenho.

✓ Use matrizes de bytes em vez de coleções de bytes.

✗ Não use matrizes para propriedades se a propriedade tiver que retornar uma nova matriz (por exemplo, uma cópia de uma matriz interna) sempre que o getter de propriedade for chamado.

Implementando coleções personalizadas

✓ Considere herdar de `Collection<T>`, `ReadOnlyCollection<T>` ou `KeyedCollection<TKey, TItem>` ao projetar novas coleções.

✓ Implemente `IEnumerable<T>` ao criar novas coleções. Considere a implementação de `ICollection<T>` ou `IList<T>`, quando aplicável.

Ao implementar essa coleção personalizada, siga o padrão de API estabelecido por `Collection<T>` e `ReadOnlyCollection<T>` o mais próximo possível. Ou seja, implemente os mesmos membros explicitamente, nomeie os parâmetros como essas duas coleções e assim por diante.

✓ Considere implementar interfaces de coleção não genéricas (`IList` e `ICollection`) se a coleção geralmente for passada para APIs que tomam essas interfaces como

entrada.

✗ Evite implementar interfaces de coleção em tipos com APIs complexas não relacionadas ao conceito de uma coleção.

✗ Não herde de coleções base não genéricas, como `CollectionBase`. Em vez disso, use `Collection<T>`, `ReadOnlyCollection<T>` e `KeyedCollection<TKey, TItem>`.

Nomeação de coleções personalizadas

Coleções (tipos que implementam `IEnumerable`) são criadas principalmente por dois motivos: (1) para criar uma nova estrutura de dados com operações específicas de estrutura e, muitas vezes, características de desempenho diferentes das estruturas de dados existentes (por exemplo, `List<T>`, `LinkedList<T>`, `Stack<T>`) e (2) para criar uma coleção especializada para manter um conjunto específico de itens (por exemplo, `StringCollection`). As estruturas de dados são usadas com mais frequência na implementação interna de aplicativos e bibliotecas. As coleções especializadas devem ser expostas principalmente em APIs (como tipos de propriedade e parâmetro).

✓ Use o sufixo "Dicionário" em nomes de abstrações que implementam `IDictionary` ou `IDictionary<TKey, TValue>`.

✓ Use o sufixo "Coleção" em nomes de tipos que implementam `IEnumerable` (ou qualquer um de seus descendentes) e representa uma lista de itens.

✓ Use o nome da estrutura de dados apropriado para estruturas de dados personalizadas.

✗ Evite o uso de sufixos que impliquem uma implementação específica, como "LinkedList" ou "Hashtable", em nomes de abstrações de coleção.

✓ Considere prefixar nomes de coleção com o nome do tipo de item. Por exemplo, uma coleção que armazena itens do tipo `Address` (implementando `IEnumerable<Address>`) deve ser nomeada `AddressCollection`. Se o tipo de item for uma interface, o prefixo "I" do tipo de item poderá ser omitido. Portanto, uma coleção de `IDisposable` itens pode ser chamada `DisposableCollection`.

✓ Considere o uso do prefixo "ReadOnly" em nomes de coleções somente leitura, se uma coleção gravável correspondente puder ser adicionada ou já existir na estrutura.

Por exemplo, uma coleção somente leitura de cadeias de caracteres deve ser chamada `ReadOnlyStringCollection`.

Portions © 2005, 2009 Microsoft Corporation. Todos os direitos reservados.

Reimpresso com permissão da Pearson Education, Inc. das [Diretrizes de Design do Framework: convenções, linguagens e padrões para bibliotecas do .NET reutilizável, 2ª edição](#) [↗](#) por Krzysztof Cwalina e Brad Abrams, publicado em 22 de outubro de 2008 por Addison-Wesley Professional como parte da série de desenvolvimento do Microsoft Windows.

Confira também

- [Diretrizes de design do Framework](#)
- [Diretrizes de uso](#)

Serialização

Artigo • 09/05/2023

A serialização é o processo de conversão de um objeto em um formato que possa ser prontamente persistido ou transportado. Por exemplo, você pode serializar um objeto, transportá-lo pela Internet usando HTTP e desserializá-lo no computador de destino.

O .NET Framework oferece três tecnologias de serialização principais otimizadas para vários cenários de serialização. A tabela a seguir lista essas tecnologias e os principais tipos de estrutura relacionados a elas.

Nome da Tecnologia	Tipos principais	Cenários
Serialização do contrato de dados	DataContractAttribute DataMemberAttribute DataContractSerializer NetDataContractSerializer DataContractJsonSerializer ISerializable	Persistência geral Serviços Web JSON
Serialização XML	XmlSerializer	Formato XML com controle total sobre a forma do XML
Serialização de runtime (binário e SOAP)	SerializableAttribute ISerializable BinaryFormatter SoapFormatter	Comunicação remota .NET

✓ PENSE sobre a serialização ao criar tipos.

Escolhendo a tecnologia de serialização correta que receberá suporte

✓ CONSIDERE dar suporte à Serialização de Contrato de Dados se houver possibilidade de que as instâncias do seu tipo precisem ser persistentes ou usadas nos Serviços Web.

