Interfaces em C# — Guia de Autoestudo

Uma linha de domínio unificada guia os exemplos: Relatórios, Exportação e Entrega (Report → Transform → Compress → Transport/Store). Este documento, técnico e didático, foca interfaces, composição e DI para desenvolvedores C# intermediários/avançados. Você encontrará heurísticas de design, padrões modernos de C#, exercícios e um mini-projeto final para consolidar aprendizado.

Por que Interfaces: contratos, alternância e coesão

Interfaces definem capacidades plugáveis: descrevem o que um componente faz sem impor como. Isso viabiliza alternância de implementações (CSV vs JSON, ZIP vs GZIP, HTTP vs SMTP) e composição por eixos. Ao acoplar o cliente ao contrato, preservamos **LSP** e ampliamos testabilidade.

Em arquiteturas modernas, alternar detalhes é requisito desde o início: trocar armazenamento local por S3/MinIO, simular SMTP por HTTP, isolar I/O em testes. Interfaces permitem **DI/IoC**, **feature flags** e **design orientado a políticas** sem ramificações condicionais espalhadas.

DI/IoC (Inversão de Controle)

Ideia: quem **usa** uma dependência não a **cria**; recebe-a pronta via **Injeção de Dependência** (DI).

> **Por quê?** Reduz acoplamento, facilita testes (mocks/stubs) e permite **alternar implementações** sem tocar no cliente.
>
> **Como no .NET:** você registra contratos no contêiner (`IServiceCollection`) e o runtime injeta as implementações no construtor.

Feature flags (alternar comportamento em runtime)

Ideia: "chaves" de configuração que ligam/desligam comportamentos, **sem recompilar**.

> **Uso saudável:** a flag **não** fica espalhada em `if` por todo o código; ela **define uma política** central que escolhe estratégias registradas no DI.
>
> **Benefício:** experimentos (A/B), *rollouts* e *fallbacks* previsíveis.

Design orientado a políticas, sem ramificações condicionais

A chave é **concentrar a decisão** em um **único orquestrador/factory**, e **compor** peças por **chaves** (políticas) — em vez de `if`/`else` espalhados.

Por fim, interfaces reforçam **ISP**: contratos pequenos, coesos e fáceis de simular. Evitamos canivetes suíços e downcasts, favorecendo orquestração explícita por composição. O resultado é menor acoplamento, maior clareza e testes rápidos.

O que é ISP?

O **Interface Segregation Principle** (Princípio da Segregação de Interfaces), o "I" do SOLID, diz: **clientes não devem ser forçados a depender de métodos que não usam**. Na prática, preferimos **interfaces pequenas e coesas**, cada uma representando uma capacidade específica, em vez de um contrato "canivete suíço" com muitos membros opcionais.

Sinais de Violação

Interface "gorda", métodos lançando `NotImplementedException`, clientes usando só 10–30% do contrato, necessidade de `as`/downcasts.

Benefícios da Aplicação

Menor acoplamento, **testes mais simples** (mocks focados), e **código mais claro**.

Aquecimento conceitual: interfaces sem código

Para entender o poder das interfaces em C#, faremos uma ponte do mundo real para o conceito de contratos de capacidade. Imagine interfaces não como código, mas como acordos e funções que descrevem o "o quê" sem impor o "como". Neste aquecimento, focaremos no raciocínio conceitual, sem sintaxe de C#, classes ou Injeção de Dependência.

Três cenas do cotidiano

Entrega de Relatório

Você precisa entregar um relatório. Pode exportá-lo como PDF ou CSV, compactá-lo (ou não) como ZIP, e enviá-lo por e-mail ou salvá-lo em um drive. A beleza é que você troca as "peças" (formato, compressão, método de entrega) sem alterar o objetivo principal: entregar o relatório.

Pagamento em Loja

Em uma loja, o objetivo é pagar. Aceitam cartão de crédito, Pix, boleto bancário. A "maquininha" ou o método específico de pagamento muda, mas a ação de pagar — cumprir a transação financeira — permanece a mesma. O cliente só precisa saber que pode pagar, não como o sistema processa cada método internamente.

Tradução de Texto

Você tem um texto em português e precisa dele em inglês. Esse texto pode ser traduzido por um tradutor humano ou por um serviço online de IA. O "contrato" aqui é claro: "traduzir o texto da língua A para a língua B". O mecanismo por trás da tradução é um detalhe de implementação que pode ser alternado.

Padrão de raciocínio: Contratos de Capacidade

Estas cenas revelam um padrão fundamental por trás do design de software orientado a interfaces:

- Existe um **objetivo estável** e bem definido (entregar, pagar, traduzir).
- Há **peças alternáveis** que atendem ao mesmo **contrato** (oferecem a mesma capacidade).
- O usuário ou sistema cliente não precisa saber **como** cada peça executa sua tarefa, apenas **o que** ela promete entregar.
- Quando o problema envolve escolher e combinar essas peças de forma flexível, interfaces são a ferramenta perfeita para modelar esses contratos de capacidade.

Este padrão de raciocínio é a essência do porquê interfaces são tão valiosas: elas permitem construir sistemas flexíveis onde detalhes de implementação podem ser trocados sem afetar a lógica central.

Glossário enxuto

Contrato

Uma promessa de comportamento ou capacidade (métodos/assinaturas) — o "o quê" será feito.

Implementação

O modo concreto e específico de cumprir o contrato — o "como" será feito.

Política

Um conjunto de regras que guiam a escolha e combinação de diferentes implementações (ex.: "à noite, usar CSV com ZIP via e-mail").

Orquestração

O processo de aplicar a política, selecionar as implementações corretas e conectá-las para atingir o objetivo.

Segregação (ISP)

O princípio de criar contratos pequenos e coesos, cada um representando uma única capacidade, para evitar interfaces "inchadas".

> Interface é útil quando o que varia é **a peça/implementação** e o que fica fixo é **o objetivo/contrato**.

Mapa mental antes do código

Para facilitar a compreensão do valor das interfaces em C#, propomos uma abordagem didática que inicia com conceitos triviais, avança para a programação orientada a objetos (OO) e só então introduz as interfaces. Nosso objetivo é reduzir a carga cognitiva inicial, construindo o conhecimento de forma progressiva.

Heurística leve (antes do código)

Adote esta heurística para guiar seu design antes de escrever a primeira linha de código, observando como a complexidade evolui e onde as interfaces se tornam indispensáveis:

Problema simples

Comece sem OO (abordagem funcional/procedural). Observe onde surgem `if`s por modo e código duplicado. Isso indica uma necessidade de segregação (dividir a interface em contratos menores).

Variação de comportamento

Quando diferentes comportamentos precisam coexistir, extraia classes (OO) para separar algoritmos e estratégias. Isso promove o polimorfismo e a responsabilidade única.

Alternar implementações

Ao precisar alternar implementações e testar isoladamente, crie interfaces. Use injeção de dependências para gerenciar as trocas e facilitar a testabilidade.

Guard-rails didáticos

Estas diretrizes o ajudarão a manter um design limpo e flexível:

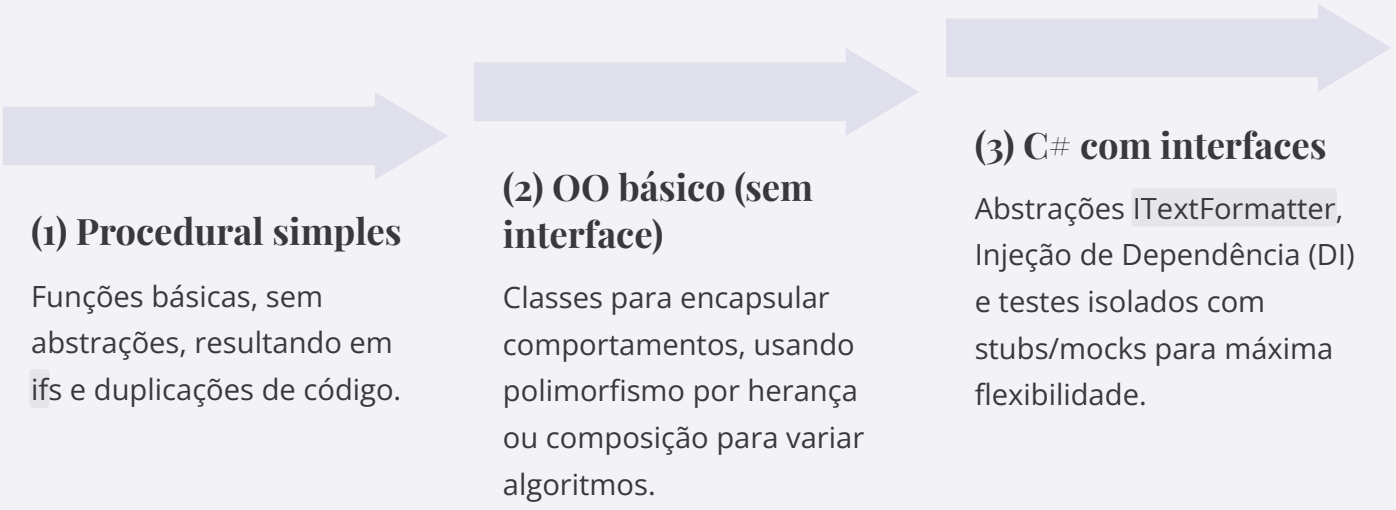
Evite `switch`/`if` por "modo" espalhados pelo código; isso é um forte indicador de acoplamento excessivo e falta de polimorfismo.

