Hydrogen Sessions

#6 - Abstractions

Abstrações

Como é óbvio, o tema das "abstrações" está intimamente ligado com o desenvolvimento de uma biblioteca de componentes como o Hydrogen, na medida em que se pretende potenciar:

- Encapsulamento, ou seja, "publicar" uma API mas "esconder" a implementação concreta (para que esta possa ser alterada sem afetar os clientes).
- Reutilização, não criando barreiras à evolução dos componentes/funcionalidades.

Há muita teoria sobre este tema mas a prática é muito mais complexa do que pode parecer à partida. Até porque essa teoria é muitas vezes permeável a interpretações diferentes e mesmo as linguagens de programação (como o C#) acabam por "oferecer" formas distintas para atingir os mesmos fins.

É importante, no entanto, ter presente alguns aspetos fundamentais da teoria:

- SOLID: https://en.wikipedia.org/wiki/SOLID
- Encapsulation: https://deviq.com/encapsulation/
- Open-closed: https://www.oodesign.com/open-close-principle.html
- Dependency inversion: https://www.oodesign.com/dependency-inversion-principle.html
- Interfaces vs abstract classes: https://www.geeksforgeeks.org/difference-between-abstract-class-and-interface-in-c-sharp/

Interfaces

Regra geral, a definição da API de determinado componente como um interface deve ser a primeira opção. Pelas seguintes razões:

- Os interfaces são mais fáceis de fazer evoluir do que as classes abstratas (ou classes base), na medida em que não impõem comportamento como as segundas, que, por isso, será mais difícil de modificar no futuro.
- Ligam particularmente bem com o service container.
- Ligam particularmente bem com as assemblies abstractions (ver adiante).

Grande parte dos componentes do Hydrogen – como vimos no tópico dos serviços – estão definidos como interfaces. Por exemplo:

```
public partial interface IPipeline<TContext>
    where TContext : class
{
    #region Methods

    /// <summary>
    /// Executes the pipeline asynchronously for the specified context instance.
    /// </summary>
    /// <param name="context">The context that holds the pipeline data.</param>
    /// <returns>
    /// The <see cref="Task"/> that represents the asynchronous operation.
    /// </returns>
    32 references | Hugo Lourenço | 1 author, 1 change | 1 incoming change
    Task InvokeAsync(TContext context);

    /// <summary>
    /// Executes the pipeline synchronously for the specified context instance.
    /// </summary>
    /// <param name="context">The context that holds the pipeline data.</param>
    /// <remarks>
    /// Beware when executing pipelines synchronously using this method. Since
    /// the pipeline handlers are intrinsically asynchronous this may lead to
    /// unexpected behavior, depending on the handlers implementation.
    /// </remarks>
    15 references | Hugo Lourenço | 1 author, 1 change | 1 incoming change
    void Invoke(TContext context);

    #endregion
}
```

Como em qualquer API, é importante, pensar muito bem nos seguintes aspetos, considerando a necessidade de fazer a API evoluir/mudar no futuro.

- As propriedades e os métodos. Apenas o estritamente necessário.
- Os parâmetros.
- A documentação.
- Métodos assíncronos e métodos síncronos.
- Etc.

Classes abstratas (ou classes base)

Claro que haverá situações em que uma classe abstrata será uma melhor solução, particularmente, quando se pretender fornecer comportamento por defeito, com a possibilidade do developer (seja dentro do Hydrogen ou já fora) adaptá-lo de alguma forma.

Um exemplo evidente disso é o padrão double-derived utilizado no código gerado dos micro serviços, para permitir que o developer acrescente comportamento ou mesmo modifique todo o código gerado.