✓ CONSIDERE dar suporte à serialização XML em vez ou além da Serialização do Contrato de Dados se você precisar ter mais controle sobre o formato XML gerado quando o tipo for serializado.

Isso talvez seja necessário em alguns cenários de interoperabilidade no qual você precise usar um constructo XML para o qual não haja suporte na Serialização do

Contrato de Dados, por exemplo, para gerar atributos XML.

✓ CONSIDERE dar suporte à Serialização em Runtime se as instâncias do tipo precisarem ultrapassar os limites da comunicação remota .NET.

✗ EVITE dar suporte à Serialização em Runtime ou à serialização XML apenas por motivos gerais de persistência. Prefira a Serialização do Contrato de Dados em vez disso.

Oferecendo suporte à serialização do contrato de dados

Os tipos podem dar suporte à Serialização do Contrato de Dados aplicando o [DataContractAttribute](#) ao tipo e o [DataMemberAttribute](#) aos membros (campos e propriedades) do tipo.

✓ CONSIDERE marcar os membros de dados do tipo público se o tipo puder ser usado na confiança parcial.

Na confiança total, os serializadores do Contrato de Dados podem serializar e desserializar tipos e membros não públicos, mas somente os membros públicos podem ser serializados e desserializados na confiança parcial.

✓ IMPLEMENTE um getter e um setter em todas as propriedades que têm [DataMemberAttribute](#). Os serializadores do Contrato de Dados exigem o getter e o setter para o tipo sejam considerados serializáveis. (No .NET Framework 3.5 SP1, algumas propriedades de coleção podem ser somente get.) Se o tipo não for usado em confiança parcial, um ou ambos os acessadores de propriedade poderão ser não públicos.

✓ CONSIDERE usar retornos de chamada de serialização para a inicialização de instâncias desserializadas.

Os construtores não são chamados quando os objetos são desserializados. (Há exceções à regra. Construtores de coleções marcadas com [CollectionDataContractAttribute](#) são chamados durante a desserialização.) Portanto, qualquer lógica executada durante a construção normal precisa ser implementada como um dos retornos de chamada de serialização.

`OnDeserializedAttribute` é o atributo de retorno de chamada mais usado. Os outros atributos da família são [OnDeserializingAttribute](#), [OnSerializingAttribute](#) e [OnSerializedAttribute](#). Eles podem ser usados para marcar os retornos de chamada que

são executados antes de desserialização, antes da serialização e, por fim, após a serialização, respectivamente.

✓ CONSIDERE usar o [KnownTypeAttribute](#) para indicar os tipos concretos que devem ser usados ao desserializar um grafo de objetos complexos.

Considere a compatibilidade com versões anteriores e posteriores ao criar ou modificar tipos serializáveis.

Tenha em mente que os fluxos serializados de versões futuras do tipo podem ser desserializados na versão atual de tipo e vice-versa.

Verifique se compreendeu que os membros de dados, até mesmo os privados e internos, não podem alterar seus nomes, tipos ou mesmo sua ordem nas versões futuras do tipo, a menos que se tome um cuidado especial para preservar o contrato usando parâmetros explícitos para os atributos do contrato de dados.

Teste a compatibilidade da serialização ao fazer alterações a tipos serializáveis. Tente desserializar a nova versão em uma versão antiga e vice-versa.

✓ CONSIDERE implementar [IExtensibleDataObject](#) para permitir o ciclo completo entre diferentes versões do tipo.

A interface permite que o serializador assegure de que nenhum dado sejam perdido durante o ciclo completo. A propriedade [IExtensibleDataObject.ExtensionData](#) é usada para armazenar dados da versão futura do tipo que é desconhecido para a versão atual e, portanto, não pode armazená-los em seus membros de dados. Quando a versão atual for serializada e desserializada subsequentemente em uma versão futura, os dados adicionais estarão disponíveis no fluxo serializado.

Suporte à serialização XML

A Serialização do Contrato de Dados é a tecnologia de serialização principal (padrão) do .NET Framework, mas há situações de serialização para as quais a Serialização do Contrato de Dados não oferece suporte. Por exemplo, ela não fornece controle total sobre o formato XML gerado ou consumido pelo serializador. Se esse controle fino for necessário, a Serialização XML precisará ser usada e você precisará criar seus tipos para oferecer suporte essa tecnologia de serialização.

✗ EVITE criar os tipos especificamente para a Serialização XML, a menos que você tenha um motivo muito forte para controlar a forma do XML gerado. Essa tecnologia de serialização foi substituída pela serialização do contrato de dados abordada na seção anterior.

✓ CONSIDERE implementar a interface [IXmlSerializable](#) se quiser ainda mais controle sobre a forma do XML serializado do que o que é oferecido aplicando os atributos de Serialização XML. Dois métodos da interface, [ReadXml](#) e [WriteXml](#), permitem que você controle totalmente o fluxo XML serializável. Você também pode controlar o esquema XML gerado para o tipo aplicando `XmlSchemaProviderAttribute`.