Uma classe deve ter apenas uma razão para mudar (Princípio da Responsabilidade Única - SRP). Classes "folha" (sem herdeiros) podem ser `sealed` para otimização.

O cliente deve programar para a interface, não para a classe concreta. Isso garante flexibilidade e permite a substituição de implementações sem impactar o código cliente.

Mapa visual dos passos

Visualize a progressão do design, da simplicidade à flexibilidade:



Sinais de alerta

Fique atento a estes "cheiros de código" que indicam que você pode estar violando os princípios das interfaces:

→

Interface "gorda" com muitos métodos, ou implementações que lançam `NotImplementedException` para funcionalidades não usadas.

→

Uso frequente de `downcasts` (`as`/`is`) ou ramificações condicionais baseadas no tipo da instância, indicando que o cliente não programa para a interface.

→

Testes difíceis ou lentos devido a dependências concretas que não podem ser facilmente substituídas por dublês (stubs/mocks).

Trilha Introdutória: Da Função ao Contrato

Para ilustrar a evolução do pensamento de design que leva ao uso de interfaces, vamos acompanhar um problema simples – formatar um texto – através de três passos incrementais. Observaremos como cada abordagem resolve os desafios da anterior e introduz novos benefícios, culminando na flexibilidade e testabilidade que as interfaces proporcionam.

Passo 1: Abordagem Procedural (Sem OO)

Começamos com a forma mais direta de resolver o problema: uma função procedural que utiliza um condicional (um `switch`, neste caso) para determinar o comportamento com base em um "modo" de operação. Embora simples para poucos casos, essa abordagem rapidamente se torna rígida e difícil de manter à medida que novos modos são adicionados.

```
string Format(string text, string mode)
{
   return mode switch
   {
      "upper" => text.ToUpperInvariant(),
      "lower" => text.ToLowerInvariant(),
      _ => text
   };
}
```

> Insight: Um `switch` (ou múltiplos `if/else if`) baseado em "modo" ou "tipo" tende a crescer e se repetir em diferentes partes do código. Cada nova regra de formatação exigiria modificações na função central.

O `switch expression` é uma forma enxuta de tomar decisões em C# (desde o C# 8) que **produz um valor** em vez de executar blocos de comandos como o `switch` tradicional. Você escreve uma única expressão que "mapeia" a entrada para um resultado por meio de **braços** (`=>`), normalmente finalizando com o curinga `_` para o caso padrão. Ele favorece **código declarativo**, reduz `if/else` encadeado, evita *fall-through*, funciona muito bem com **pattern matching** (tipos, `null`, enums, condições `when`, padrões relacionais), e exige **exaustividade**: se faltar um caso obrigatório, o compilador acusa. Por ser uma expressão, é ideal para formatações, cálculos, seleção de estratégias e conversões simples — mantendo **clareza, imutabilidade** (sem efeitos colaterais) e fácil testabilidade.

Passo 2: Orientação a Objetos (Sem Interfaces)

Para mitigar a rigidez da abordagem procedural, aplicamos os princípios da Orientação a Objetos. Criamos uma classe base abstrata e implementações concretas para cada variação de formatação. Isso encapsula o comportamento, permitindo que cada "modo" (agora uma classe) seja uma unidade coesa e responsável por seu próprio comportamento. O cliente consome essas classes através do polimorfismo, sem a necessidade de condicionais explícitos.

```
abstract class TextFormatter {
  public abstract string Apply(string text);
}

sealed class UpperCaseFormatter : TextFormatter {
  public override string Apply(string t) => t.ToUpperInvariant();
}

sealed class LowerCaseFormatter : TextFormatter {
  public override string Apply(string t) => t.ToLowerInvariant();
}
```

> Insight: Cada variação de comportamento (como formatar para maiúsculas ou minúsculas) se torna uma classe coesa. O cliente agora interage com objetos, não mais com `string modes`, e o `switch` é substituído pela seleção da instância apropriada.

Passo 3: C# com Interfaces (Plugável e Testável)

O passo final introduz as interfaces, elevando a abstração para um "contrato". Em vez de depender de uma classe base concreta ou abstrata, definimos um contrato explícito (`ITextFormatter`) que qualquer classe pode implementar. Isso remove o acoplamento direto a uma hierarquia de classes, permitindo maior flexibilidade, testabilidade (com dublês) e a capacidade de "plugar" diferentes implementações via Injeção de Dependência (DI) em tempo de execução, como demonstrado com o registro simples.

```
public interface ITextFormatter {
  string Apply(string text);
}

public sealed class UpperCaseFormatter : ITextFormatter {
  public string Apply(string t) => t.ToUpperInvariant();
}

public sealed class LowerCaseFormatter : ITextFormatter {
  public string Apply(string t) => t.ToLowerInvariant();
}

public static class Renderer
{
   public static string Render(ITextFormatter fmt, string text)
     => fmt.Apply(text);
}

// Uso (ex.: Program.cs)
var fmt = new UpperCaseFormatter();        // escolhe a peça
var output = Renderer.Render(fmt, "Hello"); // injeta por parâmetro
Console.WriteLine(output);
```

> Insight: As interfaces nos permitem programar para um contrato, não para uma implementação específica. Isso torna o código **plugável** (fácil de trocar implementações), **testável** (fácil de criar dublês) e altamente **flexível** para futuras extensões sem alterar o código cliente.

Construtor recebe a interface

```
public sealed class ReportPrinter
{
    private readonly ITextFormatter _fmt;
    public ReportPrinter(ITextFormatter fmt) => _fmt = fmt;

    public string Print(string text) => _fmt.Apply(text);
}

// Uso
var printer = new ReportPrinter(new UpperCaseFormatter());
Console.WriteLine(printer.Print("Hello"));
```

“Catálogo” simples por chave (sem `if` em todo lugar)

```
var formatters = new Dictionary<string, ITextFormatter>(StringComparer.OrdinalIgnoreCase)
{
    ["upper"] = new UpperCaseFormatter(),
    ["lower"] = new LowerCaseFormatter()
};

string Render(string mode, string text) => formatters[mode].Apply(text);

// Uso
Console.WriteLine(Render("upper", "Hello"));
```

Procedural: Funções

Lógica direta, `switch` por "modo", acoplamento.

OO: Classes

Comportamento encapsulado, polimorfismo, menos `switch`.

Interfaces: Contratos

Abstração total, plugável, testável, DI.

Interfaces em C#: O Contrato Essencial

As interfaces são um pilar fundamental da programação orientada a objetos em C#, promovendo abstração, flexibilidade e testabilidade. Elas definem um contrato de comportamento que as classes devem cumprir, sem especificar a implementação. Nesta seção, exploramos a sintaxe, os diferentes tipos de implementação e o poder das interfaces genéricas com constraints.

Sintaxe, Herança Múltipla e Nomenclatura

Em C#, a convenção de nomenclatura para interfaces é prefixá-las com a letra **I**, como em `IReportExporter` ou `ICompressor`. Isso facilita a identificação de interfaces no código.

Diferentemente da herança de classes, C# permite a herança múltipla de interfaces. Uma interface pode herdar de uma ou mais interfaces, combinando seus contratos em um único. Além disso, uma classe pode implementar um número ilimitado de interfaces, adotando todos os contratos definidos por elas. Isso permite que uma única classe possa se comportar de múltiplas maneiras, dependendo da interface pela qual é acessada.

```
public interface ITransport : IAsyncDisposable, IHealthCheck { /* ... */ }

public interface IReportExporter
{
    byte[] Export(object reportModel);
}

public interface ITransform
{
    object Apply(object reportModel);
}
```

Implementação Implícita vs. Explícita

Ao implementar uma interface em C#, você pode escolher entre implementação implícita ou explícita.

