Departamento de Engenharia Informática

Instituto Superior de Engenharia do Porto

Instituto Politécnico do Porto

Introdução ao Desenvolvimento de Aplicações Baseadas em Componentes Usando a Plataforma .NET

Paulo Sousa

—♦—

Outubro de 2004

© 2002, 2004 Paulo Sousa
Departamento de Engenharia Informática
Instituto Superior de Engenharia do Porto (ISEP/IPP)
Rua Dr. António Bernardino de Almeida, 431
4200-072 PORTO
Portugal
Tel. +351 228 340 500
Fax +351 228 325 219

Criado em Outubro, 2003
Última modificação em 07 Outubro, 2004 **(v 1.3)**
Email: psousa@dei.isep.ipp.pt
URL: http://www.dei.isep.ipp.pt/~psousa/aulas/ADAV/guiao_componentes_dotNet.pdf

Índice

Índice de Figuras

Índice de Tabelas

1 Introdução

Este documento pretende introduzir os aspectos relacionados com o desenvolvimento de soluções em ambiente .net utilizando a ferramenta de desenvolvimento Visual Studio .net 2003.

Adicionalmente, o documento apresenta guiões de trabalho para exemplificar a criação de aplicações .net baseadas em componentes.

2 A Plataforma .net

A plaforma .net é um ambiente de execução, CLR (Common Language Runtime) (Figura 1) e uma plataforma de desenvolvimento, *base class library* (Figura 2), i.e., conjunto de classes base sobre a qual se desenvolve

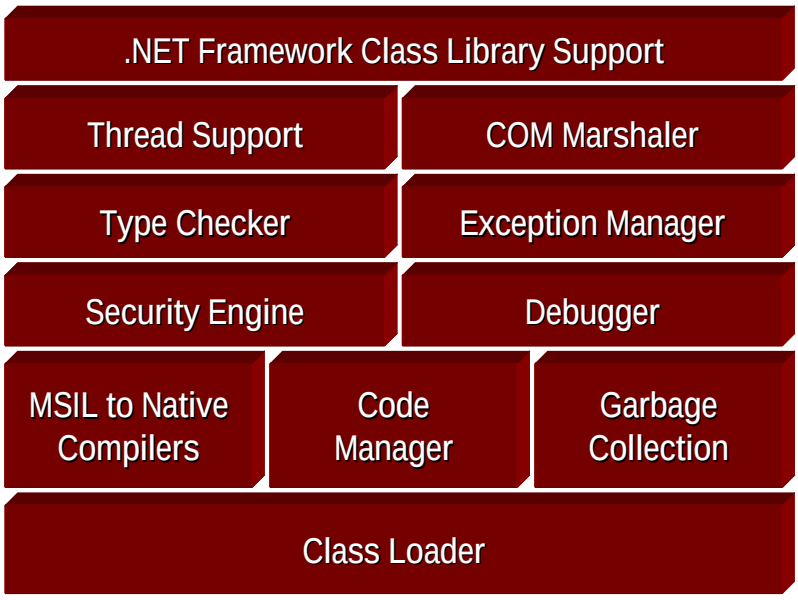

Figura 1 - CLR

As classes da biblioteca de classes base do .net estão organizadas em *namespaces*[1] (Figura 2). Um *namespace* é um "espaço de nomes", ou seja, um contexto de identificação de tipos que permite organizar o código de forma estruturada (hierárquica) e agrupar logicamente tipos (ex., classes) relacionadas. Por exemplo, todas as classes base para comunicações via rede estão dentro do *namespace* `System.Net` ou em *namespaces*

[1] Para ver um mapa com toda a hierarquia de *namespaces* da plataforma .net consultar o URL http://msdn.microsoft.com/library/en-us/cpref/html/cpref_start.asp.

dentro desse em caso de especialização (ex., `System.Net.Sockets`). Os *namespaces* não estão relacionados com herança nem com os *assemblies* (é possível ter classes de um mesmo *namespace* em *assemblies* diferentes).

Os *namespaces* são uma técnica de organizar o código e evitar conflito de nomes. Por exemplo, ao desenvolver aplicações web ou Windows é possível criar páginas com controlos do tipo *text box*, no entanto a classe que representa uma *text box* é diferente para cada um dos tipos de aplicações. As classes podem ter o mesmo nome, ex. `TextBox`, desde que estejam em *namespaces* diferentes, ex., `System.Web.UI` e `System.Windows.Forms`.

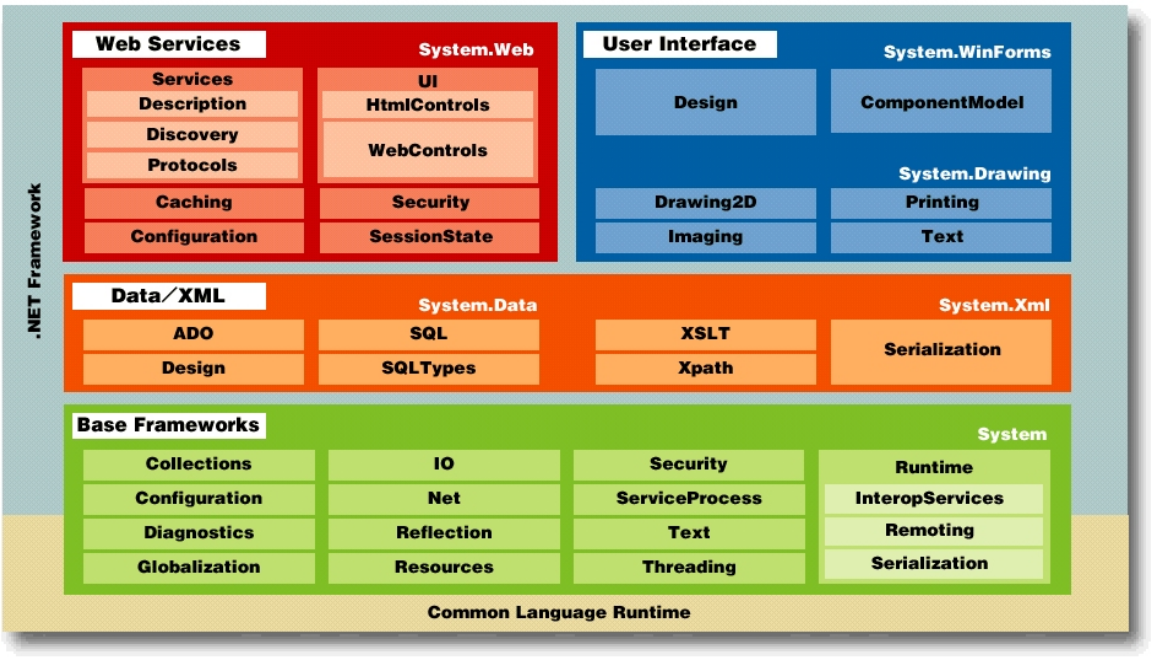

Figura 2 - namespaces existentes na plataforma .net

Para se referirem no código a uma classe devem utilizar o nome completo da classe (incluindo o *namespace*) . Por exemplo:

```
System.String x = “olá mundo”;
System.Random r = new System.Random();
```

No entanto, para simplificar a codificação, é possível indicar ao compilador que se deseja utilizar classes de um determinado *namespace*, usando a directiva `using` (`import` em visual basic.net), e dessa forma é possível referir essas classes sem o nome completo. Por exemplo:

```
using System;
```

```
...
String x = “olá mundo”;
Random r = new Random();
```

Desta forma a escrita do código fica mais simplificada.

Um outro conceito importante na plataforma .net é o conceito de ***assembly***. Um *assembly* é um bloco de distribuição binária da aplicação (pode ser um executável ou uma DLL), que contém uma colecção de tipos e de recursos (ex., imagens JPEG), é “versionavel”, e ao qual podem ser afectadas permissões de segurança no contexto .net.

A figura seguinte representa o processo de execução de um programa na plataforma .net.

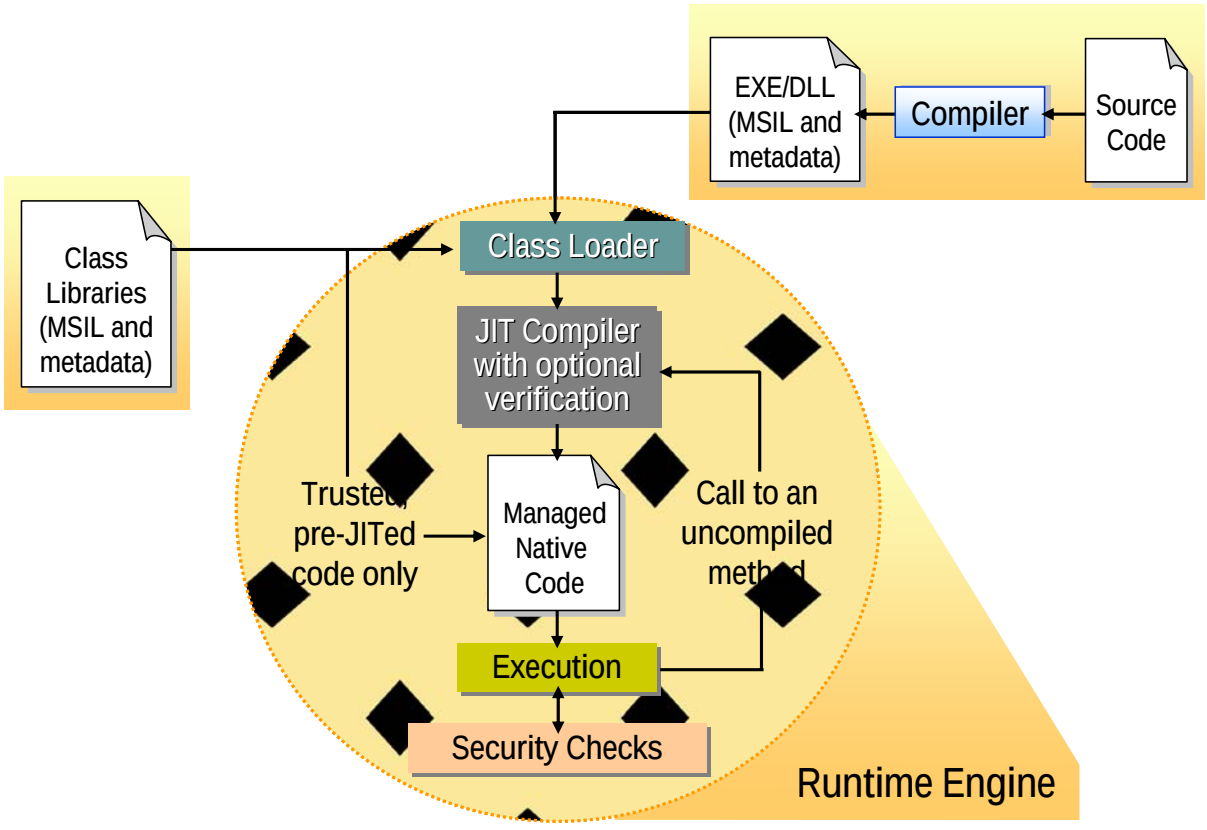

Figura 3 - processo de compilação e execução em .NET

Os tipos de dados dividem-se em duas categorias:

- *Value type*: contém directamente os dados e **não** pode ser `null`

- *Reference type*: contém referência para objecto e pode ser `null`

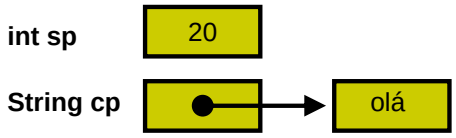

Figura 4 - exemplo de um value type e um reference type

Como a maior parte das variáveis dos programas .net são tipos de referência e não de valor, é necessário ter alguns cuidados na atribuição e comparação de variáveis. Veja-se por exemplo o seguinte extracto de código C# :

```
Pessoa p1 = new Pessoa(“António”);
Pessoa p2;
p2 = p1;
p2.Nome = “joão”;
System.Console.WriteLine(“p1.Nome : {0}”, p1.Nome);
```

Como as variáveis de classe são referências, a atribuição **não copia** os objectos. Ambas as variáveis referenciam o mesmo objecto, logo a última linha vai escrever no ecrã:

```
p1.Nome : joão
```

e não:

```
p1.Nome : António
```

Na figura seguinte podemos ver graficamente aquilo que acabou de ser explicado:

```
Pessoa p1 = new Pessoa(“António”);
Pessoa p2;
p2 = p1;
p2.Nome = “joão”;
```

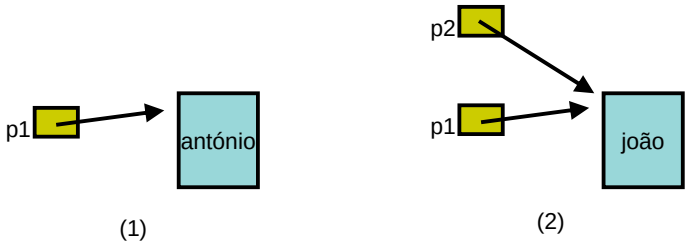

Figura 5 - atribuição de referências

Um outro cuidado adicional tem a ver com os testes de igualdade entre variáveis. Como as variáveis são referências, o operador de igualdade[2] é na realidade um teste de **identidade**, ou seja, verifica se as duas variáveis referenciam o mesmo objecto.

```
Pessoa p1 = new Pessoa("António");
Pessoa p2 = p1;
Pessoa p3 = new Pessoa("António");
Pessoa p4 = new Pessoa("luisa");

System.Console.WriteLine("p1 == p2 : {0}", p1 == p2);
System.Console.WriteLine("p1 == p3 : {0}", p1 == p3);
System.Console.WriteLine("p1 == p4 : {0}", p1 == p4);
```

O extracto de código anterior produziria o seguinte conteúdo:

```
p1 == p2 : true
p1 == p3 : false
p1 == p4 : false
```

No entanto, em determinadas situações, o que o programador deseja é um teste de **igualdade de valor** e não de identidade de objectos. Nessa situação teremos que definir explicitamente um método para comparação por valor ou redefinir a semântica do método `Equals()` da nossa classe.

Supondo que a classe pessoa tem dois atributos (nome e data de nascimento) a comparação entre dois objectos `Pessoa` seria implementada da seguinte forma:

```
public override bool Equals(object o)
{
      if (o is Pessoa)
      {
            if (this.nome == o.nome && this.dtNasc == o.dtNasc)
                  return true;
            else
                  return false;
      }
      else
            return false;
}
```

[2] Em C#, além do operador de igualdade (i.e., ==) é possível comparar dois objectos usando o método `Equals()` definido na classe `Object`. Para mais informação ver http://msdn.microsoft.com/library/en-us/cpref/html/frlrfSystemObjectClassEqualsTopic1.asp?frame=true. De notar, para os programadores Java, que em Java o operador == efectua um teste de identidade enquanto o método `equals()` efectua um teste de igualdade. Em C#, tanto o operador == como o método `Equals()` têm o mesmo comportamento semântico.

Como foi dito anteriormente, em C#, o operador == e o método `Equals()` têm o mesmo comportamento semântico. Assim sendo, ao redefinir um deles devemos também redefinir o outro. Neste caso, deveria acrescentar-se à classe `Pessoa` o seguinte operador:

```
public static bool operator==(Pessoa p1, Pessoa p2)
{
      if (p1 != null && p2 != null)
      {
            if (p1.nome == p2.nome && p1.dtNasc == p2.dtNasc)
                  return true;
            else
                  return false;
      }
      else
            return (p1 == null && p2 == null ? true : false);
}
```

De notar que segundo as regras para redefinição de operadores ao redefinir o operador == também deve ser redefinido o operador `!=`. Para mais informação ver http://msdn.microsoft.com/library//en-us/csref/html/vcwlkoperatoroverloadingtutorial.asp.

3 A Linguagem C#

3.1 Introdução

- Nova linguagem tendo por base o C/C++
    - Também vai buscar inspiração ao Java ;-)
    - Mantém o investimento e *know-how* existente
- Código mais “limpo”
- Construções sintácticas especiais para tirar partido do *framework*
- Tudo são objectos
- Ficheiros com extensão .cs
- Declaração e definição de métodos no mesmo ficheiro

3.2 Tipos de dados e operadores

A tabela seguinte apresenta os diferentes tipos de dados existentes no C#.

Tabela 1 – tipos de dados existentes no C#

Categoria	Tipos existentes
Objectos	`object`
alfanuméricos	`string`, `char`
Inteiros com sinal	`sbyte`, `short`, `int`, `long`
Inteiros sem sinal	`byte`, `ushort`, `uint`, `ulong`
Virgula flutuante	`float`, `double`, `decimal`
boleanos	`bool`

- Estes tipos são *alias* para os tipos definidos na *framework*
    - Ex., `int` == `System.Int32`
- Tipos Enumerados definidos pelo utilizador
    - Fortemente "tipados"
    - Sem conversão automática para `int`
    - Suportam operadores +, -, ++, --, &, |, ^, ~
    - Pode-se definir tipo de dados base (`byte`, `short`, `int`, `long`)

```
enum Color : byte
{
      Red = 1,
      Green = 2,
      Blue = 4,
      Black = 0,
      White = Red | Green | Blue
}
```

A tabela seguinte apresenta os operadores existentes em C# (são os mesmo do C++). Em C# é possível redefinir operadores.