Suporte à serialização em runtime

A serialização em Runtime é uma tecnologia usada pelo .NET Remoting. Se você considerar que os tipos serão transportados por meio da comunicação remota .NET, será necessário verificar se eles oferecem suporte à Serialização em Runtime.

O suporte básico da Serialização em Runtime pode ser fornecido aplicando [SerializableAttribute](#), e os cenários mais avançados envolvem a implementação de um Padrão Serializável em Runtime simples (implemente [ISerializable](#) e forneça um construtor de serialização).

✓ CONSIDERE o suporte à Serialização em Runtime se os tipos forem usados com a comunicação remota .NET. Por exemplo, o namespace [System.AddIn](#) usa a comunicação remota .NET e, portanto, todos os tipos trocados entre os suplementos `System.AddIn` precisam oferecer suporte à Serialização em Runtime.

✓ CONSIDERE implementar o Padrão Serializável em Runtime se quiser controle completo sobre o processo de serialização. Por exemplo, se você quiser transformar os dados à medida que eles forem serializados ou desserializados.

O padrão é muito simples. Tudo o que você precisa fazer é implementar a interface [ISerializable](#) e fornecer um construtor especial que é usado quando o objeto é desserializado.

✓ PROTEJA o construtor de serialização e forneça dois parâmetros tipados e nomeados exatamente conforme mostrado no exemplo aqui.

C#

```
[Serializable]
public class Person : ISerializable
{
    protected Person(SerializationInfo info, StreamingContext context)
    {
        // ...
    }
}
```

✓ IMPLEMENTE os membros [ISerializable](#) explicitamente.

✓ APLIQUE uma demanda de link para a implementação de [ISerializable.GetObjectData](#). Isso garantirá que o somente núcleo totalmente confiável e o Serializador em Runtime tenham acesso ao membro.

Portions © 2005, 2009 Microsoft Corporation. Todos os direitos reservados.

Reimpresso com permissão da Pearson Education, Inc. das [Diretrizes de Design do Framework: convenções, linguagens e padrões para bibliotecas do .NET reutilizável](#), 2ª edição [↗](#) por Krzysztof Cwalina e Brad Abrams, publicado em 22 de outubro de 2008 por Addison-Wesley Professional como parte da série de desenvolvimento do Microsoft Windows.

Confira também

- [Diretrizes de design do Framework](#)
- [Diretrizes de uso](#)

Uso de System.XML

Artigo • 12/03/2024

ⓘ Observação

Este conteúdo é reimpresso com permissão da Pearson Education, Inc. de *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition*. Essa edição foi publicada em 2008 e, desde então, o livro foi totalmente revisado [na terceira edição](#). Algumas das informações nesta página podem estar desatualizadas.

Esta seção trata do uso de vários tipos que residem em namespaces [System.Xml](#) e que podem ser usados para representar dados XML.

✗ NÃO use [XmlNode](#) ou [XmlDocument](#) para representar dados XML. Como alternativa, dê prioridade ao uso de instâncias de [IXPathNavigable](#), [XmlReader](#) ou [XmlWriter](#) ou de subtipos de [XmlNode](#). `XmlNode` e `XmlDocument` não foram projetados para a exposição em APIs públicas.

✓ NÃO use `XmlReader`, `IXPathNavigable` ou subtipos de `XmlNode` como entrada ou saída de membros que aceitam ou retornam XML.

Use essas abstrações em vez de `XmlDocument`, `XmlNode` ou [XPathDocument](#), pois isso separa os métodos de implementações específicas de um documento XML na memória e permite que eles trabalhem com fontes de dados XML virtuais que expõem `XmlNode`, `XmlReader` ou [XPathNavigator](#).

✗ NÃO crie uma subclasse de `XmlDocument` se você deseja criar um tipo que represente uma exibição XML de um modelo de objeto ou de uma fonte de dados subjacente.

Portions © 2005, 2009 Microsoft Corporation. Todos os direitos reservados.

Reimpresso com permissão da Pearson Education, Inc. das [Diretrizes de Design do Framework: convenções, linguagens e padrões para bibliotecas do .NET reutilizável, 2ª edição](#) por Krzysztof Cwalina e Brad Abrams, publicado em 22 de outubro de 2008 por Addison-Wesley Professional como parte da série de desenvolvimento do Microsoft Windows.

Confira também

- Diretrizes de design do Framework
- Diretrizes de uso

Operadores de igualdade

Artigo • 05/03/2024

ⓘ Observação

Este conteúdo é reimpresso com permissão da Pearson Education, Inc. de *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries, 2nd Edition*. Essa edição foi publicada em 2008 e, desde então, o livro foi totalmente revisado [na terceira edição](#) [↗]. Algumas das informações nesta página podem estar desatualizadas.

Esta seção discute a sobrecarga de operadores de igualdade e refere-se a `operator==` e `operator!=` como operadores de igualdade.

✗ NÃO sobrecarregue um dos operadores de igualdade e não o outro.