- **Implementação Implícita:** É a forma mais comum. Os membros da interface se tornam parte do contrato público da classe que a implementa. Isso é ideal quando o nome do membro da interface é claro e não há conflitos.

```
public class CsvExporter : IReportExporter
{
    public byte[] Export(object reportModel) { /* ... */ }
}
```

- **Implementação Explícita:** Usada quando um membro da interface não deve aparecer no contrato público direto da classe concreta, ou para resolver colisões de nomes entre múltiplas interfaces com métodos de mesma assinatura. O membro só é acessível quando a instância da classe é explicitamente convertida para o tipo da interface.

Use a implementação explícita para:

- Esconder métodos "internos" que não fazem parte da responsabilidade primária da classe.

- Evitar poluir a API pública da classe com métodos que são relevantes apenas para o contrato da interface.
- Desambiguar implementações quando duas interfaces que a classe implementa definem um método com o mesmo nome e assinatura.

```
public interface IInternalMetrics { void Collect(); }

public class CsvExporter : IReportExporter, IInternalMetrics
{
    byte[] IReportExporter.Export(object reportModel) { /* ... */ }
    void IInternalMetrics.Collect() { /* ... */ }
}
```

Interfaces Genéricas e Constraints

Interfaces genéricas permitem definir contratos com tipos placeholder (`T`, `TIn`, `TOut`), tornando-as reutilizáveis para uma variedade de tipos, enquanto mantêm a segurança de tipo. Constraints (restrições) podem ser aplicadas a esses parâmetros de tipo para garantir que apenas tipos que atendem a certos critérios (como implementar uma interface específica ou ter um construtor sem parâmetros) possam ser usados.

Isso é crucial para garantir que as operações dentro da interface genérica sejam válidas para os tipos fornecidos.

```
public interface ITransformer<TIn, TOut>
{
    TOut Apply(TIn input);
}

public interface ISorter<T> where T : IComparable<T>
{
    IEnumerable<T> Sort(IEnumerable<T> items);
}
```

Boas práticas incluem manter os nomes dos parâmetros de tipo curtos e significativos, como `T` para um tipo geral, `TIn` para entrada e `TOut` para saída. Sempre adote constraints quando as operações da interface dependem de certas capacidades do tipo (como a capacidade de ser comparado, demonstrado por `IComparable<T>` no exemplo do `ISorter`).

Membros de Interface Padrão (C# 8+)

Introduzidos no C# 8.0, os Membros de Interface Padrão (DIMs - Default Interface Members) permitem que interfaces forneçam uma implementação padrão para seus membros. Isso significa que as classes que implementam a interface não são obrigadas a fornecer uma implementação para esses membros específicos, a menos que desejem sobrescrever o comportamento padrão. Esta funcionalidade é particularmente útil para adicionar novos membros a interfaces existentes sem quebrar o código de implementações antigas.

```
public interface ICompressor
{
    byte[] Compress(byte[] data);
    // fallback trivial — aceitável quando não fere SRP
    public virtual string Extension => ".bin";
}
```

No exemplo acima, a interface `ICompressor` define um método `Compress` que deve ser implementado por qualquer classe que a adote. No entanto, ela também define uma propriedade `Extension` com uma implementação padrão que retorna `".bin"`. Classes que implementam `ICompressor` podem usar esse valor padrão ou fornecer sua própria implementação para `Extension`.

Quando Usar DIMs

Utilize DIMs com parcimônia para fornecer implementações triviais e seguras que não ferem o Princípio da Responsabilidade Única (SRP). São ideais para adicionar funcionalidade a interfaces existentes sem quebrar a compatibilidade binária ou para oferecer uma implementação conveniente que a maioria das classes implementadoras pode aceitar sem modificações.

Quando Evitar DIMs

Evite colocar lógica complexa ou que varie significativamente por política em contratos de interface. Nesses casos, prefira **extension methods** para utilidades ou exija que as classes implementadoras forneçam sua própria lógica. Também seja cauteloso quando houver chance de conflitos binários em cenários de versionamento de bibliotecas.

Princípio da Segregação de Interfaces (ISP)

O Princípio da Segregação de Interfaces (ISP) é o "I" no acrônimo SOLID e afirma que clientes não devem ser forçados a depender de interfaces que não usam. Em outras palavras, é melhor ter muitas interfaces específicas do que uma interface "gorda" e monolítica.

Preferir Contratos Pequenos e Coesos

Em vez de uma interface "canivete suíço" como um `INotifier` onipotente que lida com todos os tipos de notificação, é mais eficaz criar contratos menores e mais focados. Por exemplo, interfaces como `IEmailSender`, `ISmsSender` e `IPushSender` permitem que cada cliente dependa apenas do que realmente precisa.

Sinais de Interfaces "Gordas"

Interfaces que violam o ISP geralmente exibem certos padrões: muitos membros opcionais, métodos com implementações padrão vazias ou que lançam exceções, e clientes que utilizam apenas um pequeno subconjunto dos membros da interface. Estes são indicadores claros de que a interface precisa ser segregada.

Padrões .NET Comuns de Interfaces

O ecossistema .NET utiliza interfaces de forma extensiva para definir padrões de comportamento e capacidades. Compreender esses padrões é fundamental para escrever código robusto e interoperável.

- **IDisposable / IAsyncDisposable:** Essenciais para o gerenciamento de recursos não gerenciados. Permitem a liberação determinística de recursos (como arquivos, conexões de rede ou memória) por meio do bloco `using` ou `await using`.
- **IAsyncEnumerable<T>:** Introduzido no C# 8, permite a implementação de streams de dados assíncronos que podem ser consumidos incrementadamente, semelhante ao `IEnumerable<T>`, mas com suporte a operações assíncronas (`await foreach`).
- **IComparable<T> / IEqualityComparer<T>:** Utilizadas para definir como objetos são ordenados ou comparados para igualdade. `IComparable<T>` permite a ordenação natural de um tipo, enquanto `IEqualityComparer<T>` oferece mecanismos de comparação de igualdade personalizados, muitas vezes usados em coleções.

Um exemplo prático de interface que incorpora bons princípios e padrões .NET é a `IStorage`:

```
public interface IStorage : IAsyncDisposable
{
   Task WriteAsync(string path, Stream content,
      CancellationToken ct = default);
   Task<Stream> ReadAsync(string path, CancellationToken ct = default);
}
```

Esta interface define um contrato para operações de armazenamento, incorporando `IAsyncDisposable` para o correto gerenciamento de recursos assíncronos, além de métodos para leitura e escrita de streams de forma assíncrona com suporte a cancelamento.

Repository Progressivo: Persistência por Interfaces

Este exercício prático demonstra como as interfaces, em conjunto com o Princípio da Inversão de Dependência (DIP), permitem a construção de sistemas flexíveis e testáveis. Começaremos com uma implementação simples em memória e progrediremos para persistência em arquivos, sem alterar a lógica de negócio que utiliza o repositório.

> ### Etapa 1: Coleções em Memória (sem I/O)
>
> Começamos definindo um contrato genérico para um repositório e uma implementação em memória. O `InMemoryRepository` armazena os itens em uma lista simples, ideal para testes unitários rápidos do domínio sem depender de operações de I/O reais.

```
public interface IRepository<T>
{
    void Add(T item);
    IEnumerable<T> All();
    IEnumerable<T> Find(Func<T,bool> pred);
}

public sealed class InMemoryRepository<T> : IRepository<T>
{
    private readonly List<T> _items = new();
    public void Add(T item) => _items.Add(item);
    public IEnumerable<T> All() => _items;
    public IEnumerable<T> Find(Func<T,bool> pred) => _items.Where(pred);
}
```

> ### Etapa 2: CSV (I/O Simples e Síncrono)
>
> Em seguida, introduzimos uma implementação que persiste os dados em um arquivo CSV. Esta etapa ilustra a transição para I/O real, ainda que síncrona, e mostra como o repositório pode ser configurado com funções de serialização/desserialização específicas para CSV.

```
public sealed class CsvRepository<T> : IRepository<T>
{
    private readonly string _path;
    private readonly Func<T,string> _toCsv;
    private readonly Func<string,T> _fromCsv;
```

```
    public CsvRepository(string path, Func<T,string> toCsv,
           Func<string,T> fromCsv){
      _path=path;
      _toCsv=toCsv;
      _fromCsv=fromCsv;
    }

    public void Add(T item) =>
      File.AppendAllLines(_path, new[]{ _toCsv(item) });
    public IEnumerable<T> All() =>
      File.Exists(_path) ? File.ReadAllLines(_path).Select(_fromCsv) :
        Enumerable.Empty<T>();
    public IEnumerable<T> Find(Func<T,bool> pred) => All().Where(pred);
}
```