No Hydrogen há vários exemplos deste tipo de encapsulamento (ver ServiceClient por exemplo). O seguinte exemplo é mais simples mas ilustra um tipo de cenário em que a classe abstrata é uma boa solução.

```
public abstract partial class ServiceClientCredentials
{
    Public Fields

    Constructors

    #region Public Methods

    /// <summary>
    /// Allows to initialize the credentials depending on the specified service client.
    /// </summary>
    /// <typeparam name="T">The type of the service client.</typeparam>
    /// <param name="serviceClient">The service client instance that is using these credentials.</param>
    1 reference | Hugo Lourenço | 1 author, 1 change | 1 incoming change
    public virtual void InitializeServiceClient<T>(ServiceClient<T> serviceClient)
        where T : ServiceClient<T>
    {
    }

    /// <summary>
    /// Allows to apply the credentials before sending the request for the first time.
    /// </summary>
    /// <param name="request">The request that is being sent.</param>
    /// <param name="cancellationToken">The cancellation token.</param>
    /// <returns>
    /// A <see cref="Task"/> that represents the asynchronous operation.
    /// </returns>
    8 references | Hugo Ribeiro, 42 days ago | 2 authors, 2 changes | 1 incoming change
    public virtual Task HandleRequestAsync(HttpRequestMessage request, CancellationToken cancellationToken)...

    /// <summary>
    /// Allows to apply the credentials after sending the request based on the failure response received.
    /// </summary>
    /// <param name="request">The request that is being sent.</param>
    /// <param name="response">The response received after sending the request for the first time.</param>
    /// <param name="cancellationToken">The cancellation token.</param>
    /// <returns>
    /// A <see cref="Task{TResult}"/> that represents the asynchronous operation.
    /// A value indicating whether the request should be sent again.
    /// </returns>
    3 references | Hugo Lourenço | 1 author, 1 change | 1 incoming change
    public virtual Task<bool> HandleResponseAsync(HttpRequestMessage request, HttpResponseMessage response, CancellationToken cancellationToken)...

    #endregion
}
```

Aplicam-se, neste caso, as mesmas preocupações no desenho da API referidas antes para os interfaces, e ainda:

- Métodos abstratos versus métodos com implementação.
- A implementação por defeito deve considerar muito bem a forma como poderá (ou não) ser estendida nas classes derivadas.
- Os aspetos referidos de seguida.

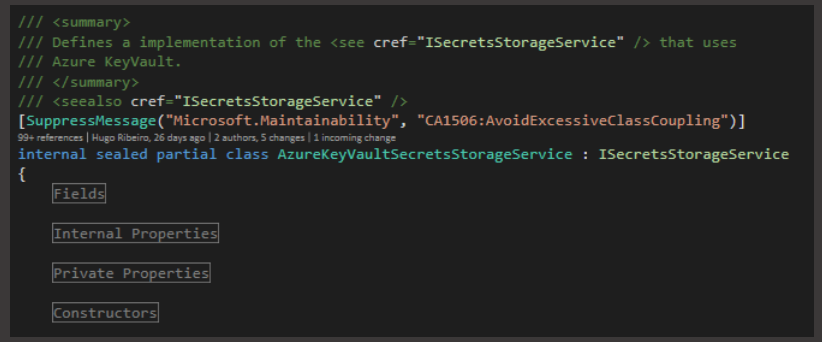
Public vs internal

Qualquer classe, propriedade, método, etc. que seja public faz parte da API do Hydrogen e não poderá ser modificada sem uma quebra de compatibilidade.

Por isso, esta é uma decisão crítica que deve ser sempre bem ponderada.

É muito fácil adicionar, por exemplo, determinada classe e torná-la pública por defeito, mesmo quando ela só é utilizada internamente. Isto é um erro. Deve assumir-se sempre que qualquer "coisa" pública vai ser utilizada pelos clientes do Hydrogen e, por isso, só as classes que são desenhadas explicitamente para esse fim, devem ser públicas. O que também é verdade para os membros (propriedades, métodos, etc.) de classes públicas.