Tabela 2 – operadores existentes em C#

Categoria	Operadores
Atribuição	=

Relacionais	< <= > >= == !=
Lógicos	&& || !
Aritméticos	+ - * / % += -= *= /= ++ --

3.3 Sintaxe

3.3.1 Constantes

- Pré-definidas (`null, true, false`)
- De utilizador:

```
const string Ver = “1.0b”;
```

3.3.2 Classes e namespaces

- Organização do código dentro de classes
- Classes organizadas dentro de *namespaces*

```
namespace Demo
{
	public class MyClass
	{
		...
	}
}
```

3.3.3 Construtores

- Seguem as regras do C/C++
- Mesmo nome da classe
- Sem tipo de retorno
- Podem ter ou não argumentos

```
public class MyClass
{
	...
	public MyClass() { ... }
	public MyClass(string title) { ... }
```

```
}
```

3.3.4 Métodos

- Sintaxe semelhante ao C/C++
- Podem ser públicos ou privados
- Suporta *overloading*

```
public class MyHelloWorld
{
      ...
      public void SayHello()
      { ... }

      private void SetTitle(String Title)
      { ... }

      private void SetTitle(String title, Color c)
      { ... }
}
```

3.3.5 Passagem de parâmetros

O C# suporta passagem de argumentos para um método, por valor e por referência. Neste último caso, é possível indicar se o parâmetro é apenas de saída - `out` - ou de entrada e saída - `ref`.

```
// passagem por valor
public void func1(int x)
{ ... }

// passagem por referência – apenas de saída
public void func2(out int x)
{ ... }

// passagem por referência – entrada e saída
public void func2(ref int x)
{ ... }
```

3.3.6 Herança

- Apenas existe herança simples, ou seja, uma classe apenas pode derivar de outra.

```
public class MyClassBase
{
      ...
      public void Func()
      { ... }
}
```

```
public class MyClassDeriv : MyClassBase
{
	...
	// explicitamente indicar a ocultação do método da classe base
	public new void Func()
	{... }
}
```

- Métodos **não** são virtuais por defeito

```
public class MyClassBase
{
	...
	// indicar explicitamente que o método é virtual
	public virtual void Func()
	{ ... }
}

public class MyClassDeriv : MyClassBase
{
	...
	// indicar explicitamente a redefinição do método virtual da
	classe base
	public override void Func()
	{ ... }
}
```

- Aceder aos métodos da classe base

```
public class MyClassDeriv : MyClassBase
{
	...
	public override void Func()
	{
		base.Func();
		...
	}
}
```

3.3.7 Propriedades

- Sintaxe alternativa para acesso a membros de dados da classe mas com as vantagens dos métodos

```
public class Button : Control
{
	// atributo privado da classe
	private string caption;

	// propriedade pública
	public string Caption
```

```
        {
                // acessor ou getter
                get { return caption; }

                // mutator ou setter
                set { caption = value; Repaint(); }
        }
        ...
}
```

3.3.8 Objectos

- criação de objectos

```
// definição da classe
public class MyClass { ... }

// definição da variável
MyClass obj;

// criação do objecto
obj = new MyClass();
```

- aceder ao próprio objecto

```
// utilização da pseudovariavel
this.title = “ADAV”;
```

3.3.9 Arrays

- Suportados ao nível da biblioteca base de classes em `System.Array`

```
// declaração do vector
string[] vec;

// criação do vector
vec = new string[10];

// número de elementos pode ser dinâmico
vec = new string[n];
```

3.3.10 Ciclos

```
// repetição n vezes
for (int x = 0; i < vec.Length; i++)
      Console.WriteLine(vec[i]);

// repetição condicional
int i = 0;
```

```
while (i < vec.Length)
{
      Console.WriteLine(vec[i]);
      i++;
}

// enumeração
foreach (String x in vec)
      Console.WriteLine(x);
```

3.3.11 Condicionais

```
// teste de decisão
if (i < vec.Length)
      Console.WriteLine(vec[i]);
else
      Console.WriteLine("Erro!!!");

// teste múltiplo
switch (x)
{
      case 1: ...; break;
      case 2: ...; goto case 3; // fall trough explícito
      case 3: ...; break;
      default: ...; break;
}
```

3.3.12 Comentários em XML

- Suportados pelo Visual Studio .net
- Documentam o código (classes e métodos)

```
class MinhaClasse
{
      /// <summary>
      /// Returns the attribute with the given name and
      /// namespace</summary>
      /// <param name="name">
      /// The name of the attribute </param>
      /// <param name="ns">
      /// The namespace of the attribute, or null if
      /// the attribute has no namespace</param>
      /// <return>
      /// The attribute value, or null if the attribute
      /// does not exist</return>
      /// <seealso cref=" ="GetAttr(string)"/>
      public string GetAttr(string name, string ns) {
            ... ...
      }
}
```

4 Desenvolvimento utilizando o visual studio

4.1 IDE

O ambiente de desenvolvimento do visual studio (Figura 6) é uma evolução do visual studio 6, que permite geração de código nativo windows e .net, no qual é possível criar soluções[3] multi-projecto (em que cada projecto pode ser numa linguagem de programação distinta). Permite a criação de diferentes tipos de projecto (ex., aplicações de linha de comando, aplicações windows e aplicações web).

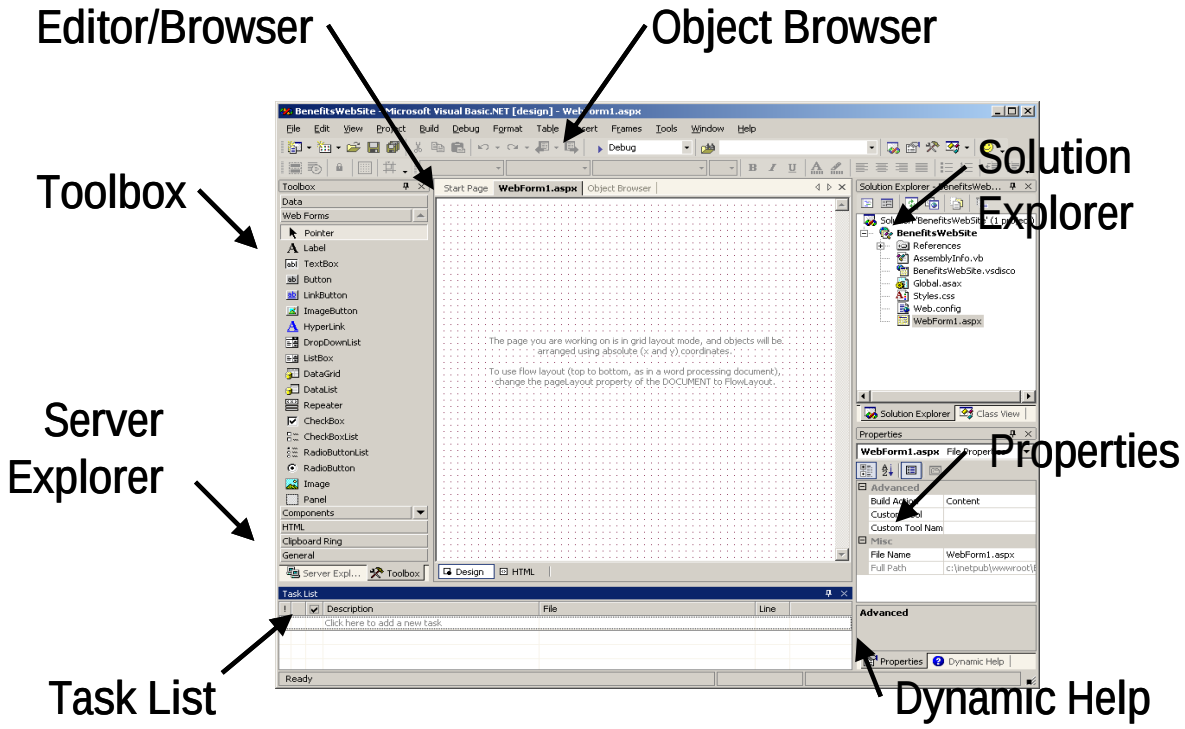

Figura 6 - Visual Studio .net

4.2 "Olá Mundo" em Consola

Criar um novo projecto do tipo "Console Application" em C#

[3] No visual studio, uma solução (*solution*) funciona como um espaço de trabalho no qual podem existir vários projectos.

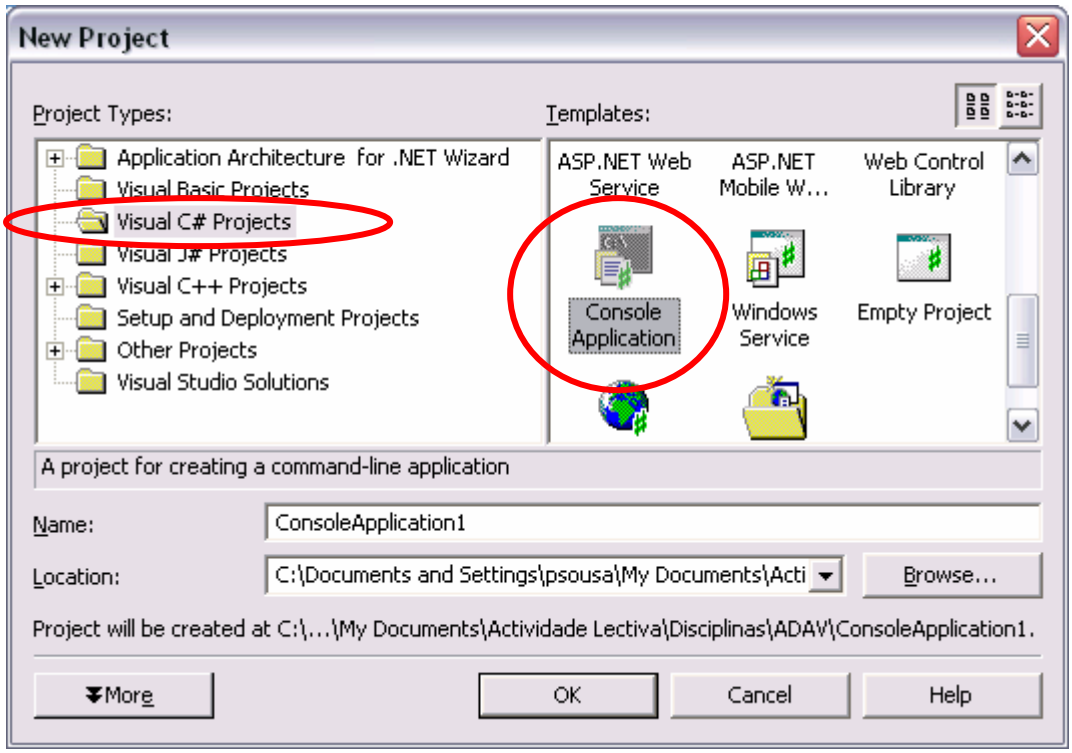

Figura 7 – criar um projecto tipo consola em C#

O Visual Studio vai gerar uma solução com um projecto contendo vários ficheiros. No ficheiro “class1.cs” existe já uma classe definida com um método `Main()`.

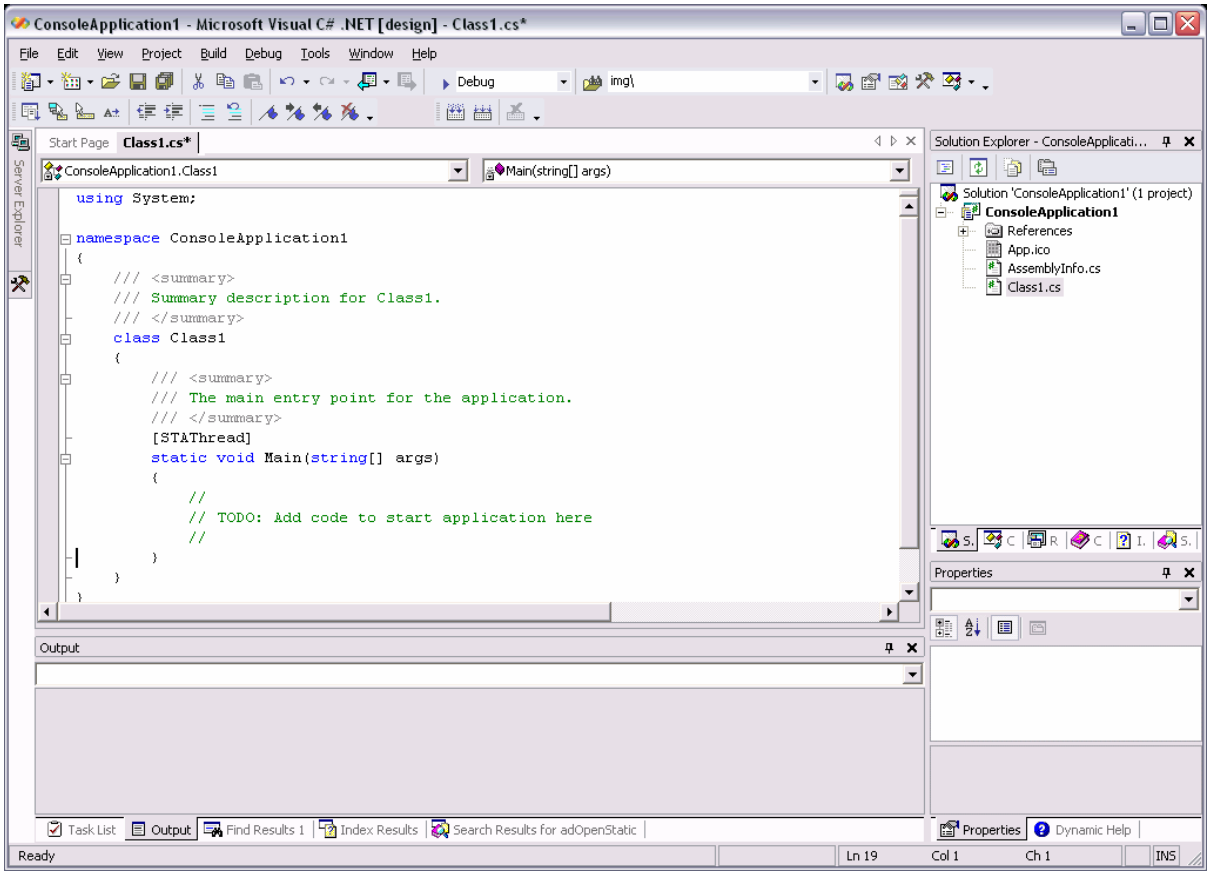

Figura 8 – código gerado automaticamente para projectos consola em C#

Colocar na implementação do método `Main()` a seguinte linha de código.

```
System.Console.WriteLine("Olá mundo!");
```

Compilar e testar.

Caso se queira criar novas classes no projecto deve-se “clicar” com o botão do lado direito do rato em cima do projecto, escolher a opção Add → Add Class e criar uma classe (Figura 9) ou escolher a opção File → Add New Item e escolher o *template* “Class” da categoria “Local Project Itens” (Figura 9).

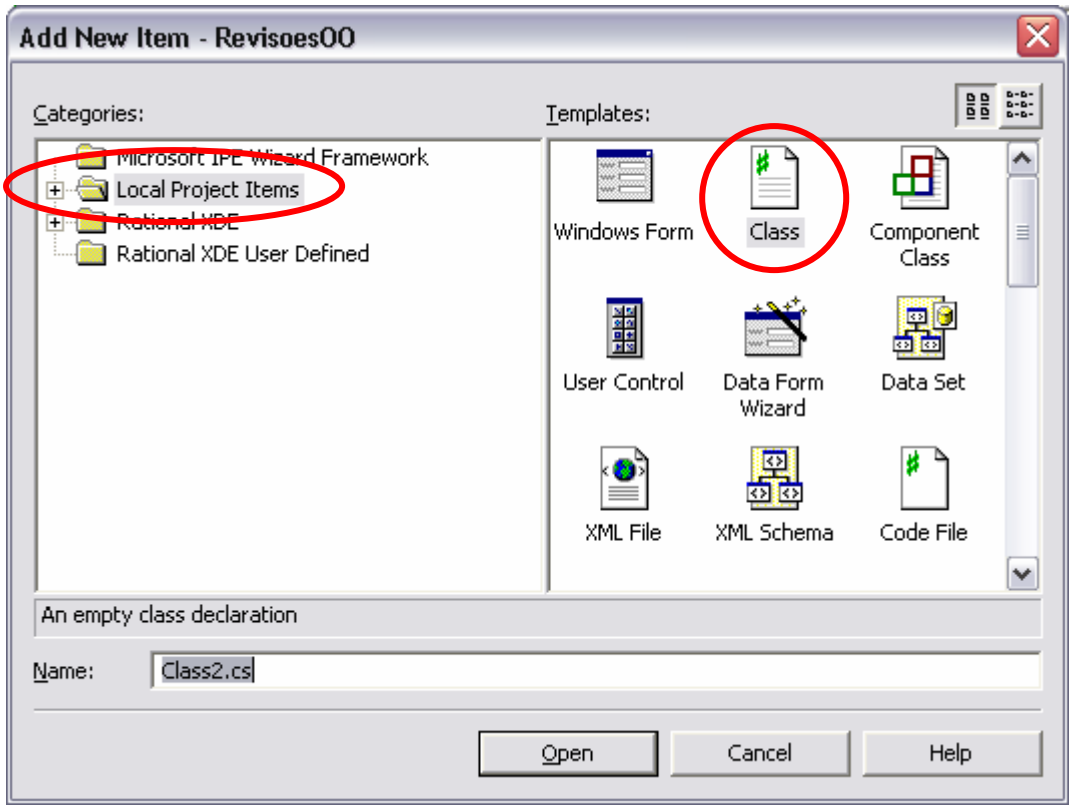

Figura 9 - adicionar uma nova classe a um projecto

No visual studio, o *solution explorer* permite visualizar a estrutura do projecto em termos de ficheiros (Figura 10) enquanto o *class explorer* permite visualizar a estrutura do projecto em termos de *namespaces* e classes (Figura 11).

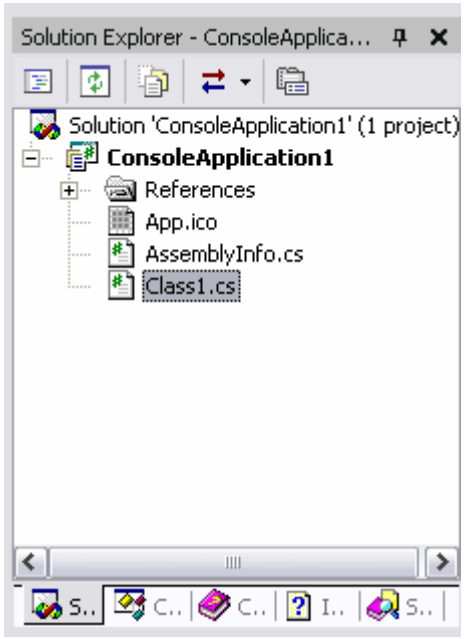

Figura 10 - solution explorer

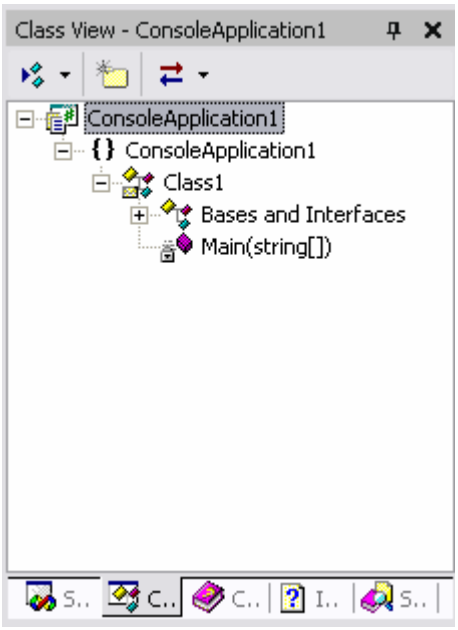

Figura 11 - class explorer

As classes criadas no Visual Studio pertencem sempre a um *namespace*. Por omissão o visual studio cria um *namespace* com o mesmo nome do projecto (que é também por omissão o nome do *assembly* a gerar), no entanto é possível modificar o *namespace* para novas classes nas propriedades do projecto (Figura 12).