> ### Etapa 3: JSON (Estrutura Mais Rica)
>
> A etapa final utiliza JSON para um formato de persistência mais robusto, capaz de lidar com estruturas de dados complexas. A interface `IRepository<T>` permanece inalterada, destacando a capacidade de trocar a implementação de persistência sem afetar o código cliente.

```
public sealed class JsonRepository<T> : IRepository<T>
{
    private readonly string _path;
    public JsonRepository(string path) => _path = path;

    public void Add(T item)
    {
        var list = All().ToList(); list.Add(item);
        File.WriteAllText(_path,
          System.Text.Json.JsonSerializer.Serialize(list));
    }

    public IEnumerable<T> All()
    {
        if (!File.Exists(_path))
          return Enumerable.Empty<T>();
        var json = File.ReadAllText(_path);
        return System.Text.Json.JsonSerializer
          .Deserialize<List<T>>(json) ?? new List<T>();
    }
```

```
    public IEnumerable<T> Find(Func<T,bool> pred) => All().Where(pred);
}
```

Os pontos cruciais deste exemplo são a manutenção de um contrato único (`IRepository<T>`) que abstrai os detalhes da persistência. Isso permite a alternância fácil de implementações por meio de Injeção de Dependência (DI) e a capacidade de realizar testes de domínio rápidos e isolados usando o `InMemoryRepository<T>`, sem a necessidade de acessar o sistema de arquivos ou outros recursos externos.

Testabilidade: Facilite a Verificação

A testabilidade é um pilar fundamental no desenvolvimento de software robusto e de fácil manutenção. Ao projetar componentes com interfaces, garantimos que eles possam ser testados de forma isolada, rápida e confiável. Este guia explora princípios práticos e um exemplo concreto para demonstrar como as interfaces facilitam a criação de código testável, mesmo antes de introduzir ferramentas complexas ou fluxos assíncronos.

Princípios Práticos para Testabilidade Eficaz

Dependa de Interfaces no Ponto de Uso

Sempre que um componente precisa interagir com outro, o ideal é que ele dependa do contrato (interface) e não da implementação concreta. Isso permite que diferentes implementações sejam "plugadas" no futuro, incluindo versões para teste.

Dublês Mínimos para Teste

Utilize stubs ou fakes para substituir dependências em testes. Stubs fornecem retornos fixos para simular cenários específicos, enquanto fakes oferecem implementações leves em memória que mimetizam o comportamento real sem complexidade externa.

Evite I/O Real nos Testes Unitários

Testes unitários devem ser rápidos e determinísticos. Isso significa que eles não devem tocar no sistema de arquivos, banco de dados ou rede. Use interfaces e dublês para isolar a lógica de negócio dessas operações de I/O.

Composição Centralizada de Dependências

A criação e orquestração das dependências devem acontecer em um ponto centralizado — seja no próprio teste, seja no ponto de entrada da aplicação. Isso evita que a responsabilidade de "saber criar" as dependências se espalhe pelo código.

Exemplo Prático: Testando um Formatador de Texto

Considere o desafio de formatar texto de diferentes maneiras (maiúsculas, minúsculas, etc.). Em vez de usar um `switch` ou uma lógica condicional complexa, podemos empregar interfaces para criar formatadores intercambiáveis e, o mais importante, facilmente testáveis.

Contrato da Interface

Primeiro, definimos uma interface simples que representa a capacidade de formatar uma string.

```
public interface ITextFormatter { string Apply(string text); }
```

Stub de Teste

Para testar um cliente que utiliza `ITextFormatter`, podemos criar uma implementação de teste (um stub) que simula o comportamento desejado sem qualquer complexidade real.

```
file sealed class StubFormatter : ITextFormatter
{
    public string Apply(string text) => $"[{text}]";
}
```

Teste do Cliente com xUnit

O teste a seguir demonstra como um componente cliente (`Renderer`) pode ser verificado usando o `StubFormatter`. O cliente não precisa saber como o formatador funciona internamente, apenas que ele implementa o contrato `ITextFormatter`.

```
[Fact]
public void Render_UsaContrato_SemConhecerImplementacao()
{
    var fmt = new StubFormatter();
    var result = Renderer.Render(fmt, "Hello");
    Assert.Equal("[Hello]", result);
}
```

> **Insight:** O cliente depende exclusivamente do contrato da interface. Isso significa que podemos trocar o `StubFormatter` por um `UpperCaseFormatter`, `LowerCaseFormatter` ou qualquer outra implementação sem modificar o código do cliente, garantindo flexibilidade e facilitando testes específicos para cada cenário.

Testando o Repository Progressivo: Flexibilidade e Testabilidade

A jornada de refatoração de um repositório, passando de uma implementação em memória para CSV e depois JSON, não apenas demonstra a flexibilidade das interfaces, mas também realça drasticamente a testabilidade do seu código. A capacidade de "plugar" diferentes estratégias de persistência sem alterar a lógica de negócio do cliente é um pilar da arquitetura orientada a interfaces.

Fake `InMemoryRepository`

Para testes unitários rápidos e isolados, uma implementação em memória do seu contrato de repositório é inestimável. Ela simula o comportamento de um repositório real, mas opera inteiramente na memória, eliminando a dependência de I/O de disco ou rede. Isso garante que seus testes sejam determinísticos e executem em milissegundos.

```
public sealed class InMemoryRepository<T> : IRepository<T>
{
    private readonly List<T> _items = new();
    public void Add(T item) => _items.Add(item);
    public IEnumerable<T> All() => _items;
    public IEnumerable<T> Find(Func<T,bool> pred) => _items.Where(pred);
}
```

Teste de Domínio Isolado (sem I/O real)

Com o `InMemoryRepository`, podemos testar a lógica de domínio de um serviço que utiliza o repositório sem qualquer efeito colateral externo. O exemplo abaixo mostra como um `BookService` pode ser testado para verificar se adiciona e lista livros corretamente, sem tocar no sistema de arquivos.

```
[Fact]
public void BookService_AdicionaELista_SemIoReal()
{
    var repo = new InMemoryRepository<Book>();
    var svc  = new BookService(repo); // serviço depende só do contrato

    svc.Add(new Book("DDD", "Evans"));

    Assert.Contains(svc.ListAll(), b => b.Title == "DDD");
}
```

Teste com Outras Implementações: CsvRepository

A grande vantagem de se estruturar o código em torno de interfaces, como o `IRepository<T>`, reside na facilidade de substituir a implementação subjacente sem modificar o código do cliente. Enquanto o `InMemoryRepository<T>` é ideal para testes unitários rápidos e isolados, o `CsvRepository<T>` nos permite realizar testes de integração leves. Isso verifica a persistência de dados em um formato real (CSV) sem a complexidade de um banco de dados completo.

```
[Fact]
public void BookService_FuncionaComCsvRepository()
{
    using var tmp = new TempFile(); // helper de teste p/ limpar depois
    var repo = new CsvRepository<Book>(tmp.Path, ToCsv, FromCsv);
    var svc  = new BookService(repo);

    svc.Add(new Book("Clean Code", "Martin"));

    Assert.Contains(svc.ListAll(), b => b.Title == "Clean Code");
}
```

Como o exemplo demonstra, o código do teste para o `BookService` permanece praticamente inalterado ao trocar o repositório de memória para um CSV. Isso é um testemunho da robusta separação de preocupações que as interfaces proporcionam. O `BookService`, nosso cliente, não precisa saber os detalhes de como os livros são armazenados; ele apenas "fala" com o contrato `IRepository<T>`, garantindo que a lógica de negócio esteja sempre bem coberta.

Flexibilidade Modular

Permite alternar entre diferentes mecanismos de persistência sem refatorar o código cliente.

Acoplamento Reduzido

Desacopla a lógica de negócio da infraestrutura de persistência, tornando o sistema mais resiliente.

Testabilidade Reforçada

Facilita a criação de testes de unidade e integração claros e focados, isolando dependências.