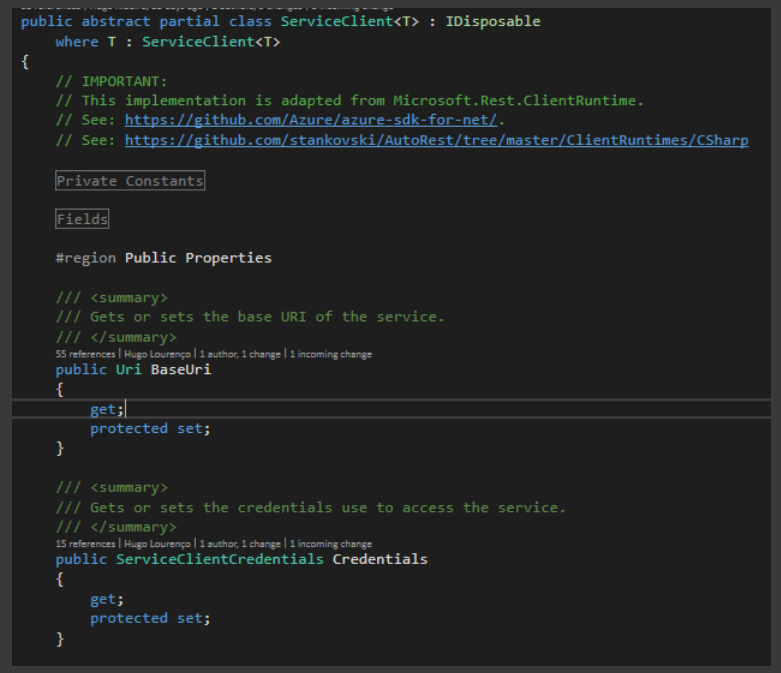
Tudo o resto deve ser internal por defeito.



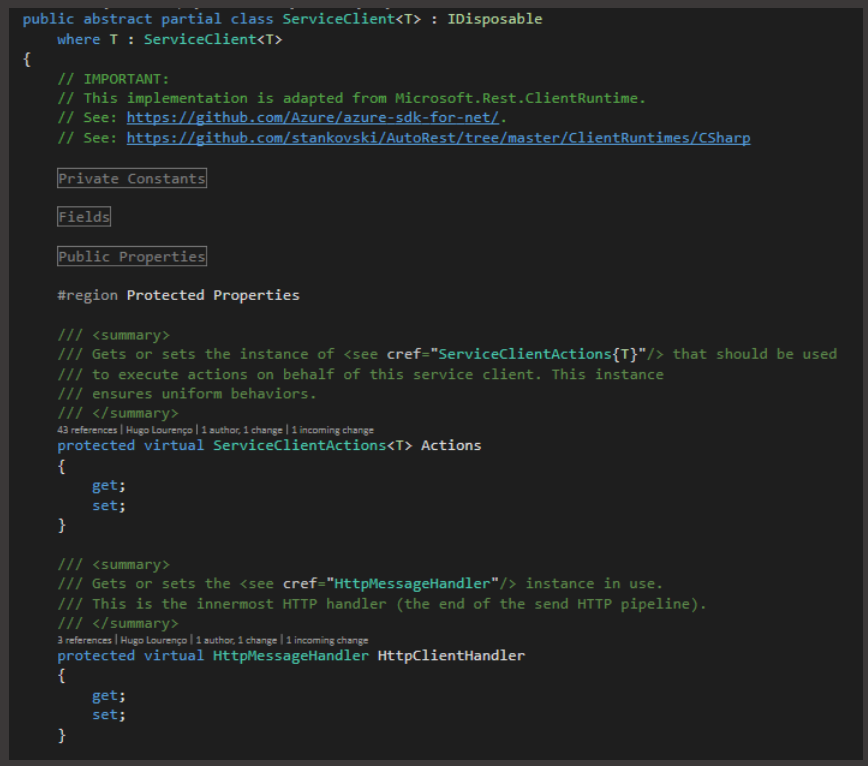
Public vs protected

Outra decisão importante é se os membros da classe abstrata devem ser públicos (acessíveis aos clientes) ou protected (acessíveis apenas às classes derivadas).