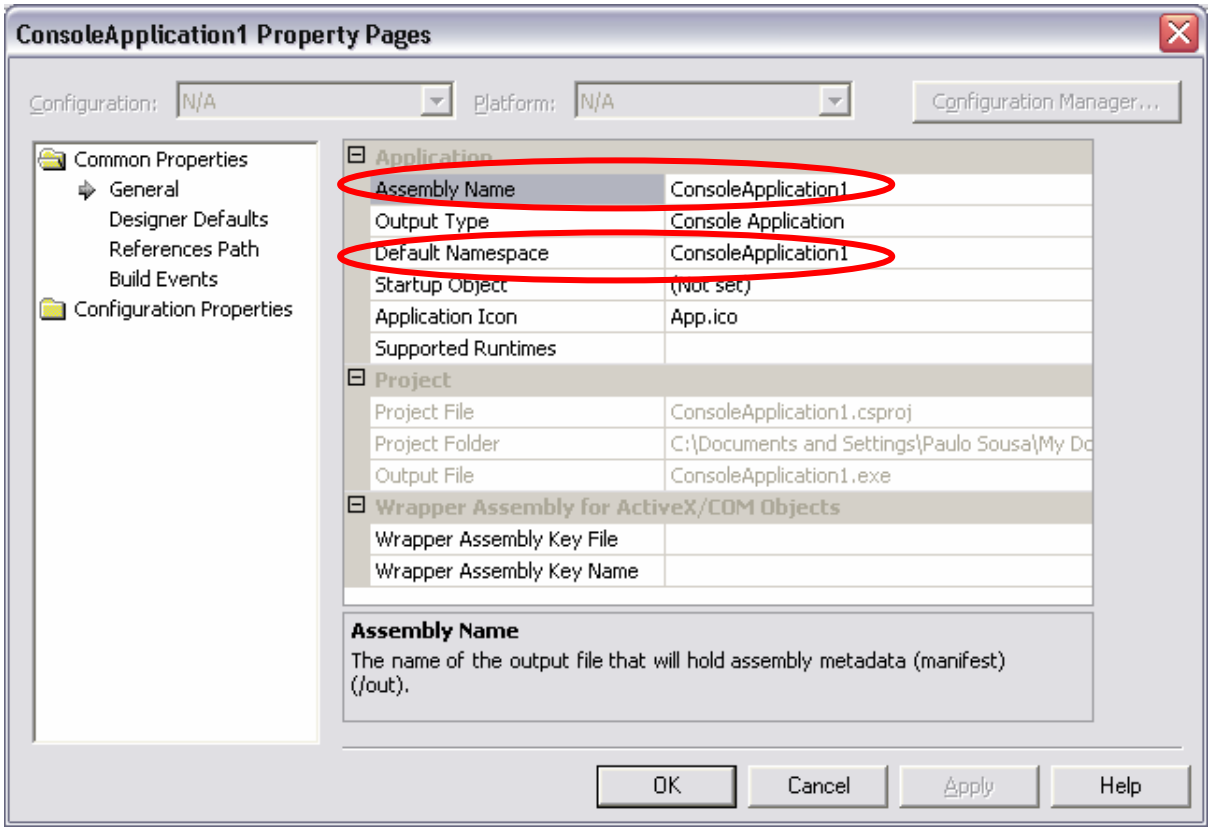

Figura 12 - propriedades de um projecto no visual studio (1)

Nos laboratórios do DEI, dará jeito criarem os projectos na vossa área de trabalho na rede, mas dirigir o resultado de compilação para um directório temporário na máquina local. Para isso podem alterar nas propriedades do projecto o "output path" (Figura 13).

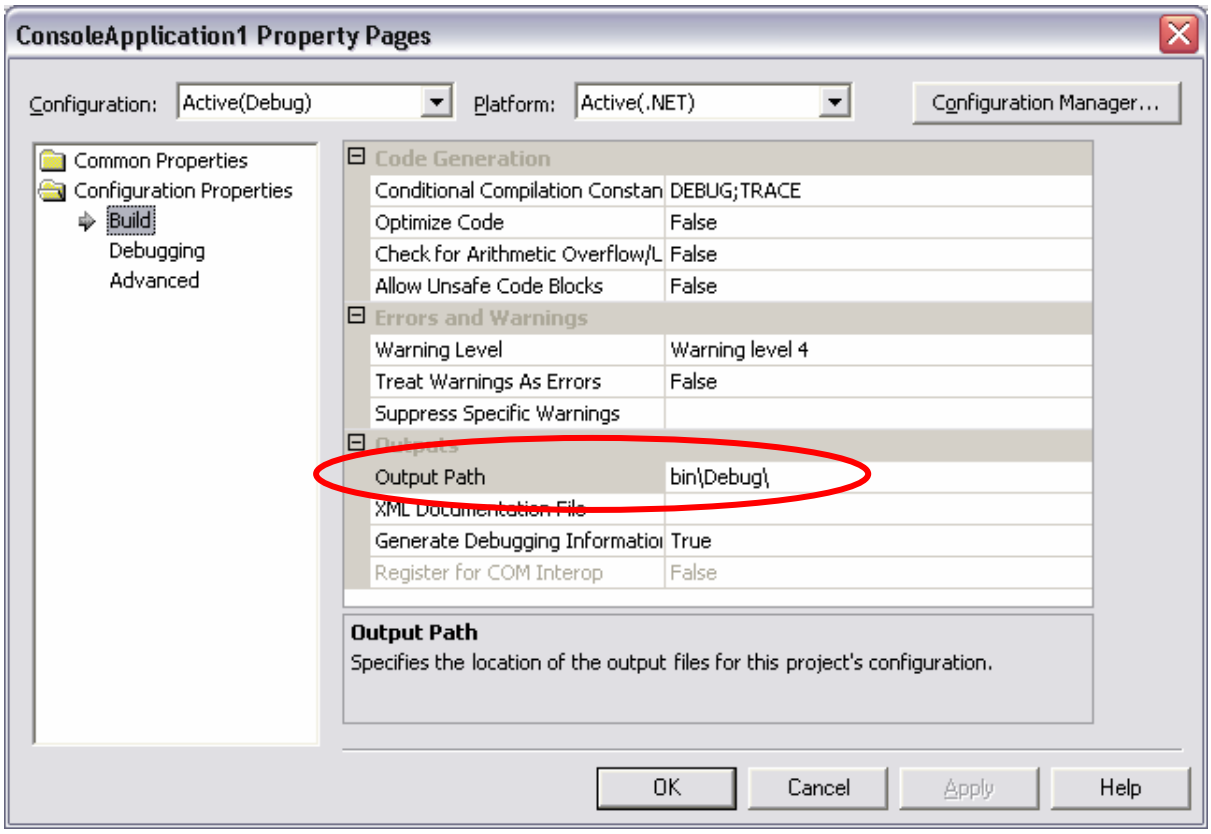

Figura 13 - propriedades de um projecto no visual studio (2)

4.3 Exercícios de revisão de conceitos OO

1 Escreva em C# o código que define uma classe de objectos `Pessoa` que representa pessoas. Considere atributos como 'nome', 'data de nascimento', BI e NIF. Implemente os métodos que julgar necessários não esquecendo que os atributos são privados a cada objecto. Escreva ainda um pequeno programa que permita criar objectos desta classe.

2 Tal como no exercício anterior, escreva em C# o código que define uma classe de objectos `Endereco` que representa endereços. Especifique os atributos necessários e implemente os métodos necessários.

3 Usando as duas classes de objectos dos dois exercícios anteriores escreva também em C# o código que define uma classe de objectos `Contacto` que representa contactos de pessoas. Não se esqueça de incluir atributos como o telefone. Reflicta sobre quais os atributos desta classe, bem como, quais os

parâmetros aconselháveis para cada um dos seus métodos à luz do conceito OO de encapsulamento e do facto da linguagem usar tipos de referência e não tipos de valor. Deve também escrever um pequeno programa que permita criar objectos desta classe.

4 Implemente em C# uma classe de objectos que permita armazenar um conjunto de referências a objectos `Contacto`. Internamente estes objectos são armazenados numa colecção do tipo `System.Collections.ArrayList` (procure no *help* do Visual Studio informação sobre como utilizar esta classe). Esta classe deve ter, além dos construtores necessários, os seguintes métodos:

`Add` – adiciona um contacto à lista

`AddAt` – adiciona um contacto numa dada ordem

`At` – devolve o contacto numa dada posição

`Find` – devolve o 1º contacto que encontra que possua a chave de pesquisa desejava (decida que campo utilziar como chave de pesquisa)..

Reflicta sobre quais os parâmetros aconselháveis para cada um destes métodos à luz do conceito OO de encapsulamento. Escreva um pequeno programa que crie pelo menos um objecto desta classe, insira contactos e procure por um dado contacto.

5 Guião de trabalho: Componente para operações aritméticas

5.1 introdução

Antes de avançar para o desenvolvimento de uma aplicação baseada em componentes em .net convém relembrar o conceito de componente e verificar como é que tal pode ser mapeado na tecnologia .net.

Segundo alguns autores, um componente é:

- "A software package which offers **service** through **interfaces**" [4]

[4] Peter Herzum and Oliver Sims, "Business Components Factory: A Comprehensive Overview of Component-Based Development for the Enterprise", John Wiley & Sons, Incorporated, 1999.

- "A coherent package of software artifacts that can be independently developed and **delivered as a unit** and that can be **composed**, unchanged, with other components to build something larger" [5]
- "A component is a **unit of composition with contractually specified interfaces** and explicit context dependencies only. A software component can be **deployed independently** and is subject to **composition** by third parties." [6]

De acordo com estas definições, **um componente é uma unidade de distribuição em formato binário de código, que fornece serviços através de um contrato bem definido** (i.e., uma interface). Por sua vez, **uma interface é um conjunto coerente de métodos**. Por conjunto coerente entenda-se métodos relacionados com um mesmo serviço e/ou trabalhando com entidades de um mesmo domínio. Um componente é utilizado por terceiros aos quais se dá o nome de "cliente", **um cliente é uma aplicação ou outro componente que faz uso dos serviços de um componente**.

Em .net para responder a estes requisitos temos (Tabela 3):

Tabela 3 - mapeamento entre requisitos para componentes e a tecnologia .net

Requisito	Mapeamento .net
Unidade de distribuição binária	*Class library*
Fornecimento de serviços	Classes
Contrato explícito	Interfaces
Reutilização	Referência à *class library*
Composição por terceiros	Composição e especialização (herança) de classes

Para uma breve introdução a estes conceitos aconselha-se a consulta do glossário disponível na Microsoft Developer Network Library[7] em http://msdn.microsoft.com/library/en-us/netstart/html/cpconGlossary.asp.

[5] D.F. D'Souza and A.C. Wills, "Objects, Components, And Frameworks with UML – The Catalysis Approach" Addison-Wesley, 1998.

[6] C. Szyperski, "Component Software: Beyond Object-Oriented Programming" Addison-Wesley, 1998.

[7] MSDN : http://www.msdn.microsoft.com/library

5.2 Solução

No visual studio, uma solução (*solution*) funciona como um espaço de trabalho no qual podem existir vários projectos de diferentes tipos (executáveis, bibliotecas de classes, etc.) para um mesmo problema. Vamos então criar uma solução em branco (i.e., sem projectos), usando a opção de menu File → New → Blank solution .

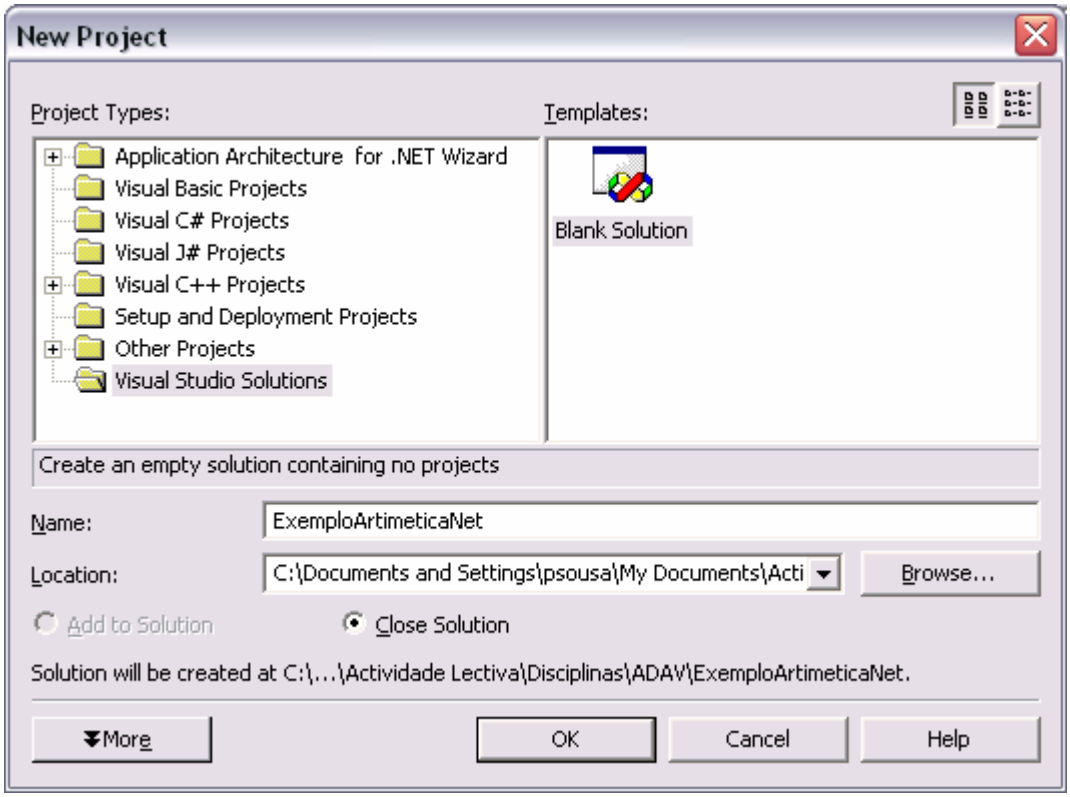

Figura 14 – criar uma "solução" sem projectos iniciais

5.3 Class Library

À solução anterior adicionar um projecto para o nosso componente (*Class Library*) usando File → New → Project.

NOTA: *Podem encontrar regras de codificação .net (ex., nomenclatura de classes) para* class libraries *em* http://msdn.microsoft.com/library/en-us/cpgenref/html/cpconnetframeworkdesignguidelines.asp*. Associado a este conjunto de regras existe uma ferramenta intitulada FxCop (disponível em* http://www.gotdotnet.com/team/fxcop/*) que executa a validação de um* assembly *.net no que toca às regras de codificação e algumas boas práticas.*

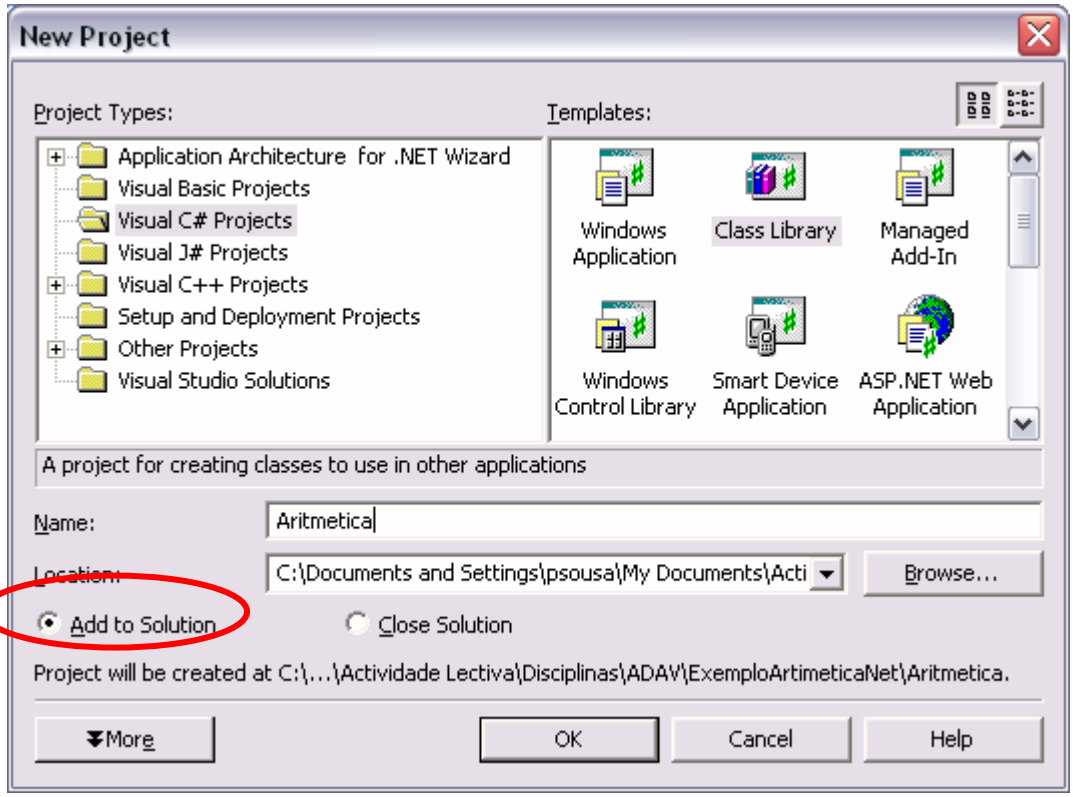

Figura 15 – criar um projecto tipo Class Library em C# e adicionar a uma solução existente

O Visual Studio vai criar o projecto e criar uma série de ficheiros pré-definidos. No “Solution Explorer” remover o ficheiro class1.cs.

Em seguida iremos especificar o contrato (a interface) do nosso componente, ou seja, que operações e que parâmetros recebe cada operação estão disponíveis para a execução de um serviço. No nosso exemplo o serviço que pretendemos fornecer é o de cálculos aritméticos básicos. Para tal, vamos adicionar uma interface ao projecto; clicando com o botão do lado direito do rato em cima do projecto, escolher a opção Add → Add New Item e criar uma classe chamada `IAritmetica`.

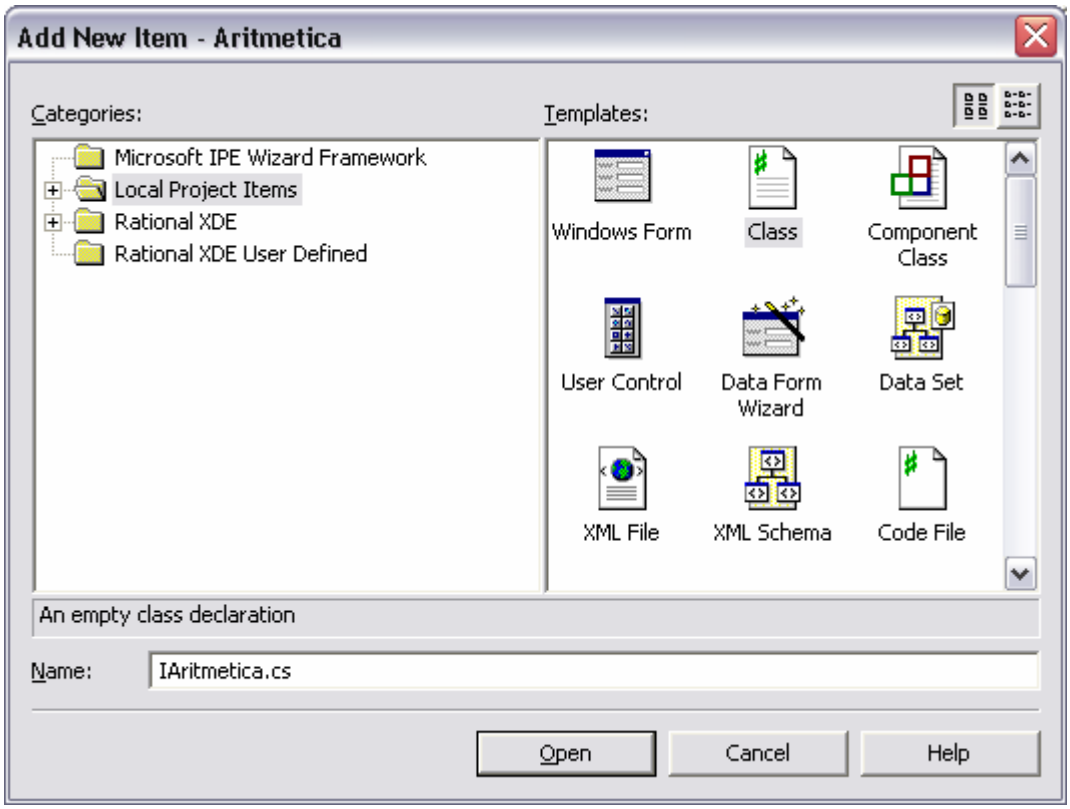

Figura 16 - adicionar uma classe a um projecto

Em seguida substituir a palavra `class` por `interface` e retirar o construtor gerado por omissão.