Boas Práticas e Próximos Passos na Testabilidade com Interfaces

Após explorar a flexibilidade e a testabilidade que as interfaces proporcionam, é fundamental consolidar algumas boas práticas. Estas diretrizes o ajudarão a escrever código mais limpo, manutenível e, claro, testável. Além disso, vamos vislumbrar algumas possibilidades de evolução para cenários mais avançados que você pode considerar à medida que aprofunda seus conhecimentos.

Nomes Descritivos

Use nomes que realmente contam a história e a intenção de suas implementações. Nomes como `StubFormatter` e `InMemoryRepository<T>` deixam claro o propósito e o comportamento esperado, tornando seu código mais legível e compreensível para toda a equipe.

Teste Comportamento Observável

Evite asserts frágeis que dependem de detalhes de implementação. Concentre-se em testar o comportamento observável do sistema. Por exemplo, em vez de verificar o estado interno de um repositório, verifique se um item adicionado aparece corretamente em uma listagem posterior. Isso garante que seus testes sejam robustos e menos propensos a quebrar com refatorações internas.

Um Dublê Por Capacidade

Aderindo ao Princípio de Segregação de Interfaces (ISP), se um stub ou fake começar a crescer demais ou a simular múltiplas responsabilidades, isso é um forte indício de que a interface original pode estar muito abrangente. Considere segregá-la em contratos menores e mais coesos para cada capacidade.

Evolução Opcional para Cenários Avançados

Para aqueles que desejam aprofundar ainda mais a utilização de interfaces e aprimorar a performance e a escalabilidade de seus sistemas, as seguintes evoluções opcionais podem ser consideradas:

- **Migração para Métodos Assíncronos:** Refatore métodos de I/O para versões assíncronas (`Task<T>`, `ValueTask<T>`) e utilize `CancellationToken` para gerenciamento de operações de longa duração, melhorando a responsividade da aplicação.
- **Fakes de `IStorage` com Memória:** Crie implementações `fake` de interfaces de armazenamento (ex: `IStorage`) utilizando coleções em memória como `Dictionary<string, byte[]>` combinado com `MemoryStream`. Isso é excelente para testes de unidade que simulam interação com armazenamento sem tocar em disco real.
- **Leitura Incremental com `IAsyncEnumerable<T>`:** Explore o uso de `IAsyncEnumerable<T>` em cenários onde a leitura de dados pode ser incremental ou sob demanda, otimizando o consumo de memória e a performance em grandes conjuntos de dados.

Cheiros e Antídotos: Downcast/Switch por Modo

Ao longo da jornada de refatoração, identificamos "cheiros" (code smells) que indicam áreas de melhoria no código. Um cheiro comum, especialmente em código que evolui de uma abordagem mais procedural, é a dependência excessiva de instruções `switch` ou sequências de `if/else if` para variar o comportamento de um método com base em um parâmetro de modo (string, enum, etc.). Este padrão frequentemente leva a um acoplamento indesejado e dificulta a extensibilidade.

O exemplo abaixo ilustra essa armadilha clássica e, em seguida, seu antídoto usando polimorfismo através de interfaces, alinhando-se com o Princípio Aberto/Fechado (OCP).

Antes: Abordagem Procedural e Acoplada

```
string Format(string text, string mode)
{
    return mode switch
    {
        "upper" => text.ToUpperInvariant(),
        "lower" => text.ToLowerInvariant(),
        _ => text
    };
}
```

Nesta versão, a lógica de formatação está diretamente acoplada aos literais de string (`"upper"`, `"lower"`). Qualquer nova regra de formatação exige a modificação deste método, violando o OCP. Além disso, o cliente que invoca `Format` precisa conhecer os "modos" disponíveis, criando um acoplamento explícito a detalhes de implementação.

Depois: Polimorfismo por Contrato

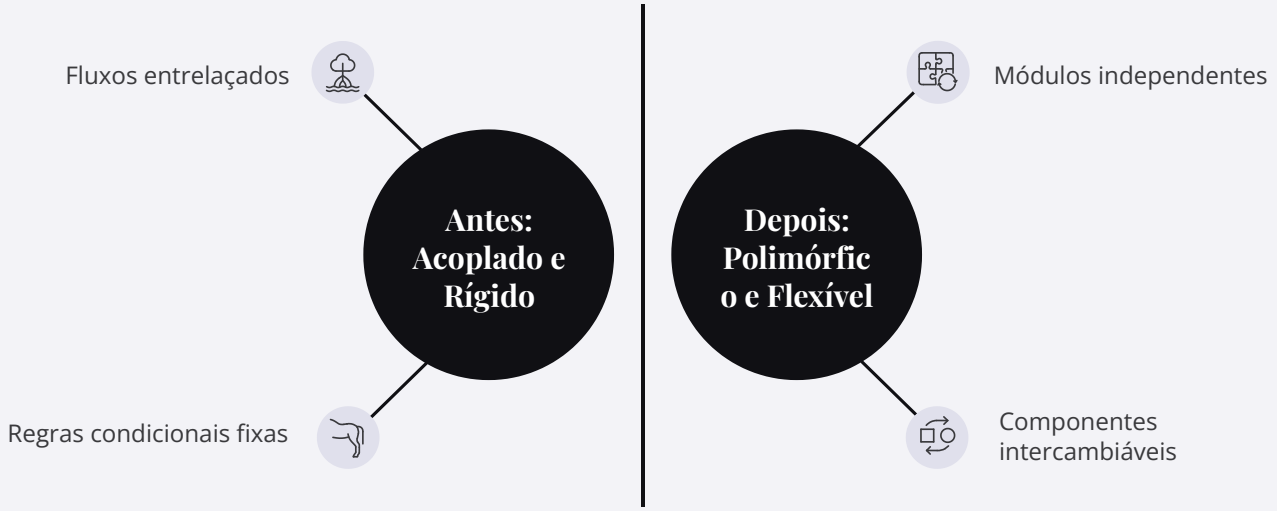
```
public interface ITextFormatter {
   string Apply(string text);
}

public sealed class UpperCaseFormatter : ITextFormatter {
   public string Apply(string t)=> t.ToUpperInvariant();
}

public sealed class LowerCaseFormatter : ITextFormatter {
   public string Apply(string t)=> t.ToLowerInvariant();
}

string Render(ITextFormatter fmt, string text) => fmt.Apply(text);
```

O antídoto reside na introdução da interface ITextFormatter. Agora, cada variação de comportamento (formatar para maiúsculas ou minúsculas) se torna uma classe coesa. O cliente interage com objetos que implementam ITextFormatter, não mais com literais de string mode. O switch é substituído pela seleção e injeção da instância apropriada, resultando em menor acoplamento, maior clareza e um sistema mais fácil de estender e testar. O cliente agora depende de um contrato (a interface), não dos detalhes de como a formatação é realizada.



Cheiros e Antídotos: Refinando Contratos de Interface

Continuando nossa jornada de identificação e correção de "cheiros" de código, vamos abordar duas armadilhas comuns no design de interfaces: as interfaces obesas e os contratos frágeis. Ambas violam princípios de design importantes e dificultam a manutenibilidade e extensibilidade do código. A boa notícia é que o Princípio de Segregação de Interfaces (ISP) e a composição explícita oferecem antídotos eficazes.

Interface Obesa

Uma **interface obesa** (ou gorda) é aquela que agrupa muitas responsabilidades e métodos não relacionados. Quando uma classe implementa essa interface, ela é forçada a implementar métodos que talvez não utilize, violando o Princípio de Segregação de Interfaces (ISP). Isso leva a implementações vazias ou a lógica desnecessária, aumentando o acoplamento e a fragilidade do sistema.

Antes: Interface Multifuncional e Acoplada

```
public interface INotifier {
    void Email(...);
    void Sms(...);
```

```
    void Push(...);
    void WhatsApp(...);
}
```

Neste exemplo, `INotifier` exige que qualquer classe que a implemente seja capaz de enviar notificações por e-mail, SMS, push e WhatsApp. Se uma classe precisa apenas enviar e-mails, ela ainda terá que fornecer implementações (talvez vazias ou que lancem exceções) para os outros métodos, o que é um claro sinal de uma interface obesa.

Depois: Segregação por Capacidade (ISP)

```
public interface IEmailSender {
    void Email(...);
}
public interface ISmsSender {
    void Sms(...);
}
public interface IPushSender {
    void Push(...);
}
```

O antídoto para interfaces obesas é aplicar o ISP: segregar a interface em contratos menores e mais coesos. Cada cliente dependerá apenas da capacidade que realmente precisa. Isso não só simplifica as implementações, mas também melhora a clareza do código e reduz o acoplamento, tornando o sistema mais flexível e testável.