✓ FAÇA a verificação se `Object.Equals` e os operadores de igualdade têm exatamente a mesma semântica e características de desempenho semelhantes.

Isso geralmente significa que `Object.Equals` precisa ser substituído quando os operadores de igualdade estiverem sobrecarregados.

✗ EVITE gerar exceções de operadores de igualdade.

Por exemplo, retorne false se um dos argumentos for nulo em vez de gerar `NullReferenceException`.

Operadores de igualdade em tipos de valor

✓ FAÇA a sobrecarga dos operadores de igualdade em tipos de valor, se a igualdade for significativa.

Na maioria das linguagens de programação, não há nenhuma implementação padrão de `operator==` para tipos de valor.

Operadores de igualdade em tipos de referência

✗ EVITE sobrecarregar operadores de igualdade em tipos de referência mutáveis.

Muitas linguagens têm operadores de igualdade internos para tipos de referência. Os operadores internos geralmente implementam a igualdade de referência e muitos desenvolvedores ficam surpresos quando o comportamento padrão é alterado para a igualdade de valor.

Esse problema é mitigado para tipos de referência imutáveis porque a imutabilidade torna muito mais difícil observar a diferença entre igualdade de referência e igualdade de valor.

✗ EVITE sobrecarregar operadores de igualdade em tipos de referência se a implementação for significativamente mais lenta do que a da igualdade de referência.

Portions © 2005, 2009 Microsoft Corporation. Todos os direitos reservados.

Reimpresso com permissão da Pearson Education, Inc. das [Diretrizes de Design do Framework: convenções, linguagens e padrões para bibliotecas do .NET reutilizável](#), 2ª edição [↗](#) por Krzysztof Cwalina e Brad Abrams, publicado em 22 de outubro de 2008 por Addison-Wesley Professional como parte da série de desenvolvimento do Microsoft Windows.

Confira também

- [Diretrizes de design do Framework](#)
- [Diretrizes de uso](#)

Padrões comuns de Design

Artigo • 09/05/2023

Há vários livros sobre padrões de software, linguagens padrão e antipadrões que abordam o assunto muito amplo dos padrões. Assim, este capítulo fornece diretrizes e discussões relacionadas a um conjunto muito limitado de padrões que são usados com frequência no design das APIs do .NET Framework.

Nesta seção

[Propriedades de dependência](#)

[Padrão de descarte](#)

Portions © 2005, 2009 Microsoft Corporation. Todos os direitos reservados.

Reimpresso com permissão da Pearson Education, Inc. das [Diretrizes de Design do Framework: convenções, linguagens e padrões para bibliotecas do .NET reutilizável](#), 2ª edição [↗](#) por Krzysztof Cwalina e Brad Abrams, publicado em 22 de outubro de 2008 por Addison-Wesley Professional como parte da série de desenvolvimento do Microsoft Windows.

Confira também

- [Diretrizes de design do Framework](#)

Propriedades de Dependência

Artigo • 07/10/2023

❗ Observação

Esse conteúdo foi reimpresso por permissão da Pearson Education, Inc. das *Diretrizes de Design da Estrutura: Convenções, Idiomas e Padrões para Bibliotecas .NET Reutilizáveis, 2ª Edição*. Essa edição foi publicada em 2008, e desde então o livro foi totalmente revisado na **terceira edição** [↗](#). Algumas das informações nesta página podem estar desatualizadas.

Uma DP (propriedade de dependência) é uma propriedade regular que armazena seu valor em um repositório de propriedades em vez de armazená-lo em uma variável de tipo (campo), por exemplo.

Uma propriedade de dependência anexada é um tipo de propriedade de dependência modelada como métodos Get and Set estáticos que representam "propriedades" que descrevem relações entre objetos e seus contêineres (por exemplo, a posição de um objeto `Button` em um contêiner `Panel`).

✓ FORNEÇA as propriedades de dependência, se você precisar das propriedades para dar suporte a recursos do WPF, como estilo, gatilhos, vinculação de dados, animações, recursos dinâmicos e herança.

Design da propriedade de dependência

✓ HERDE de [DependencyObject](#), ou um de seus subtipos, ao implementar propriedades de dependência. O tipo fornece uma implementação muito eficiente de um repositório de propriedades e dá suporte automaticamente à vinculação de dados do WPF.

✓ FORNEÇA uma propriedade CLR regular e um campo somente leitura estático público que armazena uma instância de [System.Windows.DependencyProperty](#) para cada propriedade de dependência.

✓ IMPLEMENTE propriedades de dependência chamando métodos de instância [DependencyObject.GetValue](#) e [DependencyObject.SetValue](#).

✓ NOMEIE o campo estático da propriedade de dependência sufixo do nome da propriedade com "Propriedade".

✗ NÃO defina valores padrão de propriedades de dependência explicitamente no código; em vez disso, defina-os em metadados.

Se você definir explicitamente um padrão de propriedade, poderá impedir que essa propriedade seja definida por alguns meios implícitos, como um estilo.