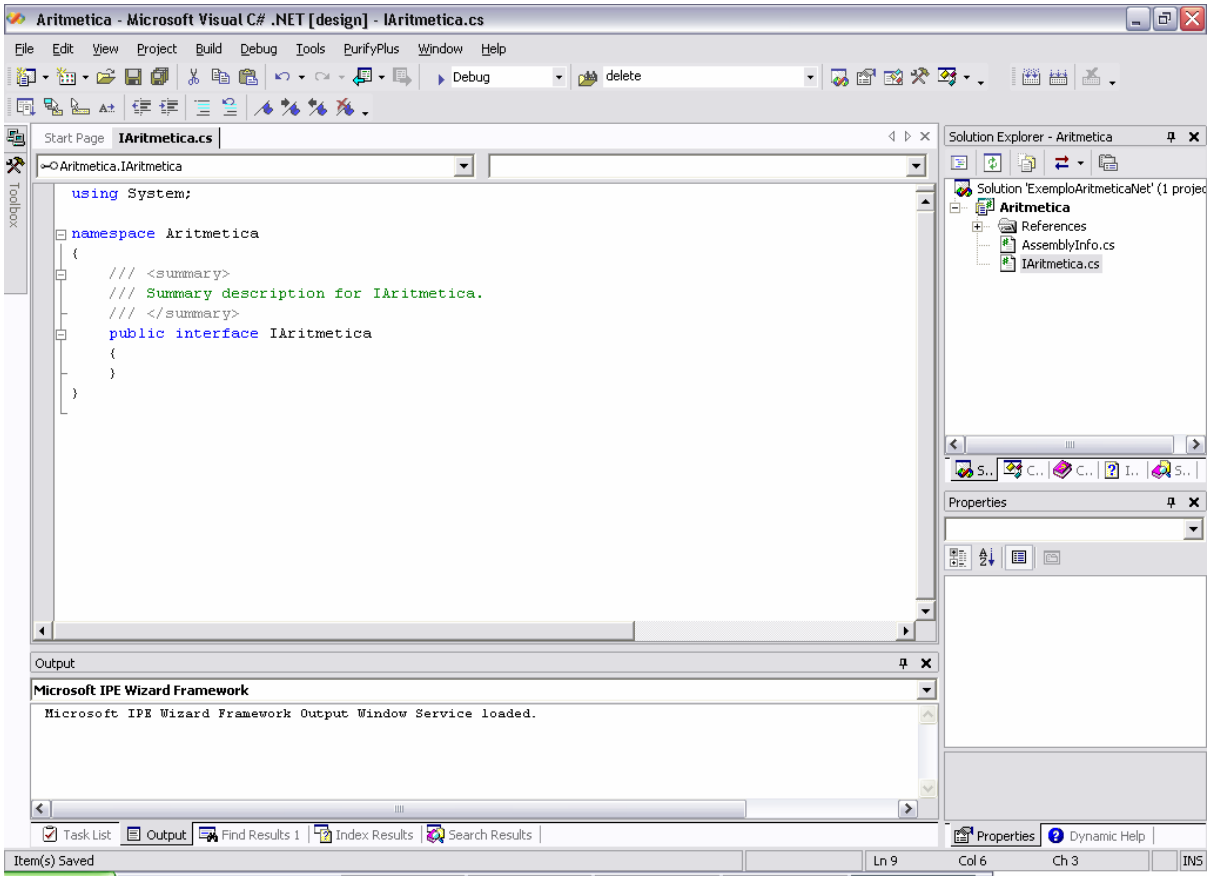

Figura 17 – esqueleto da interface

Mudando de vista no solution explorer para se ver as classes existentes no projecto (class explorer – ver Figura 18), adicionar à interface um método usando o botão do lado direito do rato (Figura 19).

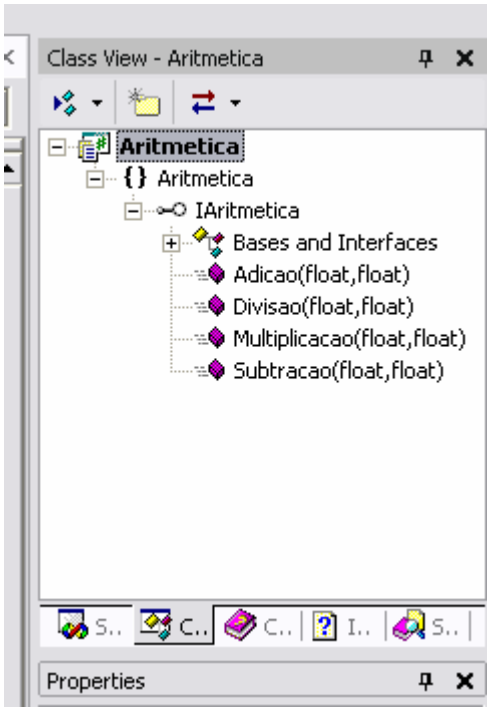

Figura 18 – vista de estrutura de classes no class explorer

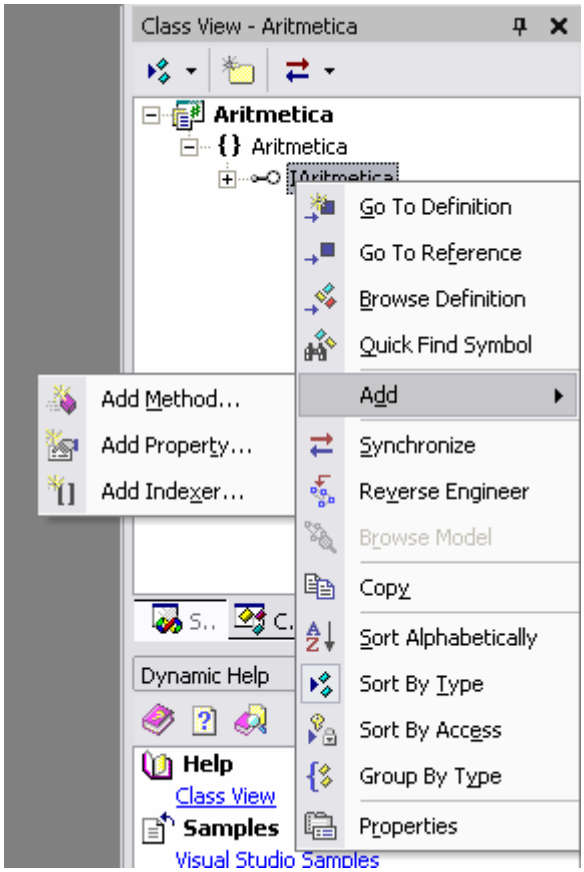

Figura 19 – adicionar um método a uma interface via wizard (passo 1)

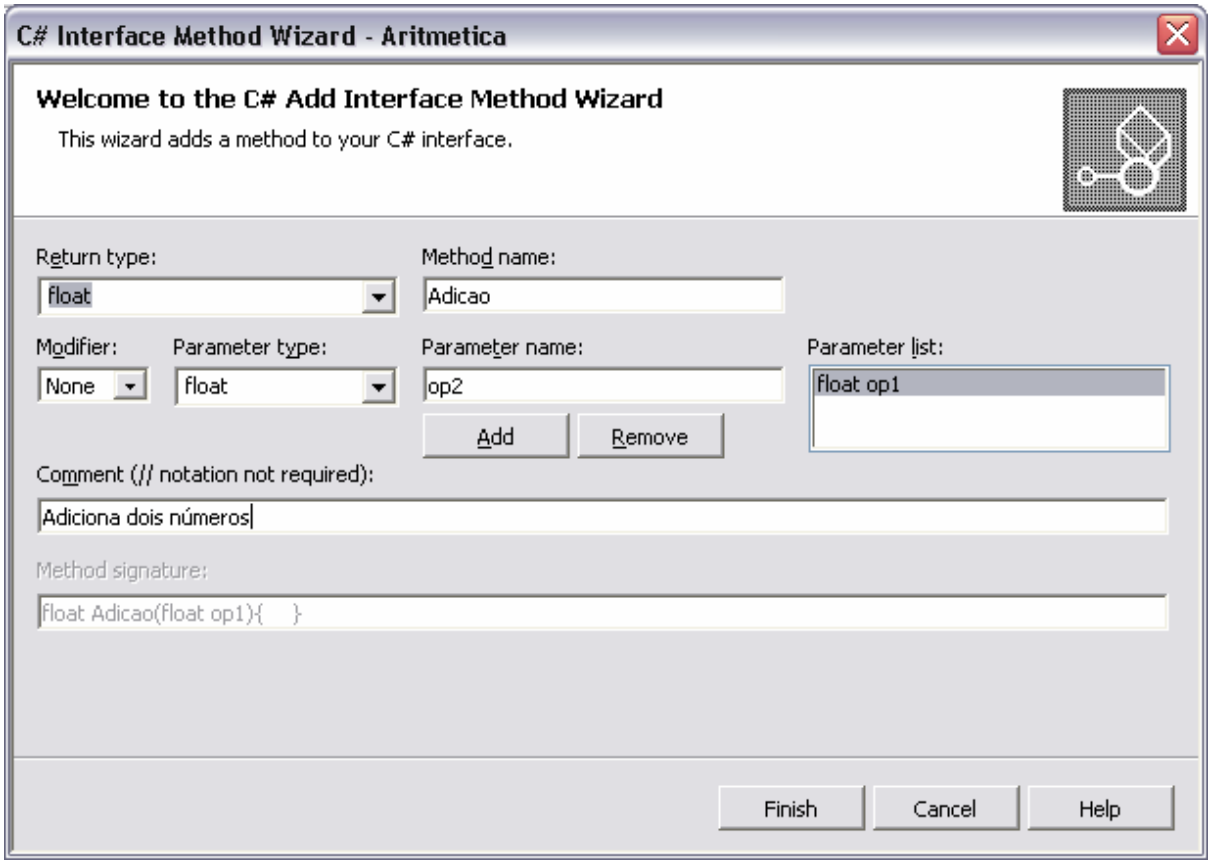

Figura 20 - adicionar um método a uma interface via wizard (passo 2)

O *wizard* vai automaticamente criar o método no ficheiro fonte.

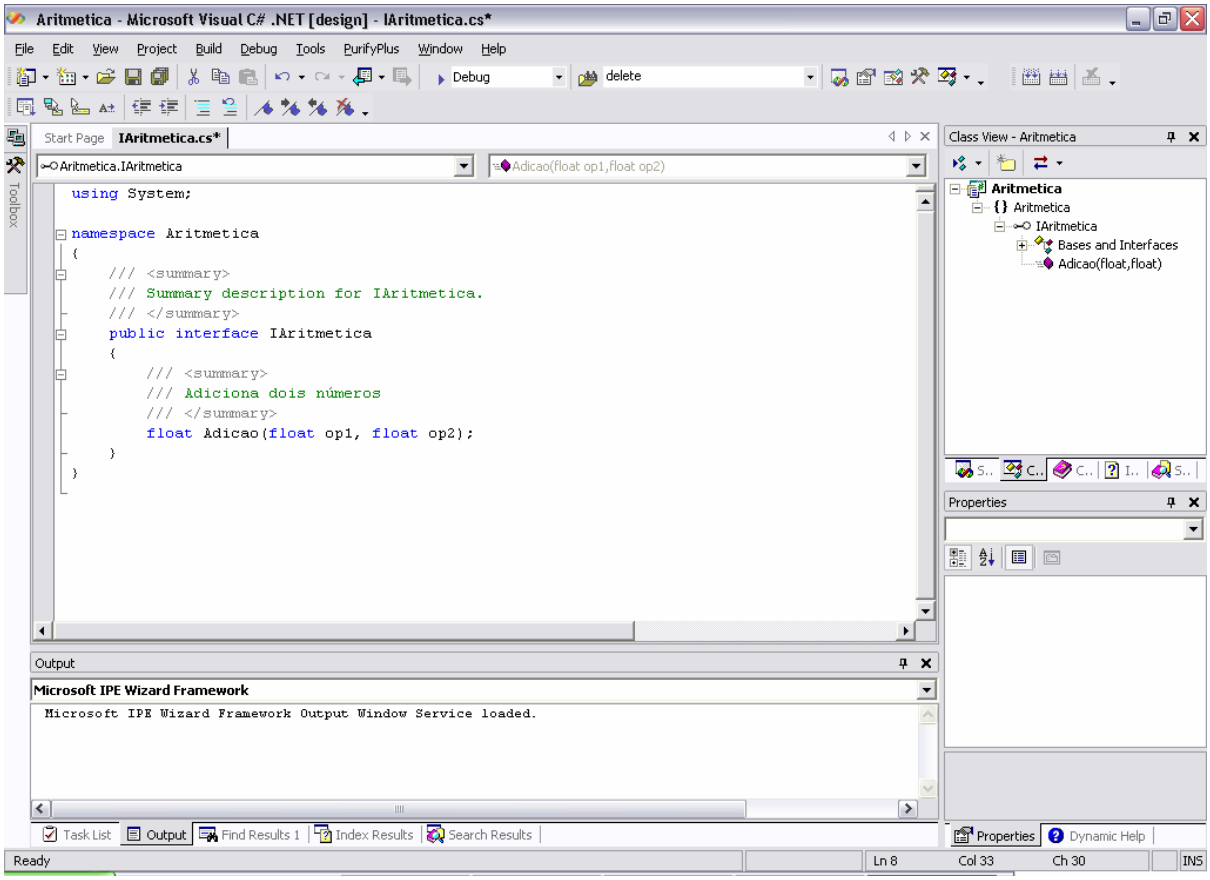

Figura 21 – código gerado automaticamente pelo wizard de adição de métodos

Da mesma forma adicionar os restantes métodos para a subtracção, divisão e multiplicação.

É também possível escrever os métodos directamente no editor de texto sem usar o wizard.

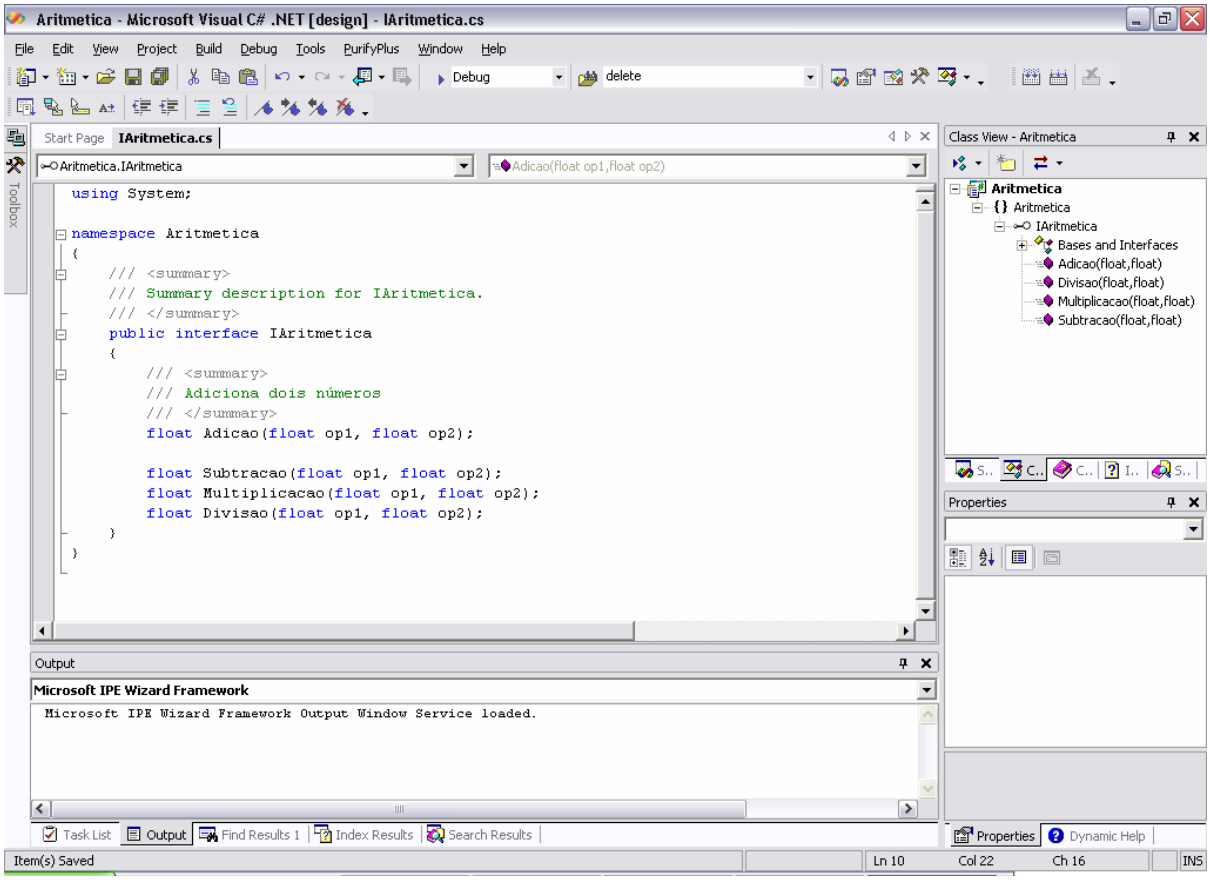

Figura 22 – interface para o serviço de aritmética

Em seguida vamos criar o prestador de serviços propriamente dito, ou seja, a implementação dos métodos definidos na interface anterior. Para tal, vamos adicionar uma classe ao projecto, seleccionando no *class explorer* o projecto "Aritmetica" e com o botão do lado direito do rato em cima do projecto, escolher a opção Add → Add Class.