Contratos Frágeis

Um **contrato frágil** surge quando uma interface inclui métodos com parâmetros opcionais excessivos ou que nem sempre fazem sentido para todas as implementações. Isso pode indicar que a interface tenta servir a múltiplos propósitos ou que uma parte da responsabilidade deveria estar em outro lugar. Tais contratos podem levar a código confuso e propenso a erros, além de dificultar a criação de implementações concisas.

Antes: Parâmetros Opcionais Inadequados

```
public interface IExporter {
    byte[] Export(...);
    void SetFooter(string? footer = null);
}
```

Aqui, a interface `IExporter` possui um método `SetFooter` com um parâmetro opcional. Mas e se nem todos os exportadores necessitarem de rodapés? Ou se a lógica de rodapé for complexa e específica, não pertencendo à responsabilidade principal de exportar? Isso gera uma dependência de um comportamento opcional que nem sempre é relevante, tornando o contrato frágil.

Depois: Separação de Responsabilidades

```
public interface IExporter {
    byte[] Export(...);
}
public interface IFooterProvider {
    string GetFooter();
}
```

A solução é separar os papéis. Em vez de acoplar a configuração do rodapé à interface de exportação, criamos uma interface `IFooterProvider`. Agora, um exportador que precisa de um rodapé pode receber um `IFooterProvider` através de injeção de dependência ou composição. Isso mantém os contratos pequenos e coesos, favorecendo a composição explícita de capacidades e eliminando a necessidade de parâmetros opcionais desnecessários na interface principal.

Refatorando Variações e Decisões Espalhadas

Continuando aprimorando o design de nossas interfaces, abordaremos agora dois "cheiros" de código que frequentemente surgem em sistemas complexos: a variação por formato dentro de repositórios e a tomada de decisão espalhada pelo código. A solução, em ambos os casos, reside em favorecer a injeção de dependências e a composição para manter os contratos coesos e as responsabilidades bem definidas.

Variação por Formato: Evitando Ramificações Internas

Quando um repositório interno gerencia múltiplas formas de persistência (CSV, JSON, etc.) através de lógicas condicionais, ele se torna acoplado a detalhes de implementação e difícil de estender. O Princípio da Responsabilidade Única (SRP) e o Aberto/Fechado (OCP) são violados.

Antes: Lógica Condicional Interna

```
public sealed class FileBookRepository : IRepository<Book>
{
    private readonly string _format; // "csv" | "json"
```

```
    public IEnumerable<Book> All() =>
      _format == "csv" ? ReadCsv() : ReadJson();
}
```

Neste cenário, a escolha do formato é feita internamente, forçando o `FileBookRepository` a conhecer os detalhes de ambos os formatos (CSV e JSON) e a ter lógica condicional para alternar entre eles. Adicionar um novo formato (e.g., XML) exigiria modificar esta classe, violando o OCP e tornando-a mais complexa.

Depois: Implementações Coesas e Seleção Externa

```
public sealed class CsvRepository<T>  : IRepository<T> { /* ... */ }
public sealed class JsonRepository<T> : IRepository<T> { /* ... */ }

// Seleção fora do cliente:
IRepository<Book> repo = useJson ? new JsonRepository<Book>(path) :
  new CsvRepository<Book>(path);
```

O antídoto é separar as responsabilidades em classes distintas (`CsvRepository` e `JsonRepository`), cada uma com uma única preocupação. A seleção da implementação correta é feita externamente, no ponto de composição (e.g., no inicializador da aplicação ou em um container IoC). Isso mantém o contrato `IRepository` único e limpo, evita "feature flags" internas e facilita a adição de novos formatos sem modificar o código existente.

Decisão Espalhada vs. Ponto Único de Composição

Similarmente, a duplicação de lógica de decisão sobre qual implementação de interface usar, espalhada por vários serviços ou módulos, introduz redundância e torna a manutenção e os testes mais difíceis.

Antes: Decisão Replicada em Vários Lugares

```
var repo = config.UseJson ? new JsonRepository<Book>(path) :
  new CsvRepository<Book>(path);
// ... este trecho repetido em múltiplos serviços ...
```

Quando a lógica de instanciar uma dependência é copiada e colada em diferentes partes da aplicação, qualquer mudança nessa lógica (por exemplo, a introdução de um novo tipo de repositório ou uma alteração na forma como a decisão é tomada) exige que todas as ocorrências sejam atualizadas, aumentando a chance de erros e tornando o código frágil.

Depois: Ponto Único de Composição (Fábrica/DI)

```
public static class RepositoryFactory
{
    public static IRepository<T> Create<T>(bool useJson, string path)
      => useJson ? new JsonRepository<T>(path) :
         new CsvRepository<T>(path);
}

// Consumo:
var repo = RepositoryFactory.Create<Book>(useJson, path);
```

Ao centralizar a lógica de criação em um ponto único, como uma fábrica simples (`RepositoryFactory`) ou utilizando um container de Injeção de Dependência (DI), garantimos que a decisão sobre qual implementação será utilizada ocorra apenas uma vez. Isso facilita a troca de implementações, simplifica a configuração em ambientes diferentes e torna o sistema muito mais testável, já que podemos injetar mocks ou outras implementações facilmente.

Para cheiros ligados a eixos plugáveis (explosão de subclasses, política fora do orquestrador, catálogo por chave), veja o Anexo A.

Composição por Eixos: Uma Abordagem Avançada

A composição por eixos é uma técnica poderosa, mas opcional, no design de software com interfaces. Ela se torna mais relevante quando o aluno já dominou o fluxo de pensamento procedural, orientado a objetos e o uso de interfaces básicas, além de ter praticado a implementação de repositórios. Seu propósito didático aqui é consolidar conceitos de orquestração, gerenciamento de políticas e a alternância de implementações sem a proliferação de declarações `if/switch` espalhadas pelo código.

Quando faz sentido usar eixos?

A modelagem por eixos é uma solução elegante para cenários de alta complexidade e variabilidade, onde o uso tradicional de herança levaria a uma "explosão de subclasses". Considere adotar essa abordagem nas seguintes situações:

Múltiplas Etapas Separadas

Quando o problema pode ser decomposto em uma sequência de etapas bem definidas e independentes, como um pipeline de processamento (ex: Exportar → Transformar → Comprimir → Transportar/Armazenar dados).

Várias Estratégias por Etapa

Se cada etapa do pipeline pode ter diversas estratégias ou implementações equivalentes (ex: formatos CSV/JSON para exportação, Normalização/Enriquecimento para transformação, compactação Zip/Gzip/Nenhum, transporte via HTTP/SMTP/S3).

Combinações Dinâmicas

Quando há a necessidade de alterar combinações dessas estratégias em tempo de execução ou por configuração (usando *feature flags*, experimentos A/B, etc.), sem precisar recompilar ou alterar o código-cliente.

Evitar Explosão de Subclasses

A heurística chave: se a tentativa de resolver a variabilidade com herança resulta em uma avalanche de classes como `CsvZipHttpExporter`, `JsonGzipS3Exporter`, etc., isso é um forte indicativo de que a composição por eixos é a solução mais apropriada.

Arquitetura Conceitual

A arquitetura por eixos plugáveis é centrada na ideia de um pipeline de componentes que podem ser trocados dinamicamente. Ela envolve a definição clara de contratos, um mecanismo de registro de implementações e uma forma de orquestrar essas implementações com base em políticas.

Contrato por Eixo

Cada etapa do pipeline é definida por uma interface (um contrato) que especifica o "o quê" fazer, ou seja, a capacidade funcional, sem ditar o "como".

Catálogo por Chave

É um registro que mapeia chaves textuais (ex: "csv", "json", "zip") para implementações concretas dos contratos de interface. Ele atua como um diretório de serviços plugáveis.

Política

Define quais chaves e, consequentemente, quais implementações devem ser utilizadas para cada cenário específico (ex: "durante o dia usar compactação Gzip, à noite usar Zip").

Orquestrador

É o componente responsável por ler a política, resolver as implementações corretas no catálogo e compor o pipeline de execução, garantindo que as dependências sejam satisfeitas e o fluxo seja executado conforme o planejado.

Contratos Mínimos e Coesos para Eixos Plugáveis

A base da abordagem de composição por eixos plugáveis reside na definição de contratos de interface que são propositadamente pequenos e coesos. Cada interface representa uma única capacidade ou etapa no pipeline, evitando a sobrecarga de responsabilidades. Isso garante que cada componente tenha uma única razão para mudar, aderindo ao Princípio da Responsabilidade Única (SRP).