✗ NÃO coloque código nos acessadores de propriedade diferentes do código padrão para acessar o campo estático.

Esse código não será executado se a propriedade for definida por meios implícitos, como um estilo, porque o estilo usa o campo estático diretamente.

✗ NÃO use propriedades de dependência para armazenar dados seguros. Até mesmo propriedades de dependência privada podem ser acessadas publicamente.

Design de propriedade de dependência anexada

As propriedades de dependência descritas na seção anterior representam propriedades intrínsecas do tipo de declaração; por exemplo, a propriedade `Text` é uma propriedade de `TextButton`, que a declara. Um tipo especial de propriedade de dependência é a propriedade de dependência anexada.

Um exemplo clássico de uma propriedade anexada é a propriedade `Grid.Column`. A propriedade representa a posição da coluna de Botão (não de Grade), mas só será relevante se o Botão estiver contido em uma grade e, portanto, estiver "anexado" a Botões por Grades.

XAML

```
<Grid>
  <Grid.ColumnDefinitions>
    <ColumnDefinition />
    <ColumnDefinition />
  </Grid.ColumnDefinitions>

  <Button Grid.Column="0">Click</Button>
  <Button Grid.Column="1">Clack</Button>
</Grid>
```

A definição de uma propriedade anexada se parece principalmente com a de uma propriedade de dependência regular, exceto que os acessadores são representados pelos métodos `Get` and `Set` estáticos:

C#

```
public class Grid {  
  
    public static int GetColumn(DependencyObject obj) {  
        return (int)obj.GetValue(ColumnProperty);  
    }  
  
    public static void SetColumn(DependencyObject obj, int value) {  
        obj.SetValue(ColumnProperty, value);  
    }  
  
    public static readonly DependencyProperty ColumnProperty =  
        DependencyProperty.RegisterAttached(  
            "Column",  
            typeof(int),  
            typeof(Grid)  
        );  
}
```

Validação da propriedade de dependência

As propriedades geralmente implementam o que é chamado de validação. A lógica de validação é executada quando é feita uma tentativa para alterar o valor de uma propriedade.

Infelizmente, os acessadores de propriedade de dependência não podem conter código de validação arbitrário. Em vez disso, a lógica de validação de propriedade de dependência precisa ser especificada durante o registro da propriedade.

✗ NÃO coloque a lógica de validação de propriedade de dependência nos acessadores da propriedade. Em vez disso, passe um retorno de chamada de validação para o método `DependencyProperty.Register`.

Notificações de alteração de propriedade de dependência

✗ NÃO implemente a lógica de notificação de alteração nos acessadores de propriedade de dependência. As propriedades de dependência têm um recurso interno de notificações de alteração que deve ser usado fornecendo um retorno de chamada de notificação de alteração para o [PropertyMetadata](#).

Coerção do valor da propriedade de dependência

A coerção da propriedade ocorre quando o valor fornecido a um setter de propriedade é modificado pelo setter antes que o repositório de propriedades seja realmente modificado.

✗ NÃO implemente a lógica de coerção em acessadores de propriedade de dependência.

As propriedades de dependência têm um recurso de coerção interno e podem ser usadas fornecendo um retorno de chamada de coerção para o `PropertyMetadata`.

Portions © 2005, 2009 Microsoft Corporation. Todos os direitos reservados.

Reimpresso com permissão da Pearson Education, Inc. das [Diretrizes de Design do Framework: convenções, linguagens e padrões para bibliotecas do .NET reutilizável](#), 2ª edição [↗](#) por Krzysztof Cwalina e Brad Abrams, publicado em 22 de outubro de 2008 por Addison-Wesley Professional como parte da série de desenvolvimento do Microsoft Windows.

Confira também

- [Diretrizes de design do Framework](#)
- [Padrões comuns de Design](#)

Padrão de descarte

Artigo • 02/06/2023

Todos os programas adquirem um ou mais recursos do sistema, como memória, identificadores do sistema ou conexões de banco de dados, durante a execução. Os desenvolvedores precisam ter cuidado ao usar esses recursos do sistema, que devem ser liberados depois de serem adquiridos e usados.

O CLR dá suporte ao gerenciamento automático de memória. A memória gerenciada (memória alocada usando o operador C# `new`) não precisa ser liberada explicitamente. Ela é liberada automaticamente pelo GC (coletor de lixo). Isso libera os desenvolvedores da tarefa tediosa e difícil de liberar memória e tem sido uma das principais razões da produtividade sem precedentes proporcionada pelo .NET Framework.

Infelizmente, a memória gerenciada é apenas um dos muitos tipos de recursos do sistema. Outros recursos ainda precisam ser liberados explicitamente e são chamados de recursos não gerenciados. O GC não foi projetado especificamente para gerenciar esses recursos não gerenciados, o que significa que a responsabilidade de gerenciá-los está nas mãos dos desenvolvedores.