Não existe uma regra fácil para tomar esta decisão, mas é necessário tomar uma decisão concreta para cada um dos membros da classe, considerando que um membro protected será sempre mais fácil de modificar no futuro que um público (porque a "superfície" afetada será inferior).



```
public abstract partial class ServiceClient<T> : IDisposable
    where T : ServiceClient<T>
{
    // IMPORTANT:
    // This implementation is adapted from Microsoft.Rest.ClientRuntime.
    // See: https://github.com/Azure/azure-sdk-for-net/.
    // See: https://github.com/stankovski/AutoRest/tree/master/ClientRuntimes/CSharp

    Private Constants

    Fields

    #region Public Properties

    /// <summary>
    /// Gets or sets the base URI of the service.
    /// </summary>
    55 references | Hugo Lourenço | 1 author, 1 change | 1 incoming change
    public Uri BaseUri
    {
        get;
        protected set;
    }

    /// <summary>
    /// Gets or sets the credentials use to access the service.
    /// </summary>
    15 references | Hugo Lourenço | 1 author, 1 change | 1 incoming change
    public ServiceClientCredentials Credentials
    {
        get;
        protected set;
    }
```



```
public abstract partial class ServiceClient<T> : IDisposable
    where T : ServiceClient<T>
{
    // IMPORTANT:
    // This implementation is adapted from Microsoft.Rest.ClientRuntime.
    // See: https://github.com/Azure/azure-sdk-for-net/.
    // See: https://github.com/stankovski/AutoRest/tree/master/ClientRuntimes/CSharp

    Private Constants

    Fields

    Public Properties

    #region Protected Properties

    /// <summary>
    /// Gets or sets the instance of <see cref="ServiceClientActions{T}"/> that should be used
    /// to execute actions on behalf of this service client. This instance
    /// ensures uniform behaviors.
    /// </summary>
    43 references | Hugo Lourenço | 1 author, 1 change | 1 incoming change
    protected virtual ServiceClientActions<T> Actions
    {
        get;
        set;
    }

    /// <summary>
    /// Gets or sets the <see cref="HttpMessageHandler"/> instance in use.
    /// This is the innermost HTTP handler (the end of the send HTTP pipeline).
    /// </summary>
    3 references | Hugo Lourenço | 1 author, 1 change | 1 incoming change
    protected virtual HttpMessageHandler HttpClientHandler
    {
        get;
        set;
    }
```

Virtual

Finalmente, há a questão de selecionar os membros da classe abstrata que podem ser “overriden” nas classes derivadas, aplicando-lhes o modifier virtual.

Não faz muito sentido que uma classe abstrata não tenha nenhum (ou tenha apenas uma pequena porção) dos seus membros virtual. Claro que haverá casos em que o “comportamento base” é algo que não se pretende que possa ser modificado, mas essas devem ser exceções.

```
/// <summary>
/// Defines the base class for SMTP message handlers.
/// A SMTP message handler is capable of sending instances <see cref="EmailMessage"/> using SMTP.
/// </summary>
/// <seealso cref="IDisposable" />
17 references | Hugo Lourenço | 1 author, 1 change | 1 incoming change
public abstract class SmtpMessageHandler : IDisposable
{
    #region Public Properties

    /// <summary>
    /// Gets or sets the amount of time after which sending messages times out.
    /// The default value is 10 seconds.
    /// </summary>
    5 references | Hugo Lourenço | 1 author, 1 change | 1 incoming change
    public virtual TimeSpan Timeout
    {
        get;
        set;
    }

    #endregion

    Protected Properties

    Constructors

    Public Methods
```

```
/// <summary>
/// Defines the base class for an entity managed by the <see cref="ITableReference" />.
/// </summary>
/// <seealso cref="ITableEntity" />
11 references | Hugo Lourenço | 1 author, 1 change | 1 incoming change
public abstract class TableEntity : ITableEntity
{
    #region Public Properties

    /// <inheritdoc />
    [IgnoreProperty]
    7 references | Hugo Lourenço | 1 author, 1 change | 1 incoming change
    public string Key1
    {
        get;
        set;
    }

    /// <inheritdoc />
    [IgnoreProperty]
    7 references | Hugo Lourenço | 1 author, 1 change | 1 incoming change
    public string Key2
    {
        get;
        set;
    }
```

Dependências circulares

Um erro tradicional na construção de bibliotecas como o Hydrogen é o de separar mal as abstrações das implementações concretas (em termos físicos), o que leva rapidamente a uma situação de dependências circulares.