A interface `IStrategy` serve como um contrato fundamental para todas as estratégias, provendo uma propriedade `Key` que permite a identificação unívoca de cada implementação no catálogo, eliminando a necessidade de testes de tipo ou instruções `switch` em tempo de execução.

```
public interface IStrategy { string Key { get; } }

public interface IReportExporter : IStrategy {
  byte[] Export(object model);
}

public interface ITransform : IStrategy {
  object Apply(object model);
}
public interface ICompressor : IStrategy {
  byte[] Compress(byte[] data);
  string Extension { get; }
}

public interface ITransport : IStrategy {
  void Send(byte[] payload, string name);
}

public interface IStorage : IStrategy {
  Task StoreAsync(string name, byte[] payload,
    CancellationToken ct = default);
}
```

Implementações Concretas

Com os contratos estabelecidos, diversas implementações concretas podem ser criadas para cada interface. Cada classe selada (`sealed class`) foca em um comportamento específico e declara explicitamente sua `Key`, tornando-a registrável e selecionável dinamicamente. Esta modularidade permite a fácil adição de novas estratégias sem impactar o código existente.

Exportadores

```
public sealed class CsvExporter : IReportExporter {
  public string Key => "csv";
  public byte[] Export(object m) {
    /*...*/
  }
}

public sealed class JsonExporter: IReportExporter {
  public string Key => "json";
  public byte[] Export(object m) {
    /*...*/
  }
}
```

Compressores

```
public sealed class ZipCompressor  : ICompressor {
  public string Key => "zip";
  public string Extension => ".zip";
  public byte[] Compress(byte[] d) {
    /*...*/
  }
}

public sealed class GzipCompressor : ICompressor {
  public string Key => "gzip";
  public string Extension => ".gz";
  public byte[] Compress(byte[] d) {
    /*...*/
  }
}
```

```
public sealed class NoopCompressor : ICompressor {
  public string Key => "none";
  public string Extension => ".bin";
  public byte[] Compress(byte[] d)=>d;
}
```

Transportadores

```
public sealed class HttpTransport : ITransport {
  public string Key => "http";
  public void Send(byte[] p, string n) {
    /* POST */
  }
}

public sealed class SmtpTransport : ITransport {
  public string Key => "smtp";
  public void Send(byte[] p, string n) {
    /* e-mail */
  }
}
```

O Catálogo de Estratégias: Fim dos Switches

O coração da flexibilidade desta abordagem é o `StrategyCatalog`. Ele é responsável por registrar todas as implementações disponíveis para um dado contrato (`IStrategy`) e fornecer a instância correta com base na `Key` fornecida pela política. Este catálogo elimina a necessidade de encadear múltiplas instruções `if` ou `switch` para determinar qual implementação usar, tornando o sistema mais limpo e extensível.

```
public interface IStrategyCatalog<T> where T: IStrategy {
  T Get(string key);
}

public sealed class StrategyCatalog<T> :
    IStrategyCatalog<T> where T: class, IStrategy
{
  private readonly IReadOnlyDictionary<string, T> _map;
  public StrategyCatalog(IEnumerable<T> strategies) =>
    _map = strategies.ToDictionary(s => s.Key,
      StringComparer.OrdinalIgnoreCase);
```

```
    public T Get(string key) => _map[key]; // lança se política inválida
}
```

Para cenários onde um contêiner de Injeção de Dependência (DI) não é utilizado, a instância do catálogo pode ser facilmente montada "manualmente" (o que chamamos de "Poor man's DI"), agrupando as implementações necessárias e passando-as ao construtor do `StrategyCatalog`.

```
// Poor man’s DI:
new StrategyCatalog<ICompressor>(
  new ICompressor[] {
    new ZipCompressor(), new GzipCompressor(), new NoopCompressor()
  }
)
```

Políticas (features/flags) e Orquestração

A flexibilidade alcançada com o catálogo de estratégias se materializa na combinação com políticas dinâmicas. Uma política (ou um conjunto de 'feature flags') define qual implementação específica de cada interface será utilizada em um determinado cenário. O orquestrador, por sua vez, é o componente que lê essa política e coordena a execução das estratégias selecionadas, sem conhecer os detalhes internos de cada uma.

A classe `DeliveryPolicy` exemplifica como a configuração pode ser externamente definida, permitindo que a aplicação altere seu comportamento sem modificações no código. Cada propriedade mapeia para a chave de uma estratégia específica no catálogo.

```
public sealed class DeliveryPolicy {
    public string Exporter {get;init;} = "csv";
    public string Transform {get;init;}="none";
    public string Compressor{get;init;}="zip";
    public string Transport{get;init;}="http";
}
```

O `ReportDeliveryOrchestrator` atua como o "maestro" desse sistema. Ele recebe todos os catálogos de estratégias e uma função para obter a política atual. Durante a execução, ele consulta a política para identificar as estratégias desejadas e as invoca em sequência, criando um fluxo de trabalho altamente adaptável e configurável.

```
public sealed class ReportDeliveryOrchestrator
{
    private readonly IStrategyCatalog<IReportExporter> _exp;
    private readonly IStrategyCatalog<ITransform> _tr;
    private readonly IStrategyCatalog<ICompressor> _zip;
```

```
    private readonly IStrategyCatalog<ITransport> _tx;
    private readonly Func<DeliveryPolicy> _policy;

    public ReportDeliveryOrchestrator(
      IStrategyCatalog<IReportExporter> exp,
      IStrategyCatalog<ITransform> tr,
      IStrategyCatalog<ICompressor> zip,
      IStrategyCatalog<ITransport> tx,
      Func<DeliveryPolicy> policy)
    {
      _exp=exp;
      _tr=tr;
      _zip=zip;
      _tx=tx;
      _policy=policy;
    }

    public void Deliver(object model, string name)
    {
       var p = _policy();
       var exporter   = _exp.Get(p.Exporter);
       var transform  = _tr.Get(p.Transform);
       var compressor = _zip.Get(p.Compressor);
       var transport  = _tx.Get(p.Transport);

       var raw   = exporter.Export(model);
       var shaped= transform.Key=="none" ? model : transform.Apply(model);
       var data  = compressor.Compress(raw);
       transport.Send(data, name + compressor.Extension);
    }
}
```

Definição da Política

Uma `DeliveryPolicy` é obtida, determinando o conjunto de estratégias a serem usadas.

Seleção das Estratégias

O orquestrador busca as implementações corretas nos catálogos com base nas chaves da política.

Execução Coordenada

As estratégias de exportação, transformação, compressão e transporte são invocadas sequencialmente.

Este padrão permite uma separação de preocupações clara: a política define "o que" fazer, o catálogo oferece "quais" opções estão disponíveis e o orquestrador gerencia "como" essas opções são combinadas e executadas. Isso resulta em um sistema modular, fácil de testar, manter e evoluir.

Validações Essenciais e Testabilidade no Modelo de Interfaces

Para garantir a robustez, a segurança e a manutenibilidade de um sistema construído sobre interfaces e orquestração de estratégias, é crucial estabelecer um conjunto claro de validações de coesão e priorizar a testabilidade em todas as etapas de desenvolvimento. Estas práticas asseguram que os componentes interagem de forma harmoniosa, que os dados são processados corretamente e que o sistema se comporta conforme o esperado, mesmo diante de cenários complexos.

Compatibilidade de Formato

Garanta que a saída de um componente é compatível com a entrada do próximo. Por exemplo, um compressor que espera dados binários não deve receber JSON diretamente sem uma etapa de serialização.

Tamanho e Limites

Verifique restrições impostas por transportes ou transformações (e.g., limite de tamanho de payload HTTP). A política deve rejeitar combinações que excedam esses limites antes da execução.

Ordem das Operações

A sequência das estratégias (Ex: `Transform` → `Compress` → `Transport`) é crítica. Alterar a ordem pode inviabilizar a auditoria ou a verificação de integridade dos dados.

Nomes e Extensões

Derive nomes e extensões de arquivos ou recursos de forma consistente, preferencialmente usando propriedades das interfaces (como `ICompressor.Extension`), para evitar erros de consumo ou compatibilidade.

A testabilidade é a espinha dorsal de qualquer arquitetura flexível. Ao adotar interfaces, abrimos portas para testes unitários e de integração mais eficientes, isolando componentes e simulando comportamentos de forma controlada. Aqui estão algumas diretrizes para garantir uma testabilidade exemplar:

Teste de Pipeline

Injete implementações "fake" (fakes ou mocks) para cada chave no catálogo e verifique rigorosamente a ordem e a correta passagem de dados entre as chamadas (Export → Transform → Compress → Transport).