O CLR fornece alguma ajuda na liberação de recursos não gerenciados. [System.Object](#) declara um método virtual [Finalize](#) (também chamado de finalizador), que é chamado pelo GC antes que a memória do objeto seja recuperada por ele e pode ser substituído para liberar recursos não gerenciados. Os tipos que substituem o finalizador são chamados de tipos finalizáveis.

Embora os finalizadores sejam eficazes em alguns cenários de limpeza, eles têm duas desvantagens significativas:

- O finalizador é chamado quando o GC detecta que um objeto se qualifica para coleta. Isso acontece em um período indeterminado após o recurso deixar de ser necessário. O atraso entre quando o desenvolvedor poderia ou gostaria de liberar o recurso e o momento em que o recurso realmente é liberado pelo finalizador pode ser inaceitável em programas que adquirem muitos recursos escassos (recursos que podem ser esgotados facilmente) ou em casos em que é caro manter os recursos em uso (por exemplo, grandes buffers de memória não gerenciada).
- Quando o CLR precisa chamar um finalizador, ele precisa adiar a coleta da memória do objeto até a próxima rodada de coleta de lixo (os finalizadores são executados entre coletas). Isso significa que a memória do objeto (e todos os objetos aos quais ele se refere) não será liberada por um período maior.

Portanto, depender exclusivamente de finalizadores pode não ser apropriado em muitos cenários em que é importante recuperar recursos não gerenciados o mais rápido possível, ao lidar com recursos escassos ou em cenários de alto desempenho em que a sobrecarga adicional de finalização do GC é inaceitável.

O Framework a interface [System.IDisposable](#), que deve ser implementada para fornecer ao desenvolvedor uma maneira manual de liberar recursos não gerenciados assim que eles não forem necessários. Ele também fornece o método [GC.SuppressFinalize](#), que pode informar ao GC que um objeto foi descartado manualmente e não precisa mais ser finalizado; nesse caso, a memória do objeto pode ser recuperada antes. Tipos que implementam a interface `IDisposable` são chamados de tipos descartáveis.

O Padrão de Descarte tem a finalidade de padronizar o uso e a implementação de finalizadores e da interface `IDisposable`.

O objetivo principal do padrão é reduzir a complexidade da implementação dos métodos [Finalize](#) e [Dispose](#). A complexidade decorre do fato de que os métodos compartilham alguns, mas não todos os caminhos de código (as diferenças são descritas posteriormente no capítulo). Além disso, há motivos históricos para alguns elementos do padrão, relacionados à evolução do suporte da linguagem ao gerenciamento de recursos determinístico.

✓ **IMPLEMENTE** o Padrão de Descarte Básico em tipos que contêm instâncias de tipos descartáveis. Consulte a seção [Padrão de Descarte Básico](#) para obter detalhes sobre o padrão básico.

Se um tipo for responsável pelo tempo de vida de outros objetos descartáveis, os desenvolvedores também precisarão de uma maneira de descartá-los. Usar o método `Dispose` do contêiner é uma maneira conveniente de tornar isso possível.

✓ **IMPLEMENTE** o Padrão de Descarte Básico e forneça um finalizador em tipos que contêm recursos que precisam ser liberados explicitamente e que não têm finalizadores.

Por exemplo, o padrão deve ser implementado em tipos que armazenam buffers de memória não gerenciados. A seção [Tipos Finalizáveis](#) aborda diretrizes relacionadas à implementação de finalizadores.

✓ **CONSIDERE** implementar o Padrão de Descarte Básico em classes que, por si só, não contêm recursos não gerenciados nem objetos descartáveis, mas provavelmente têm subtipos que contêm.

Um ótimo exemplo disso é a classe [System.IO.Stream](#). Embora essa seja uma classe base abstrata que não contém recursos, a maioria de suas subclasses contêm e, por isso, ela implementa esse padrão.

Padrão de Descarte Básico

A implementação básica do padrão envolve implementar a interface

`System.IDisposable` e declarar o método `Dispose(bool)`, que implementa toda a lógica de limpeza de recursos a ser compartilhada entre o método `Dispose` e o finalizador opcional.

O seguinte exemplo mostra uma implementação simples do padrão básico:

C#

```
public class DisposableResourceHolder : IDisposable {  
  
    private SafeHandle resource; // handle to a resource  
  
    public DisposableResourceHolder() {  
        this.resource = ... // allocates the resource  
    }  
  
    public void Dispose() {  
        Dispose(true);  
        GC.SuppressFinalize(this);  
    }  
  
    protected virtual void Dispose(bool disposing) {  
        if (disposing) {  
            if (resource != null) resource.Dispose();  
        }  
    }  
}
```

O parâmetro booleano `disposing` indica se o método foi invocado da implementação `IDisposable.Dispose` ou do finalizador. A implementação `Dispose(bool)` deve verificar o parâmetro antes de acessar outros objetos de referência (por exemplo, o campo de recurso no exemplo anterior). Esses objetos só devem ser acessados quando o método é chamado da implementação `IDisposable.Dispose` (quando o parâmetro `disposing` é igual a `true`). Se o método for invocado do finalizador (se `disposing` for `false`), outros objetos não deverão ser acessados. Isso ocorre porque os objetos são finalizados em uma ordem imprevisível e, portanto, eles ou qualquer uma de suas dependências podem já ter sido finalizados.