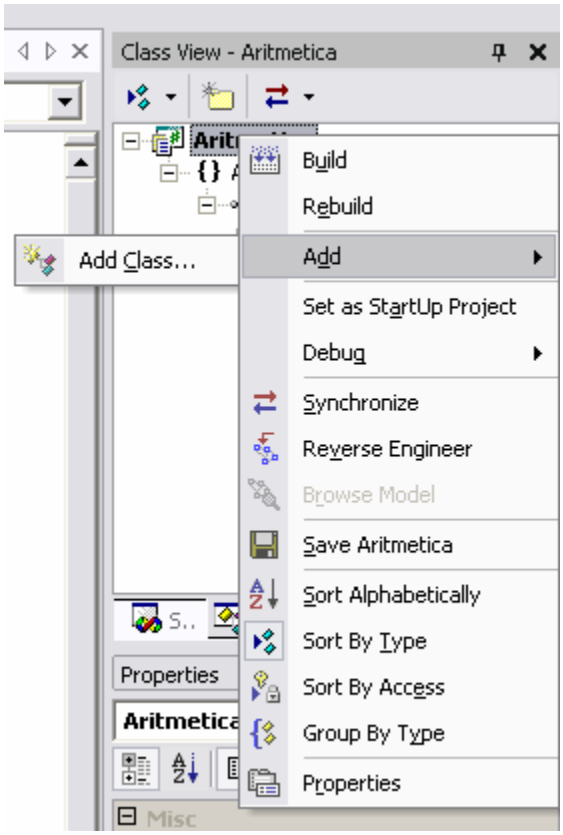

Figura 23 - adicionar uma classe via wizard a partir do class explorer (passo 1)

Irá aparecer o *wizard* de criação de classes (Figura 24) no qual indicaremos o nome da classe a criar e uma pequena descrição.

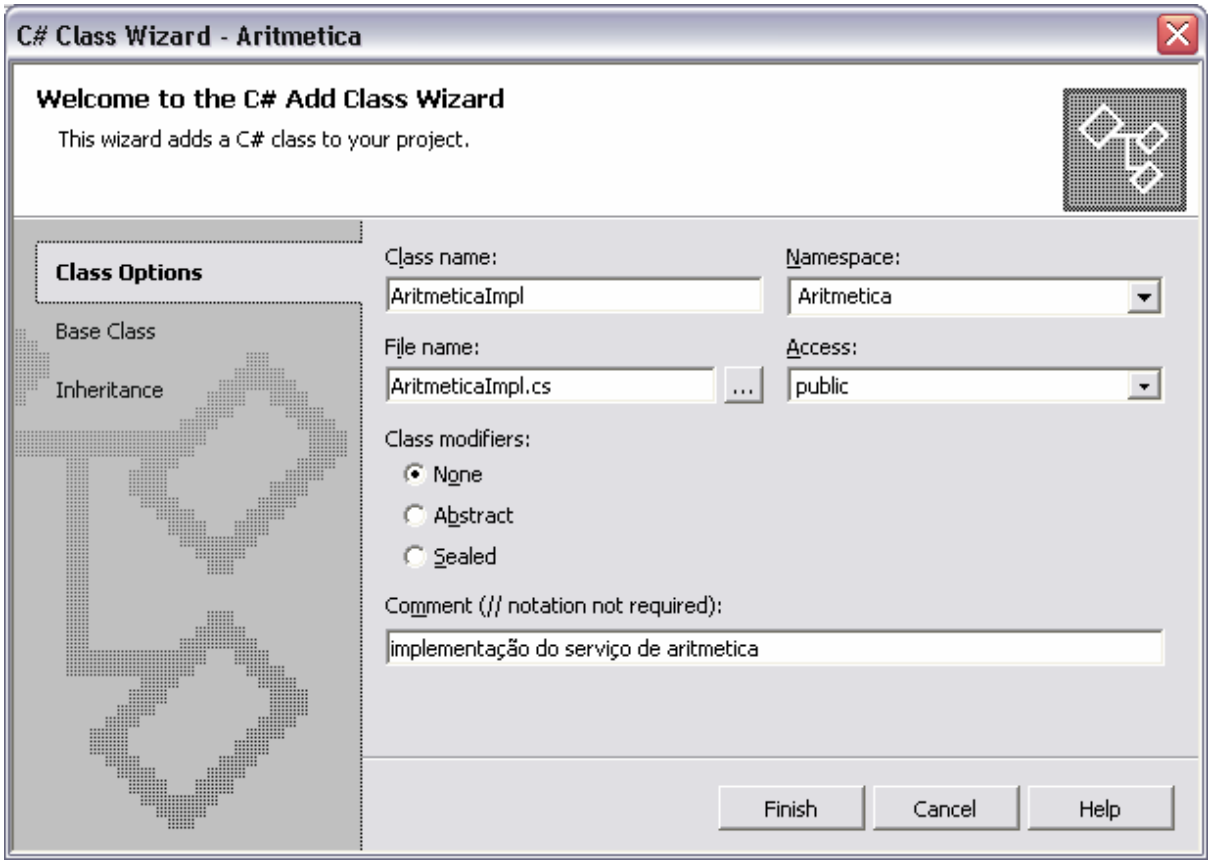

Figura 24 - adicionar uma classe via wizard a partir do class explorer (passo 2)

Em seguida iremos indicar características de herança desta classe seleccionando o separador “Inheritance” do lado esquerdo da janela do *wizard* (Figura 25).

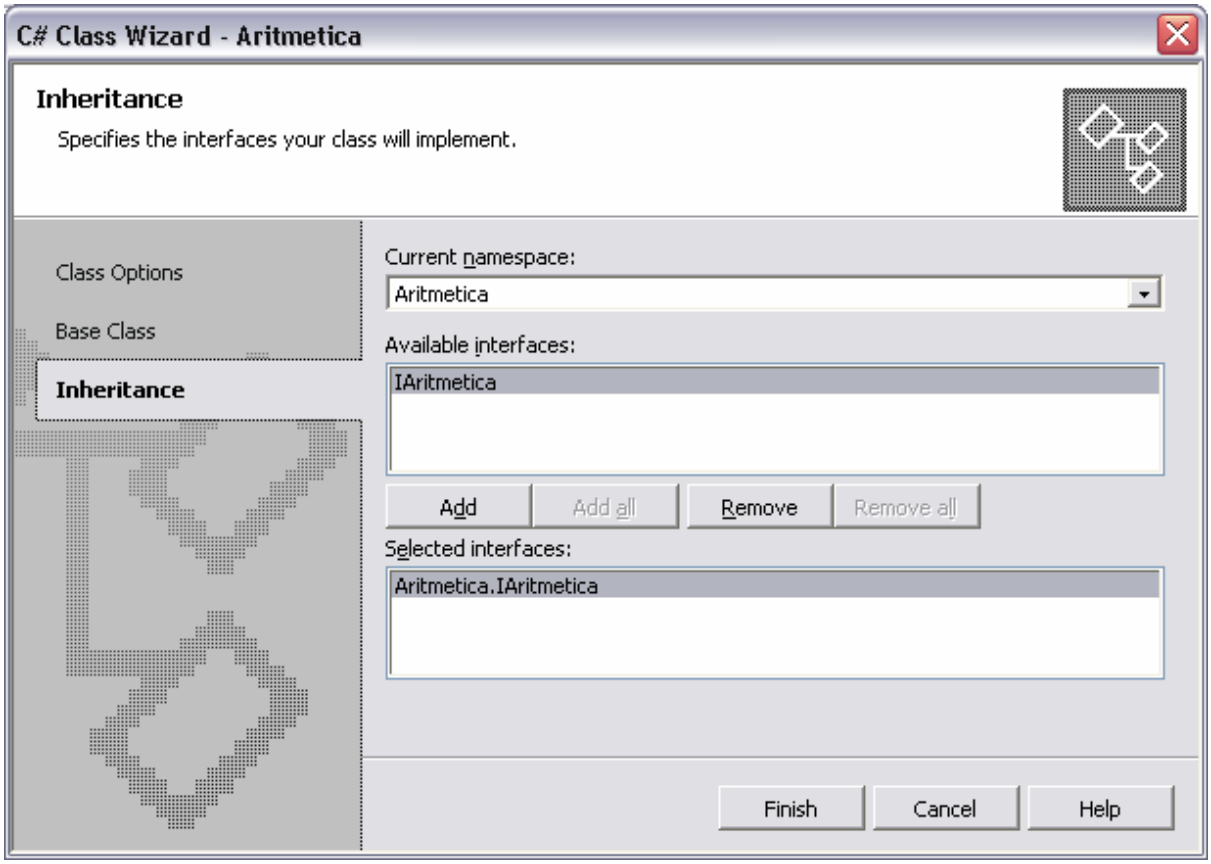

Figura 25- adicionar uma classe via wizard a partir do class explorer (passo 3)

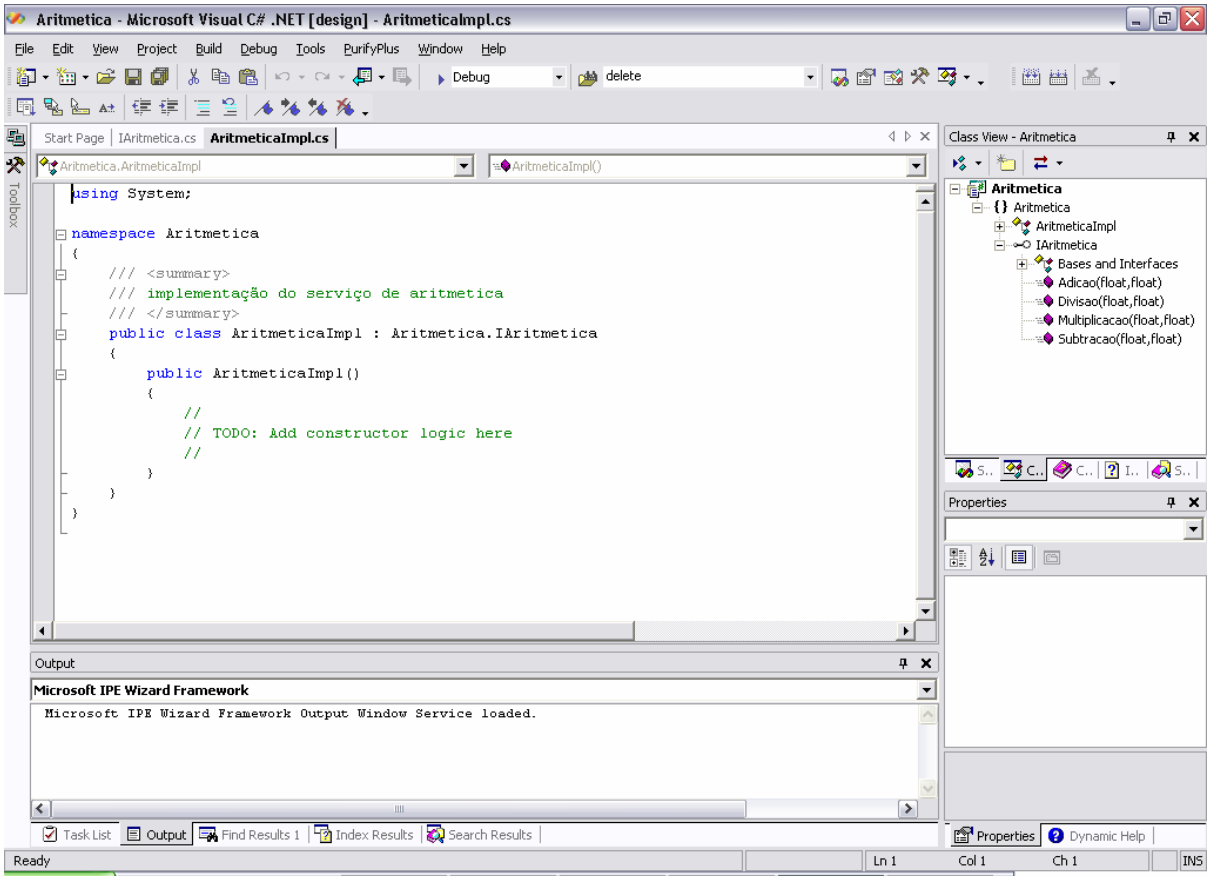

Figura 26 – classe gerada para implementação do serviço

Vamos em seguida implementar os métodos definidos na interface. Para isso no *class explorer* escolher das classes bases e interfaces da classe `AritmeticaImpl`, o item com a interface `IAritmetica` e com o botão do lado direito escolher Add → Implement Interface (Figura 27).

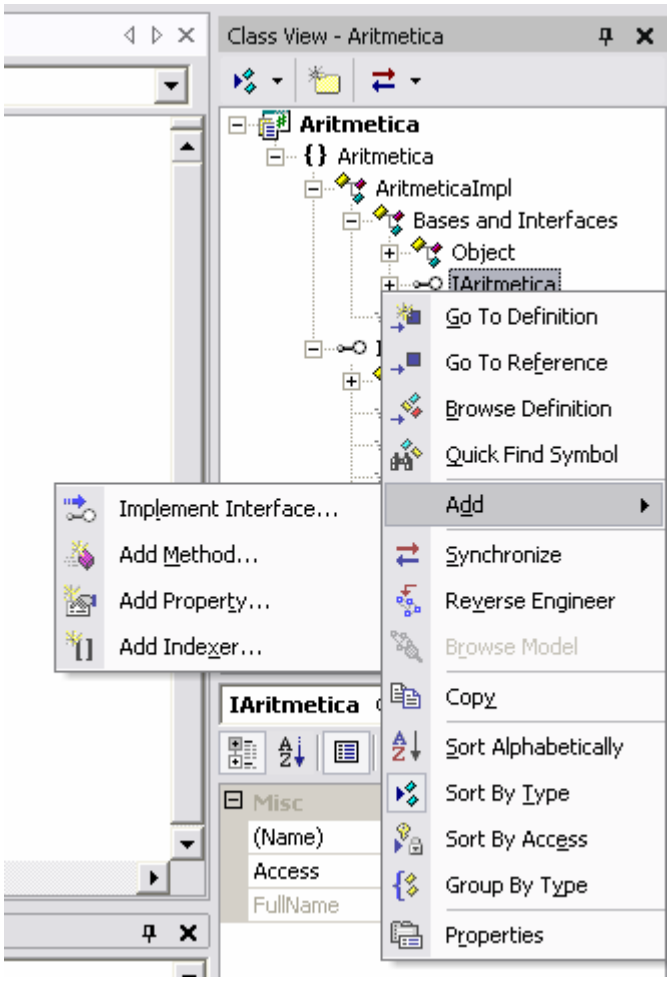

Figura 27 – implementar um interface

O Visual studio gerou o esqueleto de código associado à interface `IAritmetica` e alterou o ficheiro fonte da classe `ArtimeticaImpl` conforme se pode ver na figura seguinte.

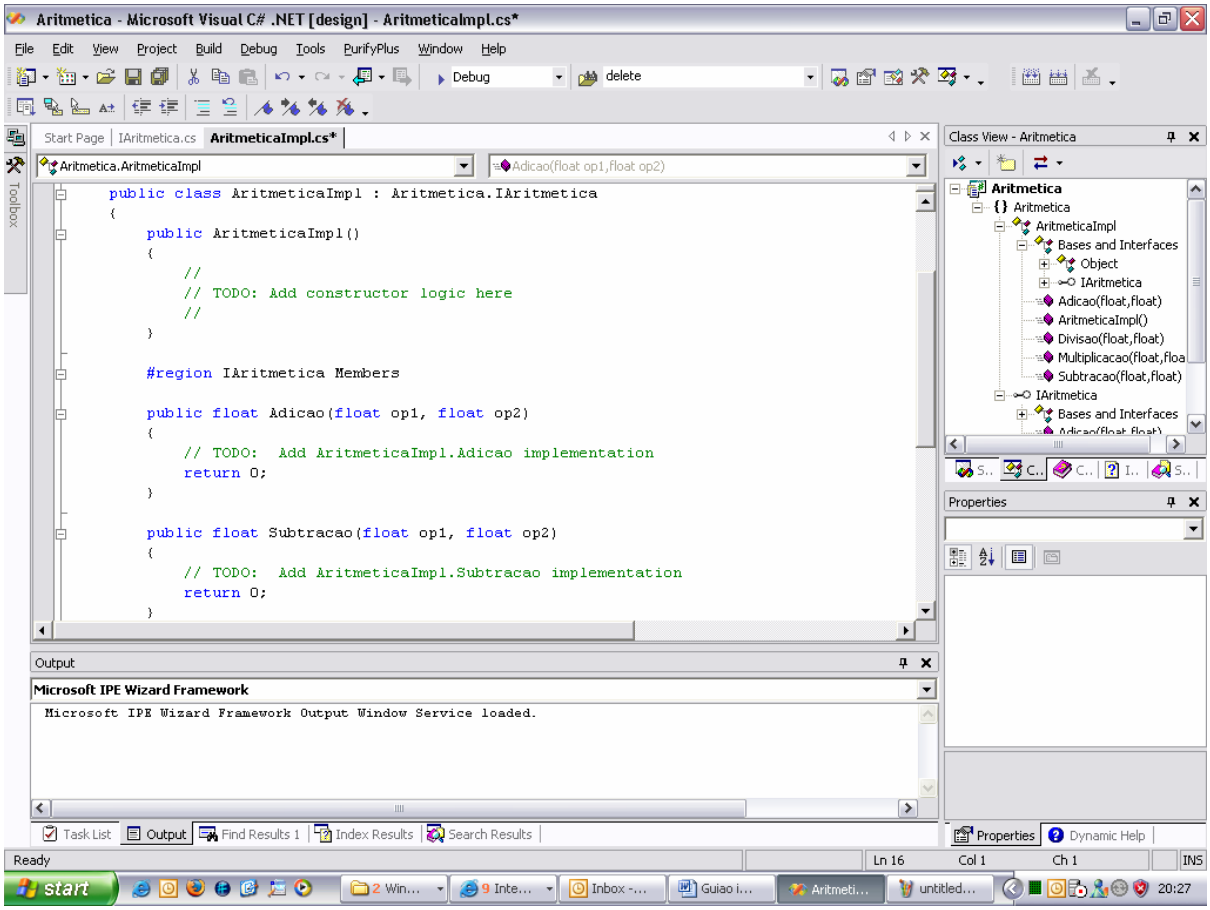

Figura 28 – classe de implementação do serviço com esqueleto dos métodos da interface

Colocar o seguinte código na implementação do método:

```
/// <summary>
/// adiciona dois números
/// </summary>
public float Adicao(float op1, float op2)
{
      return op1+op2;
}

/// <summary>
/// Subtrai dois números
/// </summary>
public float Subtracao(float op1, float op2)
{
      return op1-op2;
}

/// <summary>
/// Multiplica dois números
/// </summary>
public float Multiplicacao(float op1, float op2)
{
      return op1*op2;
}

/// <summary>
```

```
            /// Divide dois números
            /// </summary>
            public float Divisao(float op1, float op2)
            {
                  if (op2 != 0)
                        return op1/op2;
                  else
                        throw new ApplicationException("Tentativa de
efectuar uma Divisão por zero ");
            }
```

De notar que na implementação do método `Divisao()` recorremos a excepções para indicar ao utilizador desta classe situações anómalas. Neste caso utilizamos a classe predefinida `ApplicationException`, embora fosse possível definir as nossas próprias excepções[8].

Compilar (não deve dar erros).

5.4 Consola de teste

Vamos agora criar um novo projecto para testar o componente anterior. Usar File → New e escolher um novo projecto garantindo que se adiciona esse projecto à solução actual.