Sofremos este problema várias vezes na FW Elevation e no CoreLib, o que levou a adotar uma prática diferente no Hydrogen que importa explicar um pouco.

Uma dependência circular ocorre, por exemplo, quando um componente A depende de outro componente B, que depende de outro C, que por sua vez depende do componente A. Embora pareça absurdo, é relativamente fácil de ocorrer.

A solução aplicada no Hydrogen é a mesma que pode ser observada nas livrarias .NET Core (Microsoft.Extensions.* por exemplo) e consiste em aplicar o princípio Dependency Inversion também ao nível físico das assemblies, isto é, na separação das várias classes por assemblies diferentes.

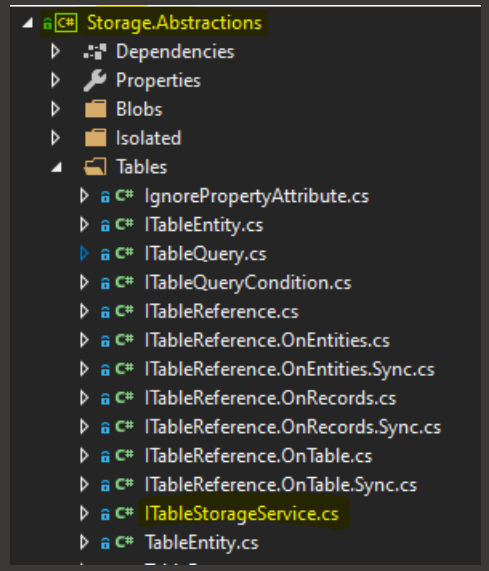
Para determinada funcionalidade XPTO:

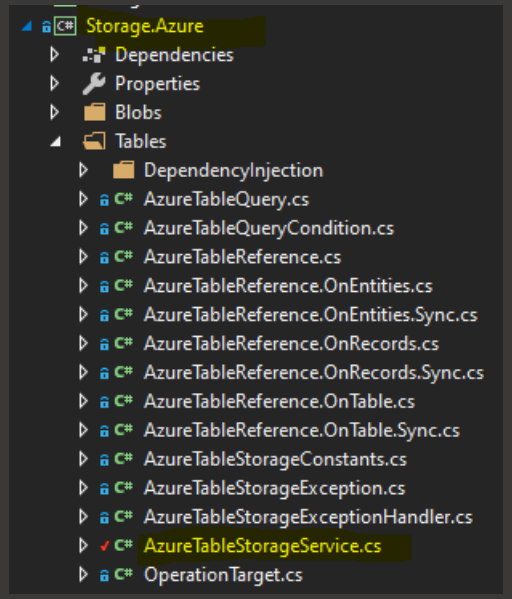
- A abstração – IXpto – é colocada numa assembly própria, com o sufixo “Abstractions” no nome (ex.: Xtpo.Abstractions).
- Qualquer dependência dessa abstração (como POCOs, enumerados, etc.) necessária à definição da API da abstração, é colocada na mesma assembly.
- A implementação concreta – ex.: DefaultXpto – é colocada noutra assembly.

Assim, a implementação concreta pode depender de quaisquer outros componentes – terá referências para Aaa.Abstractions, Bbb.Abstractions, etc. – e nunca acontecerá uma referência circular porque a implementação concreta dessas dependências, mesmo que dependa de IXpto, terá apenas uma referência para Xpto.Abstractions.

Exemplo

Este padrão está implementado um vários componentes. Como, por exemplo, os serviços de storage:





Boas práticas

1. Sempre que possível, preferir interfaces a classes base.
2. Sempre que possível, preferir membros internos a membros públicos.
3. Sempre que possível, preferir membros protected a membros públicos (ou internal).
4. Sempre que possível, tornar os membros de classes base virtual.
5. Separar sempre fisicamente as abstrações das implementações concretas (exceto em casos muito simples e/ou em casos em que as implementações concretas não têm dependências para outros componentes).