Teste de Política

Utilize o padrão "table-driven test" para varrer políticas conhecidas. Garanta que o orquestrador seleciona as chaves corretas e que a composição das estratégias reflete o que foi definido na política.

Teste de Erro

Valide o comportamento do sistema para políticas inválidas ou incompletas. Certifique-se de que exceções apropriadas (`KeyNotFoundException` ou erros de domínio específicos) são lançadas e tratadas.

Evitar I/O Real

Em testes, todas as operações que envolvem I/O (entrada/saída), como transporte de dados via rede ou acesso a disco, devem ser substituídas por versões "fake" em memória para garantir rapidez, isolamento e repetibilidade dos testes.

Evoluções Naturais e Adaptação Contínua

À medida que os sistemas evoluem e as demandas aumentam, a arquitetura baseada em interfaces oferece uma fundação sólida para incorporar novas funcionalidades e otimizações. Explorar "evoluções naturais" significa adotar padrões e práticas que aprimoram a resiliência, a performance e a capacidade de gerenciamento do sistema. Aqui, destacamos três áreas chave para aprimoramento que podem ser facilmente integradas através do modelo de interfaces.

Medição e Observabilidade

A integração de interfaces como `IHealthCheck` e `IMetrics` nas estratégias permite que cada componente reporte seu estado de saúde e métricas de desempenho. Isso fornece visibilidade crucial para monitoramento em tempo real, depuração eficiente e tomada de decisões baseada em dados, garantindo que o sistema opere com máxima eficácia e resiliência.

Operações Assíncronas

A transição de operações síncronas (ex: `void Send`) para assíncronas (`Task SendAsync`), com a propagação de `CancellationToken`, é vital para sistemas de alta performance. Isso melhora a responsividade da aplicação, otimiza o uso de recursos e evita bloqueios, permitindo que as operações de I/O sejam executadas de forma não bloqueante.

Reconfiguração em Runtime

Capacitar a política de entrega para ser observável (por exemplo, através de um "file watcher" ou de um serviço de configuração centralizado) permite a reconfiguração do sistema em tempo real, sem a necessidade de reiniciar a aplicação. Essa flexibilidade é inestimável para ajustes rápidos de estratégias ou para a implantação contínua de novas políticas.

Essas evoluções demonstram como o design centrado em interfaces não apenas simplifica a implementação inicial, mas também prepara o terreno para a adaptação e o crescimento orgânico do software, mantendo-o ágil e robusto diante de desafios futuros.

FAQ e Decisões Conscientes de Design

À medida que aprofundamos no uso de interfaces em C#, surgem perguntas comuns sobre as melhores práticas e as decisões de design mais eficazes. Esta seção aborda as dúvidas frequentes e os princípios que guiam um design consciente, preparando o caminho para arquiteturas robustas e evoluíveis.

Quando usar classe abstrata e não interface?

Use uma classe abstrata quando houver um "ritual" fixo com passos invariantes e até três "ganchos" (hooks) claros para variação. Ela permite a implementação de um método template e o reuso de código entre as classes derivadas. Se a principal variação for "quais peças" o sistema usa, e não "como executar um passo do mesmo ritual", prefira interfaces combinadas com composição. Interfaces focam no "o quê", enquanto classes abstratas podem ditar o "como" em parte.

Implementação explícita não é “esconder sujeira”?

Não, pelo contrário. A implementação explícita de interfaces é uma ferramenta poderosa para evitar a poluição da API pública de uma classe com membros que são raramente usados ou que servem a propósitos diagnósticos. Ela também é fundamental para resolver colisões de nomes quando uma classe implementa múltiplas interfaces com métodos de mesmo nome. O importante é usar com critério e documentar claramente a intenção por trás de sua aplicação.

Default Interface Members: usar ou evitar?

Embora os Default Interface Members (DIMs) possam ser tentadores, evite-os para lógica complexa. Eles são aceitáveis para implementar fallbacks triviais ou fornecer metadados (como uma propriedade `Extension`), desde que a funcionalidade não quebre os consumidores existentes e não introduza ambiguidade binária. DIMs podem complicar a rastreabilidade e a testabilidade se mal utilizados, pois a implementação padrão pode ser substituída sem aviso.

Como prevenir interfaces gordas?

A aplicação do **Princípio da Segregação de Interfaces (ISP)** é a chave. Se um cliente utiliza menos de 30% dos membros de uma interface, isso é um forte indicativo de que a interface está "gorda" e precisa ser fatiada. O ideal é extrair contratos ortogonais e compô-los através de um orquestrador. Isso garante que os clientes não sejam forçados a depender de métodos que não utilizam, resultando em menor acoplamento e maior flexibilidade.

Um design bem-sucedido não é apenas aquele que funciona, mas também aquele que é fácil de entender, manter e evoluir. A seguir, apresentamos critérios objetivos para medir o sucesso de suas escolhas de design ao trabalhar com interfaces:

Independência do Cliente

O cliente do seu sistema não precisa ser modificado quando você alterna entre diferentes implementações de uma interface. Isso demonstra um baixo acoplamento e uma forte abstração.

Testabilidade Eficiente

Os testes de unidade e integração rodam de forma rápida e confiável, sem a necessidade de I/O real (acesso a banco de dados, rede, sistema de arquivos). Isso é um sinal de que os componentes podem ser isolados e simulados facilmente.

Ausência de "Switch por Tipo"

Não há `switch` statements ou sequências de `if-else if` baseadas no tipo de uma interface ou classe. Isso indica que o polimorfismo está sendo usado de forma eficaz, e novas implementações podem ser adicionadas sem alterar o código existente.

Complexidade Ciclomática Reduzida

Após refatorações para usar interfaces e composição, a complexidade ciclomática do código (uma métrica de complexidade de fluxo de controle) é menor ou estável, indicando um código mais simples e fácil de raciocinar.

Diferenças (Diffs) Pequenas e Localizadas

Quando uma mudança é necessária, os "diffs" (as alterações no código) são pequenos, localizados e afetam apenas o componente relevante. Isso prova que o sistema é modular e que as alterações são contidas, minimizando o risco de efeitos colaterais indesejados.

Encerramento: Interfaces em C# (Síntese Prática)

Concluímos nossa jornada didática sobre interfaces em C#, que nos guiou desde o reconhecimento de objetivos estáveis e peças alternáveis no cotidiano até a formalização desses conceitos em contratos de capacidade. A premissa central é que, se a variação reside nas "peças" que combinamos, interfaces com composição são a solução ideal. Por outro lado, se existe um "ritual" fixo com poucos pontos de extensão, classes abstratas podem ser mais apropriadas. Mantivemos o foco nos princípios de Segregação de Interfaces (ISP), baixo acoplamento e testabilidade facilitada por injeção de dependência simples, dublês de teste e repositórios alternáveis sem impactar os clientes.

Do ponto de vista de engenharia, os ganhos são inegáveis: clareza do código, onde os clientes programam para um contrato; evolutividade, permitindo a troca de implementações sem alterações ruidosas; e qualidade nos testes, que podem ser executados rapidamente e sem dependências de I/O real. Os "cheiros de código" como `switch/downcast`, métodos opcionais em interfaces ou interfaces "gordas" devem ser vistos como sinais para segregar funcionalidades e concentrar a decisão de qual implementação usar em um único ponto de composição. Quando a complexidade aumenta e exige combinações de etapas independentes, a composição por eixos (como exportar, transformar, comprimir, transportar) surge como uma evolução natural.

01

Cliente Independente

O cliente do seu sistema não precisou ser modificado ao alternar as implementações? Se sim, o design está no caminho certo.

02

Testes Rápidos

Os testes rodam de forma ágil, sem I/O de disco ou rede? Isso indica alta testabilidade e isolamento.

03

Ausência de Switch/Downcast

Não há `switch` ou `downcast` no consumo da interface? O polimorfismo está sendo bem aproveitado.

04

Contratos Coesos e Descritivos

Os contratos são pequenos, coesos e seus nomes contam uma história clara sobre sua capacidade? Isso reflete a aplicação do ISP.

05

Diferenças (Diffs) Localizadas

Ao realizar mudanças, os "diffs" são pequenos, localizados e legíveis? Sua arquitetura é modular e fácil de manter.

Se as respostas a este checklist forem positivas, você está aplicando interfaces com design consciente, coeso e preparado para o crescimento.