[8] Para mais informações sobre criação, lançamento (*throw*) e tratamento (*try-catch*) de excepções ver http://msdn.microsoft.com/library/en-us/cpguide/html/cpconhandlingthrowingexceptions.asp?frame=true

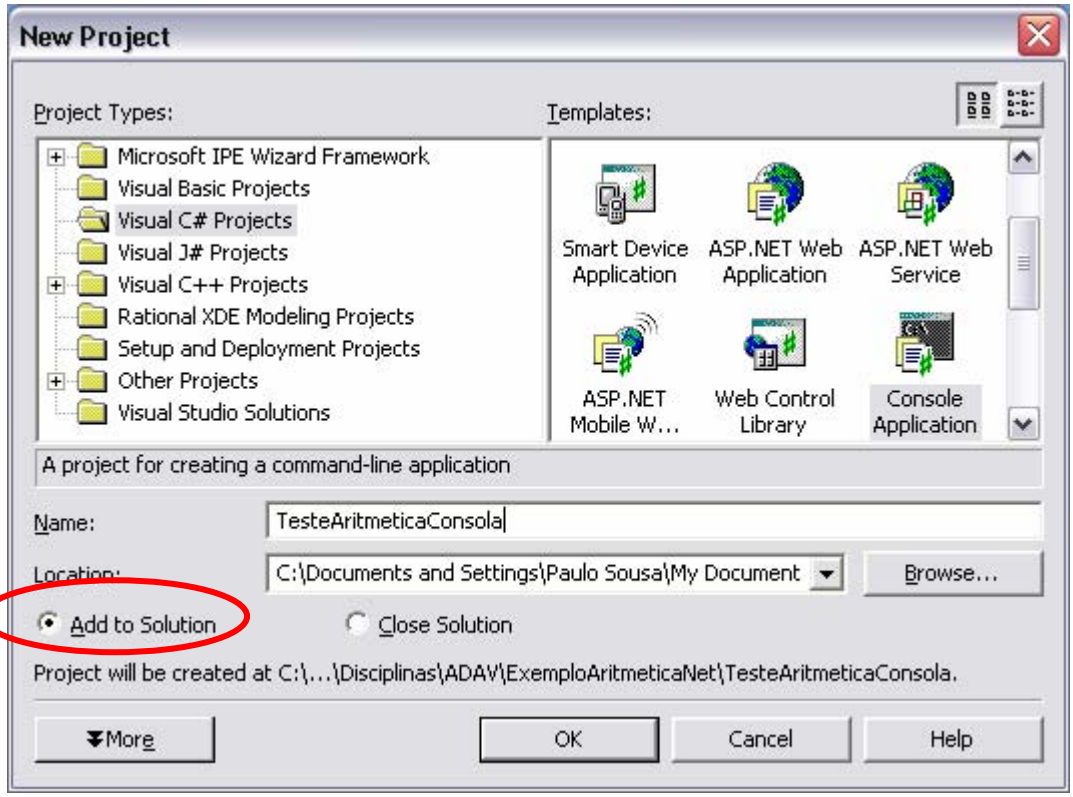

Figura 29 – criar um novo projecto de consola em C#

Para que possamos usar o componente anterior temos que adicionar uma referência ao componente. Para isso usamos o botão do lado direito do rato sobre o item “references” no *solution explorer*.

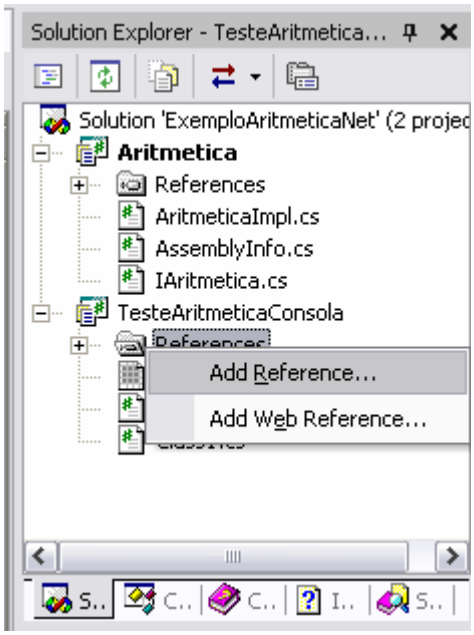

Figura 30 - Adicionar uma referência a outro projecto (passo 1)

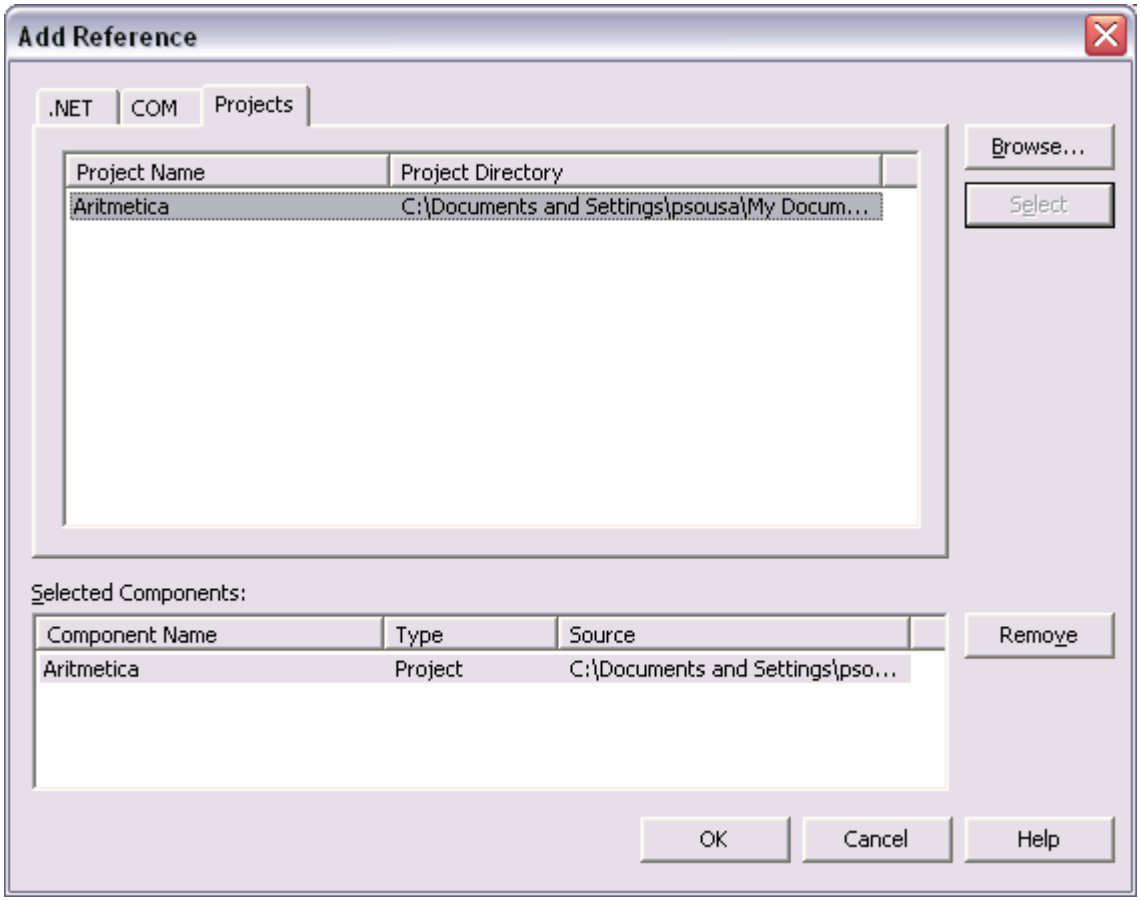

Figura 31 – Adicionar uma referência a outro projecto (passo 2)

No ficheiro class1.cs criado automaticamente, procurar o método `Main()` e colocar o seguinte código:

```
static void Main(string[] args)
{
      int op1 = 5;
      int op2 = 7;

      //declarar objecto para o serviço que pretendemos - interface
      Aritmetica.IAritmetica comp;

      //criar objecto com implementação do serviço
      comp = new Aritmetica.AritmeticaImpl();

      //invocar o serviço pretendido
      Console.WriteLine("Adição({0}, {1}) => {2}",
                        op1, op2, comp.Adicao(op1, op2));
}
```

O método `WriteLine()` permite a indicação de parâmetros indicados pelos números entre chavetas, para uma string de formatação. Para mais informação sobre o método `WriteLine()` consultar http://msdn.microsoft.com//library/en-us/cpref/html/frlrfsystemconsoleclasswritelinetopic15.asp; para informação sobre as strings

de formatação consultar http://msdn.microsoft.com/library/en-us/cpguide/html/cpconcompositeformatting.asp.

Para testar é necessário indicar ao Visual Studio qual o projecto a executar quando constam vários projectos na mesma solução. Para tal, com o botão do lado direito em cima do projecto no *solution explorer*, escolher “Set as startup project” (Figura 32).

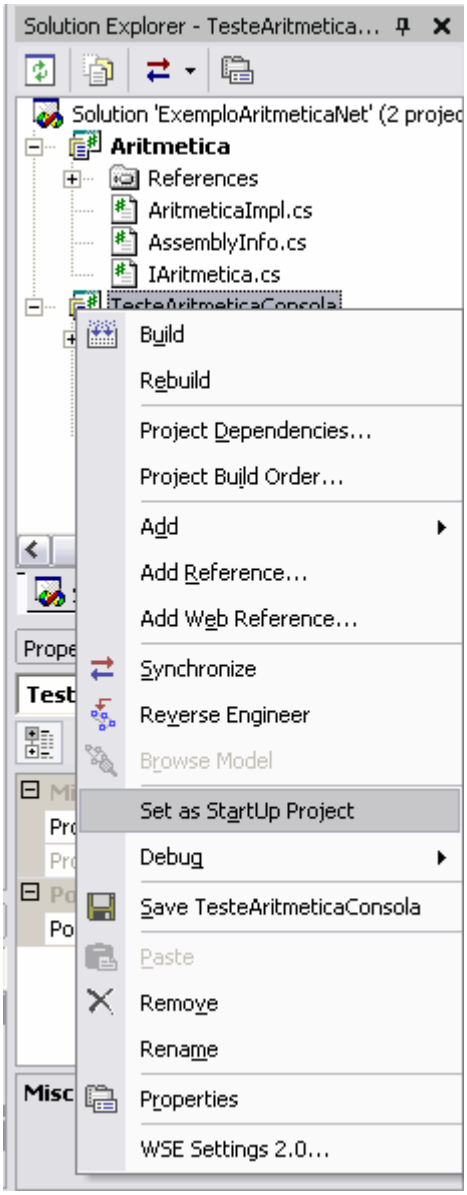

Figura 32 - seleccionar um projecto como projecto inicial para execução

Compilar e testar.

Figura 33 - aspecto geral da aplicação de consola

5.5 Aplicação Winforms de teste

Vamos agora criar outro tipo de cliente, uma aplicação windows neste caso em Visual Basic.net. usando File → New adicionar um novo projecto à solução.

Este projecto utiliza a linguagem de programação Visual Basic .net e não C# para demonstrar que é possível utilizar componentes escritos numa dada linguagem de programação a partir de clientes (programas ou outros componentes) escrita noutra linguagem de programação.

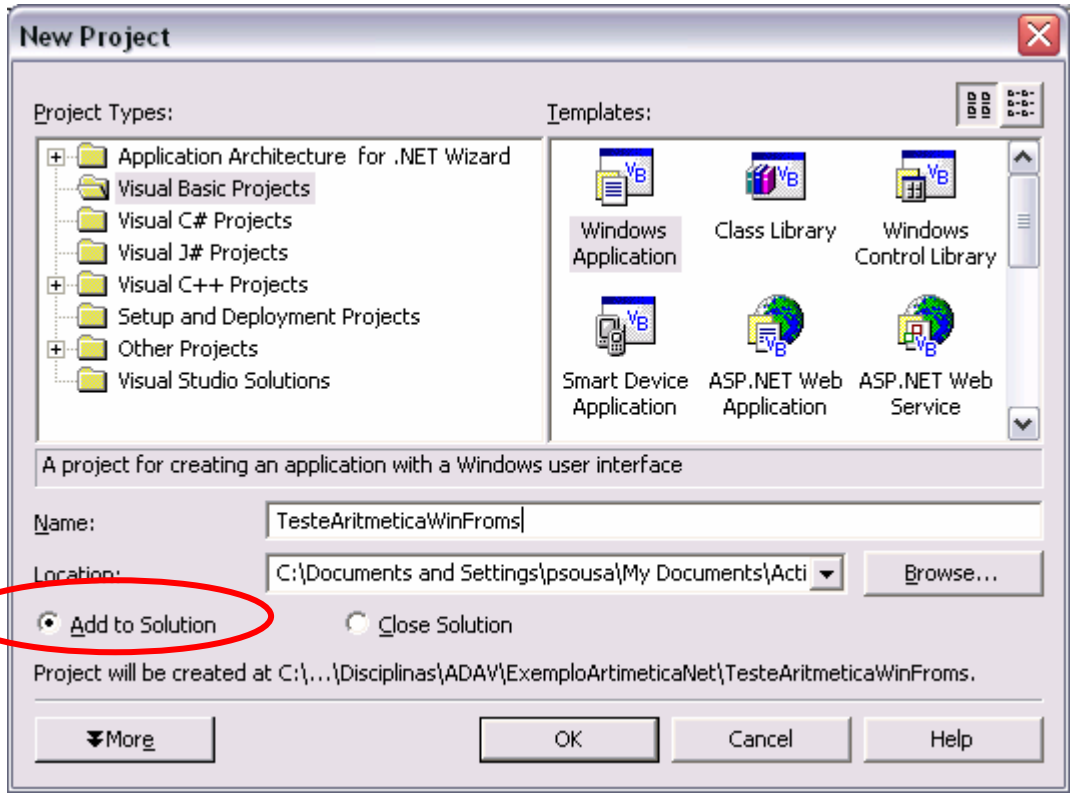

Figura 34 – criar um projecto WinForms em VB.net

Adicionar a referência ao componente como foi feito no projecto anterior.

Usando o editor visual construir o formulário da aplicação para que se pareça com o da imagem seguinte (utilize a janela “Properties” para atribuir os valores das propriedades dos controlos de acordo com a Tabela 4).

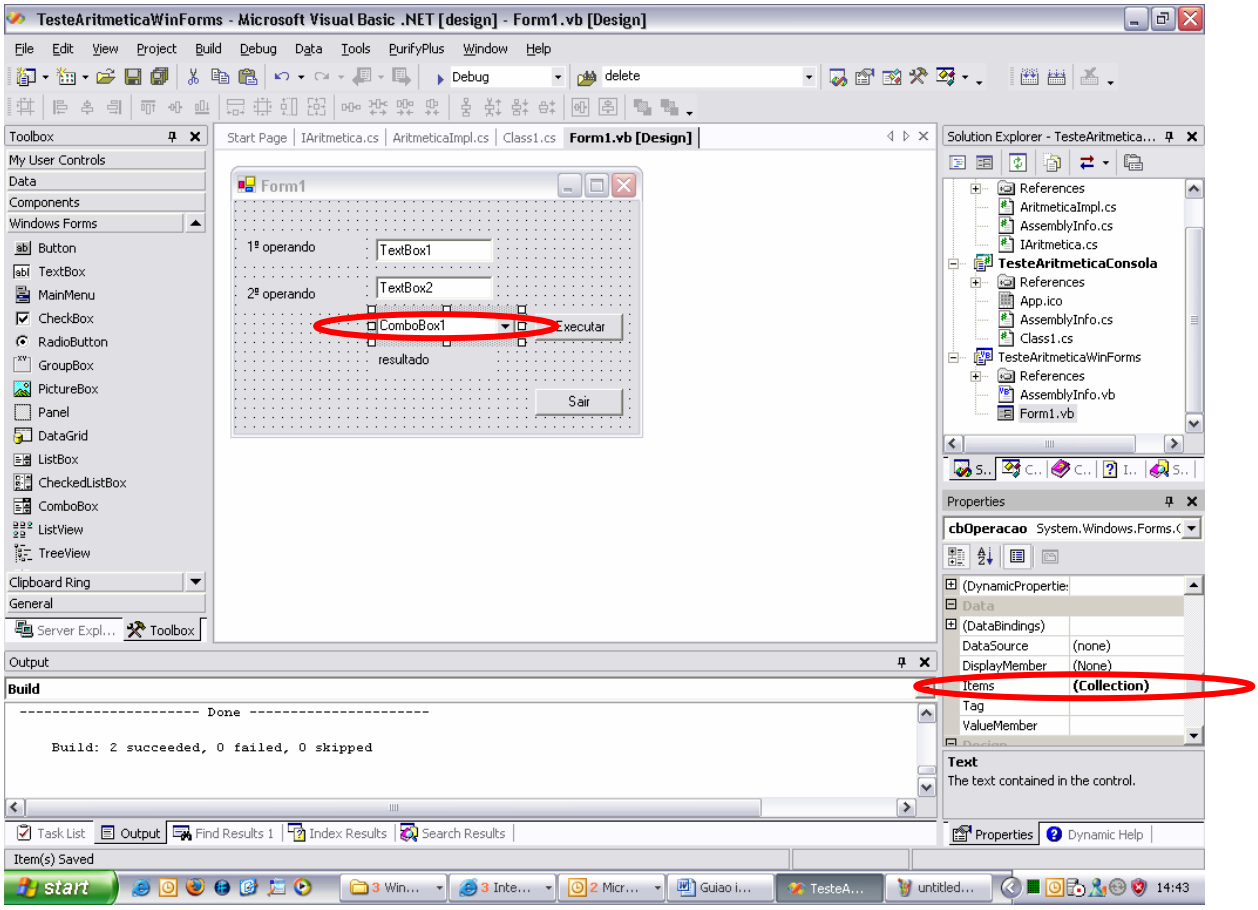

Figura 35 - adicionar itens alfanuméricos a uma ListBox ou ComboBox (passo 1)

Tabela 4 – propriedades dos controlos da aplicação WinForms

Elemento	Propriedade	Valor
1ª text box	Name	txtOp1
1ª text box	Text	
2ª text box	Name	txtOp2
2ª text box	Text	
combobox	Name	cbOperacao
1º botão	Name	btnExecutar
1º botão	Text	Executar
2º botão	Name	btnSair
2º botão	Text	Sair

1º label	`Text`	1º operando
2º label	`Text`	2º operando
3ª label	`Name`	lblResultado
3º label	`Text`	Resultado

A `ComboBox` pode ser preenchida com valores em tempo de desenho usando a propriedade `Collection`.