Além disso, esta seção se aplica a classes com uma base que ainda não implementa o Padrão de Descarte. Se você estiver herdando de uma classe que já implementa o padrão, basta substituir o método `Dispose(bool)` para fornecer lógica de limpeza de recursos adicional.

✓ **DECLARE** um método `protected virtual void Dispose(bool disposing)` para centralizar toda a lógica relacionada à liberação de recursos não gerenciados.

Toda a limpeza de recursos deve ocorrer nesse método. O método é chamado do finalizador e do método `IDisposable.Dispose`. O parâmetro será `false` se for invocado de dentro de um finalizador. Ele deve ser usado para garantir que qualquer código em execução durante a finalização não esteja acessando outros objetos finalizáveis. Detalhes sobre a implementação de finalizadores são descritos na próxima seção.

C#

```
protected virtual void Dispose(bool disposing) {  
    if (disposing) {  
        if (resource != null) resource.Dispose();  
    }  
}
```

✓ **IMPLEMENTE** a interface `IDisposable` chamando `Dispose(true)` seguido por `GC.SuppressFinalize(this)`.

A chamada para `SuppressFinalize` só deverá ocorrer se `Dispose(true)` for executada com êxito.

C#

```
public void Dispose(){  
    Dispose(true);  
    GC.SuppressFinalize(this);  
}
```

X **NÃO** torne o método sem parâmetros `Dispose` virtual.

O método `Dispose(bool)` é o que deve ser substituído por subclasses.

C#

```
// bad design  
public class DisposableResourceHolder : IDisposable {  
    public virtual void Dispose() { ... }  
    protected virtual void Dispose(bool disposing) { ... }  
}  
  
// good design  
public class DisposableResourceHolder : IDisposable {  
    public void Dispose() { ... }  
    protected virtual void Dispose(bool disposing) { ... }  
}
```

X NÃO declare sobrecargas do método `Dispose` diferentes de `Dispose()` e `Dispose(bool)`.

`Dispose` deve ser considerada uma palavra reservada para ajudar a codificar esse padrão e evitar confusão entre implementadores, usuários e compiladores. Algumas linguagens podem optar por implementar automaticamente esse padrão em determinados tipos.

✓ **PERMITA** que o método `Dispose(bool)` seja chamado mais de uma vez. O método pode optar por não fazer nada após a primeira chamada.

C#

```
public class DisposableResourceHolder : IDisposable {  
  
    bool disposed = false;  
  
    protected virtual void Dispose(bool disposing) {  
        if (disposed) return;  
        // cleanup  
        ...  
        disposed = true;  
    }  
}
```

X EVITE gerar uma exceção de dentro de `Dispose(bool)`, exceto em situações críticas em que o processo de contenção foi corrompido (vazamentos, estado compartilhado inconsistente etc.).

Os usuários esperam que uma chamada para `Dispose` não gere exceções.

Se `Dispose` puder gerar uma exceção, a lógica adicional de limpeza do bloco `finally` não será executada. Para contornar isso, o usuário precisaria encapsular cada chamada para `Dispose` (dentro do bloco `finally`!) em um bloco `try`, o que leva a manipuladores de limpeza muito complexos. Se estiver executando um método `Dispose(bool disposing)`, nunca gere uma exceção se descartar for `false`. Isso encerrará o processo se ele estiver em execução em um contexto de finalizador.

✓ **GERE** um `ObjectDisposedException` de qualquer membro que não possa ser usado após o objeto ter sido descartado.

C#

```
public class DisposableResourceHolder : IDisposable {  
    bool disposed = false;
```

```

SafeHandle resource; // handle to a resource

public void DoSomething() {
    if (disposed) throw new ObjectDisposedException(...);
    // now call some native methods using the resource
    ...
}
protected virtual void Dispose(bool disposing) {
    if (disposed) return;
    // cleanup
    ...
    disposed = true;
}
}

```

✓ **CONSIDERE** fornecer o método `Close()`, além do `Dispose()`, se fechar for a terminologia padrão na área.

Ao fazer isso, é importante tornar a implementação `Close` idêntica a `Dispose` e considerar implementar o método `IDisposable.Dispose` explicitamente.

C#

```

public class Stream : IDisposable {
    IDisposable.Dispose() {
        Close();
    }
    public void Close() {
        Dispose(true);
        GC.SuppressFinalize(this);
    }
}

```

Tipos finalizáveis

Tipos finalizáveis são tipos que estendem o Padrão de Descarte Básico substituindo o finalizador e fornecendo o caminho do código de finalização no método `Dispose(bool)`.

Os finalizadores são notoriamente difíceis de implementar da maneira correta, principalmente porque você não pode fazer determinadas suposições (normalmente válidas) sobre o estado do sistema durante a execução deles. As diretrizes a seguir devem ser consideradas atentamente.