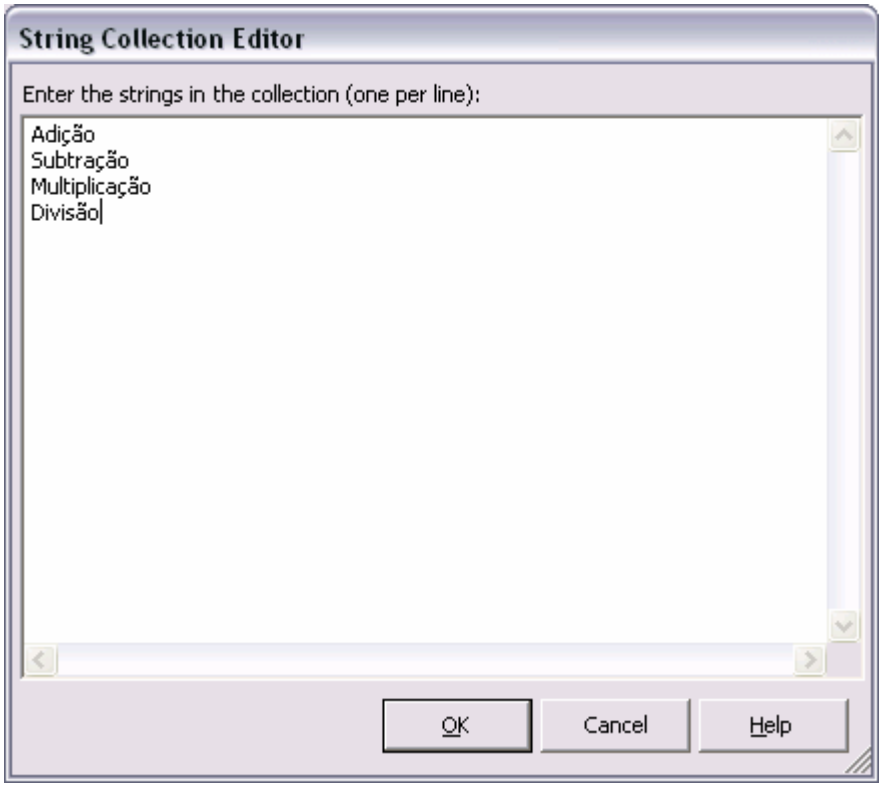

Figura 36 – adicionar itens alfanuméricos a uma ListBox ou ComboBox (passo 2)

Para testar a adição, criar um método de evento[9] associado ao botão de executar. Para isso efectuar um duplo *click* no botão e no método gerado colocar o seguinte código:

```
Private Sub btnExecutar_Click(ByVal sender As System.Object,
                              ByVal e As System.EventArgs)
                              Handles btnExecutar.Click
```

[9] Para mais informações sobre tratamento de eventos em aplicações que utilizam Win Forms consultar http://msdn.microsoft.com/library/en-us/vbcon/html/vbconeventhandling.asp; para informação geral sobre o mecanismo de eventos na plataforma .net consultar http://msdn.microsoft.com/library/en-us/vbcon/html/vborieventhandlers.asp.

```
        Dim oper1 As Single
        Dim oper2 As Single
        'declarar variavel
        Dim comp as Aritmetica.IAritmetica

        oper1 = Single.Parse(txtOp1.Text)
        oper2 = Single.Parse(txtOp2.Text)

        'criar objecto com implementação do serviço
        comp = New Aritmetica.AritmeticaImpl

        lblResultado.Text = comp.Adicao(oper1, oper2)
    End Sub
```

Vamos também inicializar o formulário criando o método que trata o evento de carregamento do formulário (indicado pela palavra reservada `Handles` e pelo evento `Load`), fazendo um duplo *click* em qualquer área livre o mesmo e colocando o seguinte código na função gerada.

```
    Private Sub Form1_Load(ByVal sender As System.Object,
                           ByVal e As System.EventArgs)
                           Handles MyBase.Load
        ´colocar como seleccionado a 1ª opção da listbox
        cbOperacao.SelectedIndex = 0
    End Sub
```

Finalmente vamos colocar código associado ao botão de saída da aplicação para fechar a janela e sair da aplicação.

```
    Private Sub btnSair_Click(ByVal sender As System.Object,
                           ByVal e As System.EventArgs)
                           Handles btnSair.Click
        Me.Close()
    End Sub
```

Seleccionar o projecto "TesteAritmeticaWinForms" como projecto *start up*, compilar e testar.

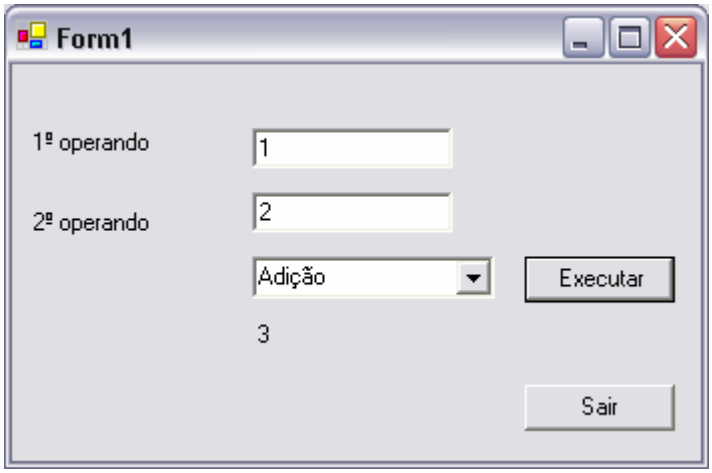

Figura 37 – aspecto geral da aplicação windows

Vamos agora colocar em funcionamento as restantes opções:

```
Private Sub btnExecutar_Click(ByVal sender As System.Object,
                              ByVal e As System.EventArgs)
                              Handles btnExecutar.Click
      Dim oper1 As Single
      Dim oper2 As Single
      'declarar variavel
      Dim comp as Aritmetica.IAritmetica

      oper1 = Single.Parse(txtOp1.Text)
      oper2 = Single.Parse(txtOp2.Text)

      'criar objecto com implementação do serviço
      comp = New Aritmetica.AritmeticaImpl

      If cbOperacao.SelectedItem = "Adição" Then
            lblResultado.Text = comp.Adicao(oper1, oper2)
      End If
      If cbOperacao.SelectedItem = "Subtracção" Then
            lblResultado.Text = comp.Subtracao(oper1, oper2)
      End If
      If cbOperacao.SelectedItem = "Multiplicação" Then
            lblResultado.Text = comp.Multiplicacao(oper1, oper2)
      End If
      If cbOperacao.SelectedItem = "Divisão" Then
            lblResultado.Text = comp.Divisao(oper1, oper2)
      End If
End Sub
```

Compilar e testar.

Como se pode ver, no caso da divisão, se tentarmos dividir por zero o programa gera uma excepção. Podemos programaticamente controlar essa excepção fazendo a seguinte alteração:

```
If cbOperacao.SelectedItem = "Divisão" Then
      Try
            lblResultado.Text = comp.Divisao(oper1, oper2)
      Catch ex As ApplicationException
            System.Windows.Forms.MessageBox.Show("Não pode dividir
      por Zero.")
      End Try
End If
```

Compilar e testar.

Adicionalmente, pode também ser gerada uma excepção caso se introduzam caracteres não numéricos nas caixas de texto (ou mesmo se a caixa de texto estiver vazia) quando se tenta fazer o `Single.Parse()`. Para corrigir esse problema devemos alterar o código do método para o seguinte:

```
Private Sub btnExecutar_Click(ByVal sender As System.Object,
```

```
                                ByVal e As System.EventArgs)
                                Handles btnExecutar.Click
        Dim oper1 As Single
        Dim oper2 As Single
        'declarar variavel
        Dim comp As Aritmetica.IAritmetica

        'criar objecto com implementação do serviço
        comp = New Aritmetica.AritmeticaImpl

        Try
            oper1 = Single.Parse(txtOp1.Text)
            oper2 = Single.Parse(txtOp2.Text)

            If cbOperacao.SelectedItem = "Adição" Then
                lblResultado.Text = comp.Adicao(oper1, oper2)
            End If
            If cbOperacao.SelectedItem = "Subtracção" Then
                lblResultado.Text = comp.Subtracao(oper1, oper2)
            End If
            If cbOperacao.SelectedItem = "Multiplicação" Then
                lblResultado.Text = comp.Multiplicacao(oper1,
            oper2)
            End If
            If cbOperacao.SelectedItem = "Divisão" Then
                Try
                    lblResultado.Text = comp.Divisao(oper1,
                oper2)
                Catch ex As ApplicationException
                    System.Windows.Forms.MessageBox.Show("Não
                pode dividir por Zero.")
                End Try
            End If
        Catch ex As FormatException
            System.Windows.Forms.MessageBox.Show("Por favor preencha
as caixas de texto apenas com números")
        End Try
    End Sub
```

Compilar e testar.

5.6 Aplicação ASP.net de teste

Vamos agora utilizar o nosso componente a partir de uma aplicação web, para isso criar um novo projecto web em C# na solução que temos vindo a usar.

NOTA: *nos laboratórios do DEI não será possível criar este projecto pois os alunos não têm permissões de escrita num servidor web que suporte ASP.net*.

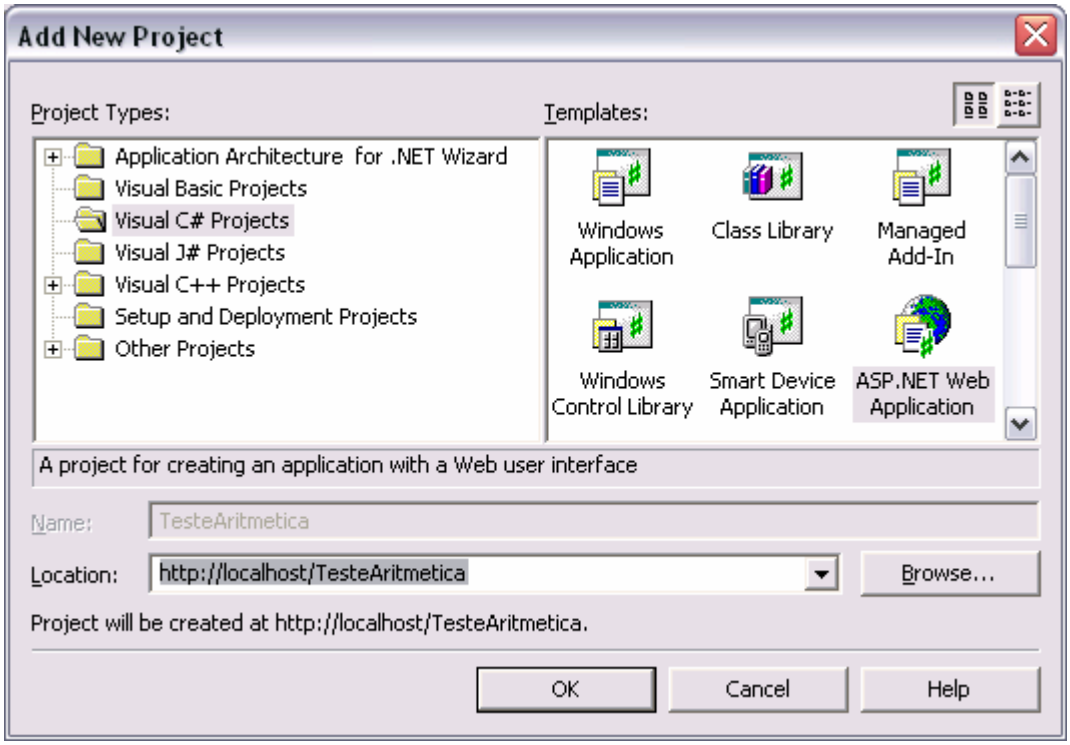

Figura 38 – criar um projecto tipo aplicação web asp.net em C#

E adicionar a referência ao componente tal como foi feito nos passos anteriores.

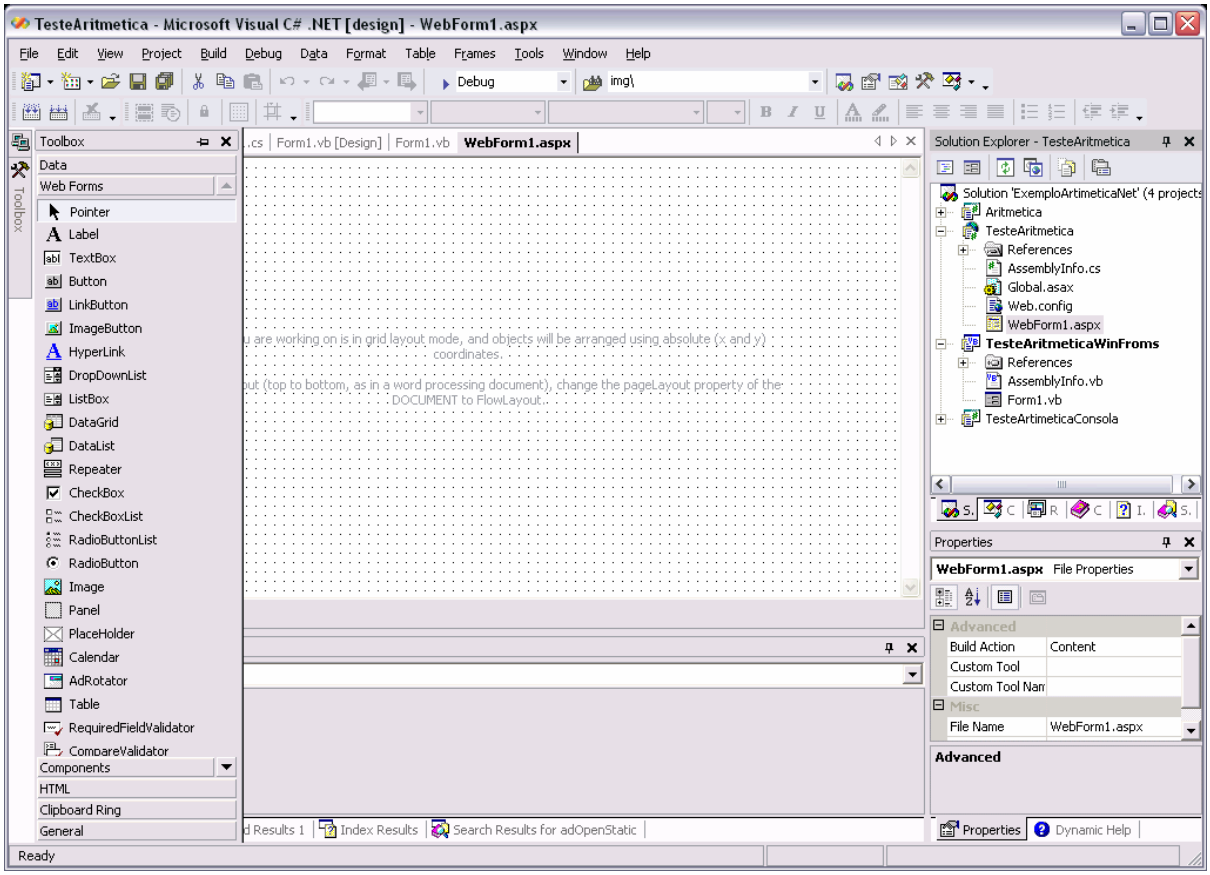

Figura 39 – toolbox de controlos disponíveis para formulários web

Usando a *toolbox*, desenhar o formulário que se pode ver na figura seguinte.

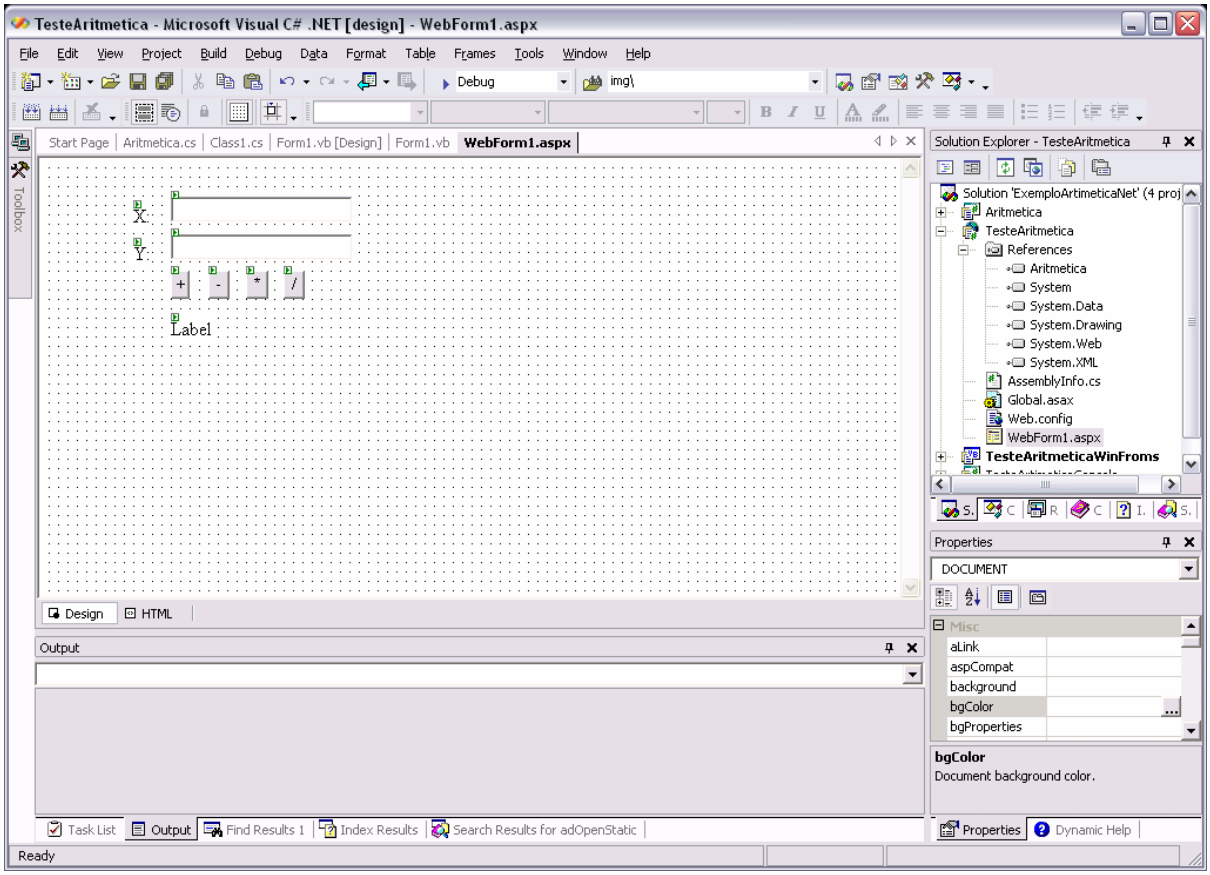

Figura 40 – formulário web a criar para testar componente

Para este exemplo vamos utilizar um botão para cada operação. Para se colocar o botão de adição a funcionar vamos associar um método ao evento de *click* do botão. Para isso fazemos duplo *click* sobre o botão e o Visual Studio automaticamente vai criar um método e associa-lo ao evento *click*. Nesse método colocar o seguinte código.

```
using Aritmetica; // simplifica a escrita do código


private void btnAdicao_Click(object sender, System.EventArgs e)
{
      IAritmetica comp = new AritmeticaImpl();

      try
      {
            lblResultado.Text = comp.Adicao(
                              float.Parse(txtOp1.Text),
                              float.Parse(txtOp2.Text)
                              ).ToString();
      }
      catch (FormatException ex)
      {
            lblResultado.Text = "";
      }
}
```

De reparar que agora em C# é necessário converter os valores da caixa de texto de strings para inteiros usando o método `float.Parse()` e em seguida converter o resultado inteiro para string para ser visualizado usando o método `ToString()`. Neste exemplo também tratamos um caso adicional em que o utilizador tenha colocado caracteres não numéricos nas caixas de texto tratando a excepção `FormatException` que é gerada pelo método `float.Parse()`. Adicionalmente, o tratamento de eventos em C# é diferente do Visual Basic não existindo em C# o equivalente à palavra reservada `Handles` do VB; para associar o evento ao *handler*, o visual studio adicionou a seguinte linha de código no método de inicialização da página web (`InitializeComponent`):

```
this.btnAdicao.Click +=
            new System.EventHandler(this.btnAdicao_Click);
```

Compilar e testar.

Para colocar os restantes botões a funcionar, basta criar os métodos para os eventos de click e colocar nesses métodos o seguinte código.

```
private void btnSubtracao_Click(object sender, System.EventArgs e)
{
      IAritmetica comp = new AritmeticaImpl();

      try
      {
            lblResultado.Text = comp.Subtracao(
                                    float.Parse(txtOp1.Text),
                                    float.Parse(txtOp2.Text)
                                    ).ToString();
      }
      catch (FormatException ex)
      {
            lblResultado.Text = "";
      }
}

private void btnMultiplicacao_Click(object sender, System.EventArgs
e)
{
      IAritmetica comp = new AritmeticaImpl();

      try
      {
            lblResultado.Text = comp.Multiplicacao(
                                    float.Parse(txtOp1.Text),
                                    float.Parse(txtOp2.Text)
                                    ).ToString();
      }
      catch (FormatException ex)
      {
```

```
                lblResultado.Text = "";
            }
        }

        private void btnDivisao_Click(object sender, System.EventArgs e)
        {
            IAritmetica comp = new AritmeticaImpl();

            try
            {
                lblResultado.Text = comp.Divisao(
                                        float.Parse(txtOp1.Text),
                                        float.Parse(txtOp2.Text)
                                        ).ToString();
            }
            catch (FormatException ex)
            {
                lblResultado.Text = "";
            }
            catch (ApplicationException ex)
            {
                lblResultado.Text = "Não é possivel dividir por zero";
            }
        }
```

Compilar e testar.

Como conclusão podemos dizer que criar componentes em .net é tão fácil como criar uma classe e coloca-la numa *Class Library* para que possa ser reutilizada em formato binário. Esse componente pode então ser utilizado de forma igual em vários tipos de clientes e em várias linguagens de programação. Para se conseguir reutilizar o componente apenas necessitamos de adicionar uma referência ao componente/projecto.

5.7 Melhorias propostas e questões

1 Criar um cliente em Winforms usando linguagem C# e com botões separados para cada operação

2 Criar um novo serviço no componente para concatenação de duas string e modificar os clientes anteriores para testar este novo método.

3 Pesquise o significado da palavra reservada `params` em C# e implemente o seguinte método:

```
public string Concatenar(params string[] args)
```

implemente num dos clientes uma função para teste deste novo método.

6 Guião de trabalho: Evolução do componente para operações aritméticas

6.1 introdução

Um aspecto importante do desenvolvimento orientado aos componentes é a evolução das interfaces dos mesmos. No caso do componente de aritmética desenvolvido na secção 5.3, vamos querer criar uma nova versão do componente com funcionalidades adicionais, por exemplo, a operação potência.

Um aspecto muito importante a ter em conta, é que uma vez uma interface publicada (ou seja, potencialmente utilizada por outras aplicações), essa interface torna-se **imutável**. Ou seja, **não se devem alterar as interfaces após a sua definição e publicação**. Caso se pretenda aumentar funcionalidades cria-se uma nova interface.

6.2 Passos preparatórios

NOTA: *este passo apenas é necessário para efeitos de demonstração*

Antes de começar as alterações vamos copiar o directório "bin" da aplicação de teste de consola para um directório temporário para mais tarde demonstrar que a evolução do componente não implicou alterações nem recompilação das aplicações clientes já existentes.

6.3 Alterações ao componente

Vamos abrir o projecto anterior e criar uma nova interface chamada `IAritmetica2` que deriva de `IAritmetica` de acordo com o seguinte código:

```
public interface IAritmetica2 : IAritmetica
{
      double Potencia(float b, int expoente);
      double RaizQuadrada(float n);
}
```

Vamos agora implementar esta nova interface. Para isso vamos abrir o ficheiro da classe `AritmeticaImpl`. No código vamos acrescentar a declaração da interface que pretendemos implementar (`IAritmetica2`). De reparar que o visual studio irá perguntar se pretendemos gerar automaticamente o esqueleto para os métodos definidos na interface, bastando para tal pressionar na tecla TAB (Figura 41).

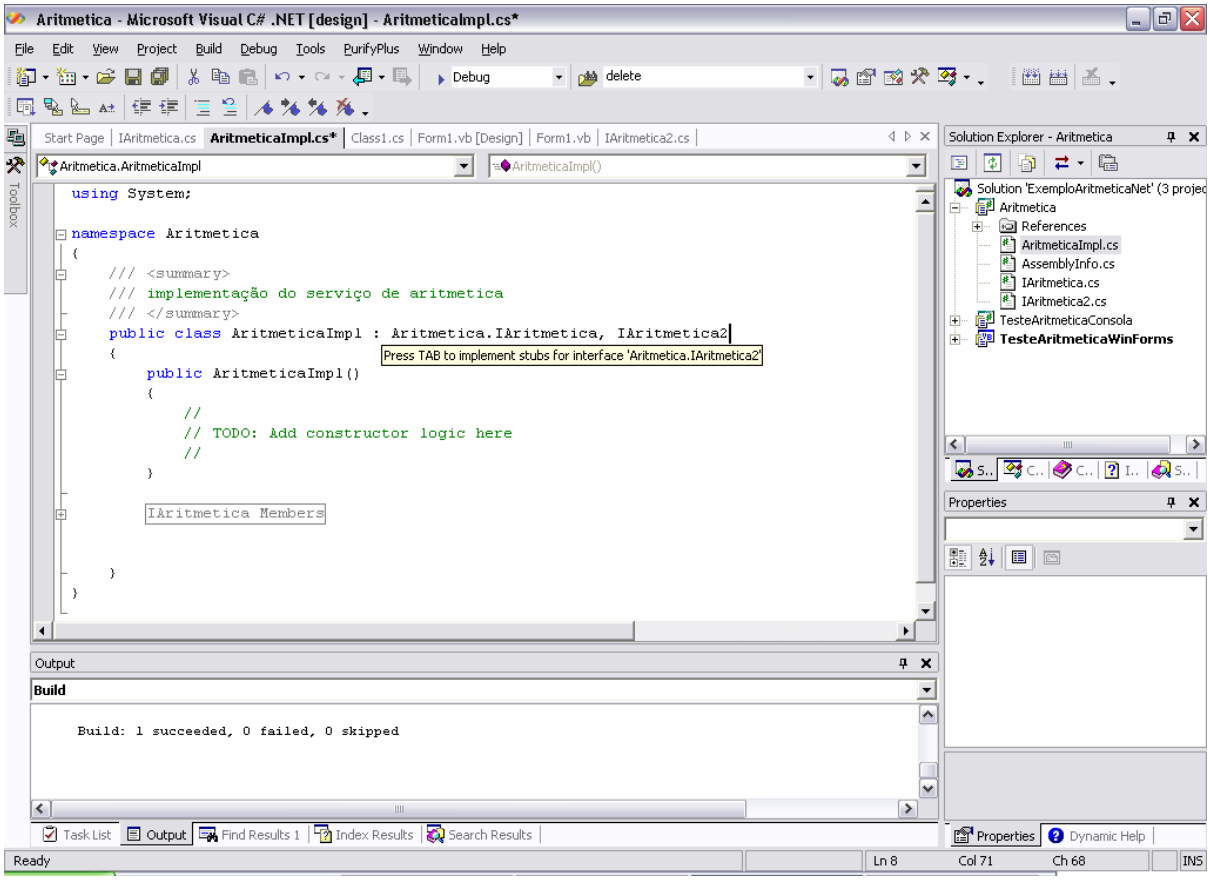

Figura 41 - geração automatica de interface

Para implementar os métodos iremos utilizar os métodos existentes na classe `Math` da plataforma .net.

```
public double Potencia(float b, int expoente)
{
      return Math.Pow(b, expoente);
}

public double RaizQuadrada(float n)
{
      return Math.Sqrt(n);
}
```

Compilar, não deve dar erros.

6.4 Aplicação de teste

Para experimentar esta nova versão do componente iremos criar uma nova aplicação de teste adicionando um novo projecto de consola em C#, denominado “TesteArti2” à solução.

Adicionar a referência ao componente.

Para testar colocar o seguinte código:

```
// criar instância do componente e requisitar nova interface
Aritmetica.IAritmetica2 comp = new Aritmetica.AritmeticaImpl();

int b = 2;
int e = 2;
float n = 25;

System.Console.WriteLine("{0}^{1} = {2}\n", b, e,
                         comp.Potencia(b, e));
System.Console.WriteLine("raiz quadrada de {0} = {1}\n", n,
                         comp.RaizQuadrada(n));
```

Definir este novo projecto como startup project.

Compilar (não deve dar erros) e testar.

6.5 Teste da aplicação cliente para versão anterior

Efectuar os seguintes passos:

1. abrir uma janela do explorer ou linha de comando no directório temporário para onde se copiou a aplicação de teste
2. executar o programa
3. verificar na janela do explorer a versão da DLL correspondente ao nosso componente (Figura 42)

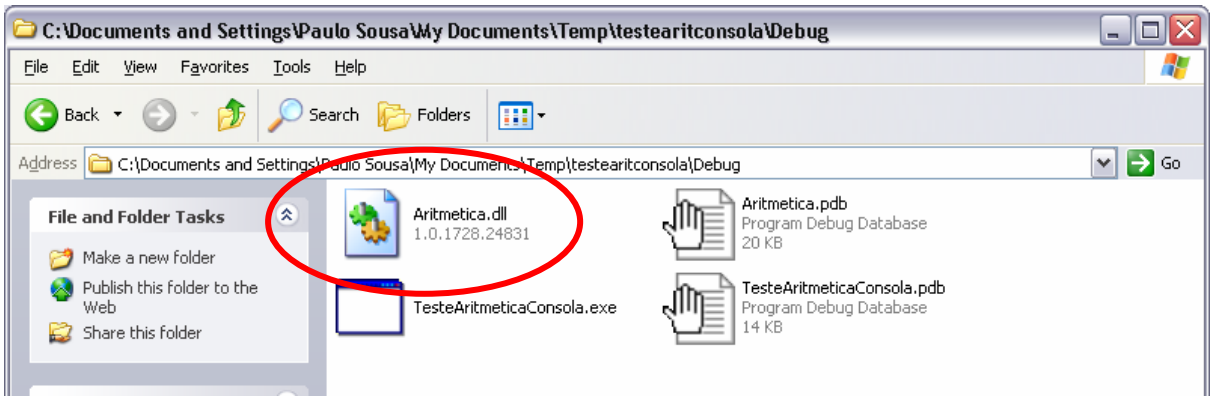

Figura 42 - versão da DLL da 1ª versão do componente

4. abrir uma nova janela do explorer no directório "bin" correspondente ao componente e verificar a versão da DLL (Figura 43).[10]

[10] As versões das DLL existentes no vosso computador serão diferentes das apresentadas na figura. O que importa reter é que a versão existente é diferente da nova versão compilada após as alterações.

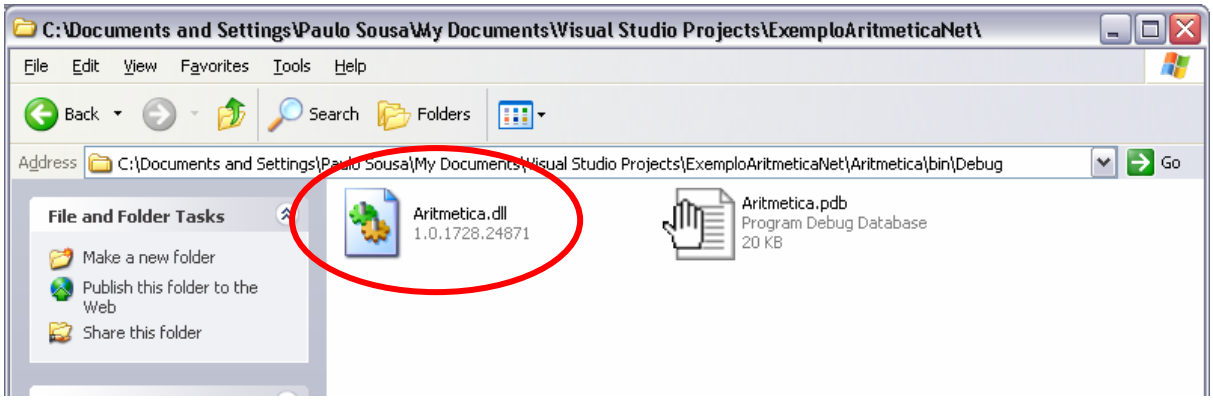

Figura 43 – versão da DLL para a 2ª versão do componente

5. copiar a nova DLL para o directório temporário
6. executar a aplicação e verificar que continua a funcionar.

Com estes passos consegue-se demonstrar que se pode evoluir a aplicação e que como se respeitou o **princípio da imutabilidade das interfaces** as aplicações clientes já existentes continuam a funcionar sem necessidade de alteração de código nem sequer recompilação .

6.6 Questões

1 Tendo em conta as alterações pedidas para efectuar nos pontos 2 e 3 da secção 5.7 Melhorias propostas e questões (pág. 58):

   a. Verifique se implementou uma nova interface ou alterou a interface existente já publicada. Reflicta sobre a decisão que tomou.

   b. será que esse serviço devia ser implementado neste componente ou num componente separado?

2 Suponha que deseja adicionar um serviço para aritmética com números complexos ($x+iy$). Como procederia para adicionar esse serviço ao componente? Das quatro hipóteses seguintes indique vantagens e desvantagens de cada uma e escolha justificadamente a opção que seguiria:

   a. Alterava a interface existente

   b. Aproveitava a alteração actual e incluía esses métodos na interface `IAritmetica2`

   c. Criava evolução da interface existente (eg., `IAritmetica3`)

   d. Criava nova interface, `IArtimeticaComplexa`, separada

7 Guião de trabalho: Separação da criação de classes

7.1 Introdução

Um dos problemas do componente que temos vindo a trabalhar é que continua a expor alguns pormenores de implementação internos às aplicações clientes, nomeadamente a classe que implementa os serviços. Essa situação nem sempre é desejável.

Por exemplo, a equipa de desenvolvimento podia querer separar a implementação da 2ª versão da interface da 1ª e para isso criar uma nova classe (ex, `Aritmetica2Impl`). No entanto, essa situação traria alguma confusão aos utilizadores do componente pois agora teriam duas interfaces e duas classes para perceber e necessitariam de saber que apenas poderiam usar a interface `IAritmetica` com a classe `ArtimeticaImpl` e a interface `IAritmetica2` com a classe `Aritmetica2Impl`.

Por outro lado, a solução apresentada também coloca alguns problemas, pois caso se pretenda estender os serviços com novas interfaces, a classe de implementação começará a ficar muito grande e aumentará a dificuldade de manutenção de código da mesma.

Uma solução possível é recorrer ao padrão[11] Factory e delegar a criação das classes de implementação de serviços numa classe utilitária – a fábrica.

7.2 Alteração do componente

Vamos abrir o projecto do componente e criar uma nova classe chamada `Factory` com os seguintes métodos:

```
public class Factory
{
      public static IAritmetica CreateIAritmeticaService()
      {
            return new AritmeticaImpl();
      }

      public static IAritmetica2 CreateIAritmetica2Service()
      {
            return new AritmeticaImpl();
      }
```

[11] Um padrão é uma solução tipificada para um problema recorrente. Podem encontrar mais informação em “Design patterns : elements of reusable object-oriented software”, Erich Gamma, Richard Helm, Ralph Johnson e John Vissides, e obter implementações base em C# em http://www.dofactory.com/Patterns/Patterns.aspx .

```
}
```

Esta classe passa a ser então o ponto de contacto com a aplicação cliente. O utilizador do componente apenas necessita saber as interfaces existentes e requisitar à fábrica um objecto que implemente o serviço desejado. De notar que se optou por uma regra de nomenclatura `CreateXXXService`, em que `XXX` representa a interface desejada.

Futuras evoluções apenas necessitarão da criação de novos métodos fábrica. Deve-se ter em atenção que o princípio de imutabilidade das interfaces também deve aqui ser respeitado, não sendo aconselhável alterar os métodos fábrica já existentes, mas sim criar novos métodos.

Como agora a criação das classes está separada numa classe utilitário, essas classes podem evoluir de forma independente. Um outro passo necessário é "esconder" as classes de implementação do exterior, para isso vamos alterar o tipo de visibilidade da classe `AritmeticaImpl` de `public` para `internal`. O que faz com que esta classe não possa ser instanciada fora do *assembly* (neste caso, a DLL) onde é definida.

```
internal class AritmeticaImpl : IAritmetica , IAritmetica2
{
      ...
}
```

7.3 Aplicação de teste

Para experimentar esta nova versão do componente iremos criar uma nova aplicação de teste adicionando um novo projecto de consola em C#, denominado "TesteFactory" à solução.

Adicionar a referência ao componente.

Para testar colocar o seguinte código:

```
int op1 = 5;
int op2 = 7;

// obter implementação de serviço desejado
Aritmetica.IAritmetica comp =
                  Aritmetica.Factory.CreateIAritmeticaService();

//invocar o serviço pretendido
Console.WriteLine("Adição({0}, {1}) => {2}",
                  op1, op2, comp.Adicao(op1, op2));
```

Compilar (não deve dar erros) e testar.

7.4 Implicações nas aplicações já existentes

Estas alterações têm um grande impacto nas aplicações já existentes implicando alteração do código fonte e recompilação para utilização da classe fábrica.

Uma hipótese de minimizar esse impacto é deixar a classe `AritmeticaImpl` com visibilidade `public`. Desse modo todas as aplicações existentes continuam a funcionar criando directamente a classe de implementação, enquanto que os novos clientes usariam apenas a classe de fábrica. Nesse caso, todas as implementações de novos serviços deveriam ser colocadas em novas classes com visibilidade `internal`.

8 Informação Adicional

MSDN Library

http://msdn.microsoft.com/library

Design Guidelines for Class Library Developers

http://msdn.microsoft.com/library/en-us/cpgenref/html/cpconnetframeworkdesignguidelines.asp

.net framework center

http://msdn.microsoft.com/netframework/

C#

http://msdn.microsoft.com/vcsharp/

ECMA

http://www.ecma-international.org/

Introduction to C# @ ECMA

http://www.ecma-international.org/activities/Languages/Introduction%20to%20Csharp.pdf

Common Language Infrastructure @ ECMA

http://www.ecma-international.org/activities/Languages/ECMA%20CLI%20Presentation.pdf

Using ADO.net

http://msdn.microsoft.com/netframework/using/understanding/data/default.aspx?pull=/library/en-us/dndotnet/html/usingadonet.asp

ASP.net

http://www.asp.net

Winforms

http://www.windowsforms.net/

Laboratório .net do ISEP/IPP

http://www.dei.isep.ipp.pt/labdotnet/

Open CLI

http://sourceforge.net/projects/ocl

Mono (.net @ Unix)

http://www.go-mono.com/