Observe que algumas das diretrizes se aplicam não apenas ao método `Finalize`, mas a qualquer código chamado de um finalizador. No caso do Padrão de Descarte Básico

definido anteriormente, isso significa a lógica que é executada dentro de `Dispose(bool disposing)` quando o parâmetro `disposing` é `false`.

Se a classe base já for finalizável e implementar o Padrão de Descarte Básico, você não deverá substituir `Finalize` novamente. Em vez disso, substitua o método `Dispose(bool)` para fornecer lógica adicional de limpeza de recursos.

O seguinte código mostra um exemplo de tipo finalizável:

C#

```
public class ComplexResourceHolder : IDisposable {

    private IntPtr buffer; // unmanaged memory buffer
    private SafeHandle resource; // disposable handle to a resource

    public ComplexResourceHolder() {
        this.buffer = ... // allocates memory
        this.resource = ... // allocates the resource
    }

    protected virtual void Dispose(bool disposing) {
        ReleaseBuffer(buffer); // release unmanaged memory
        if (disposing) { // release other disposable objects
            if (resource != null) resource.Dispose();
        }
    }

    ~ComplexResourceHolder() {
        Dispose(false);
    }

    public void Dispose() {
        Dispose(true);
        GC.SuppressFinalize(this);
    }
}
```

X EVITE tornar os tipos finalizáveis.

Considere atentamente qualquer caso em que você acreditar que um finalizador é necessário. Há um custo real associado a instâncias com finalizadores, tanto do ponto de vista do desempenho quanto da complexidade do código. Prefira usar wrappers de recursos, como `SafeHandle`, para encapsular recursos não gerenciados sempre que possível; nesse caso, um finalizador se torna desnecessário porque o wrapper é responsável pela própria limpeza de recursos.

X NÃO torne tipos de valor finalizáveis.

Somente tipos de referência realmente são finalizáveis pelo CLR e, portanto, qualquer tentativa de colocar um finalizador em um tipo de valor será ignorada. Os compiladores de C# e C++ impõem essa regra.

✓ **TORNE** um tipo finalizável se ele for responsável por liberar um recurso não gerenciado que não tenha o próprio finalizador.

Ao implementar o finalizador, basta chamar `Dispose(false)` e colocar toda a lógica de limpeza de recursos dentro do método `Dispose(bool disposing)`.

C#

```
public class ComplexResourceHolder : IDisposable {  
  
    ~ComplexResourceHolder() {  
        Dispose(false);  
    }  
  
    protected virtual void Dispose(bool disposing) {  
        ...  
    }  
}
```

✓ **IMPLEMENTE** o Padrão de Descarte Básico em cada tipo finalizável.

Isso proporciona aos usuários do tipo um meio de executar explicitamente a limpeza determinística desses mesmos recursos pelos quais o finalizador é responsável.

X NÃO acesse nenhum objeto finalizável no caminho do código do finalizador, pois há um risco significativo de que eles já tenham sido finalizados.

Por exemplo, um objeto finalizável A que tem uma referência a outro objeto finalizável B não pode usar B de modo confiável no finalizador A, ou vice-versa. Os finalizadores são chamados em ordem aleatória (além de uma garantia de ordenação fraca para finalização crítica).

Além disso, observe que objetos armazenados em variáveis estáticas serão coletados em determinados pontos durante o descarregamento de um domínio do aplicativo ou ao sair do processo. Acessar uma variável estática que se refere a um objeto finalizável (ou chamar um método estático que pode usar valores armazenados em variáveis estáticas) poderá não ser seguro se [Environment.HasShutdownStarted](#) retornar true.

✓ **TORNE** o método `Finalize` protegido.

Desenvolvedores de C#, C++ e VB.NET não precisam se preocupar com isso, pois os compiladores ajudam a impor essa diretriz.

X NÃO permita que exceções escapem da lógica do finalizador, exceto para falhas críticas do sistema.

Se uma exceção for gerada de um finalizador, o CLR desligará todo o processo (desde o .NET Framework versão 2.0), impedindo que outros finalizadores sejam executados e que os recursos sejam liberados de maneira controlada.

✓ **CONSIDERE** criar e usar um objeto finalizável crítico (um tipo com uma hierarquia de tipos que contém [CriticalFinalizerObject](#)) para situações em que um finalizador absolutamente precisa ser executado, mesmo em face de descarregamentos forçados do domínio do aplicativo e anulações de thread.

Portions © 2005, 2009 Microsoft Corporation. Todos os direitos reservados.

Reimpresso com permissão da Pearson Education, Inc. das [Diretrizes de Design do Framework: convenções, linguagens e padrões para bibliotecas do .NET reutilizável](#), 2ª edição [↗](#) por Krzysztof Cwalina e Brad Abrams, publicado em 22 de outubro de 2008 por Addison-Wesley Professional como parte da série de desenvolvimento do Microsoft Windows.

Confira também

- [IDisposable.Dispose](#)
- [Object.Finalize](#)
- [Diretrizes de design do Framework](#)
- [Padrões comuns de Design](#)
- [Coleta de lixo](#)