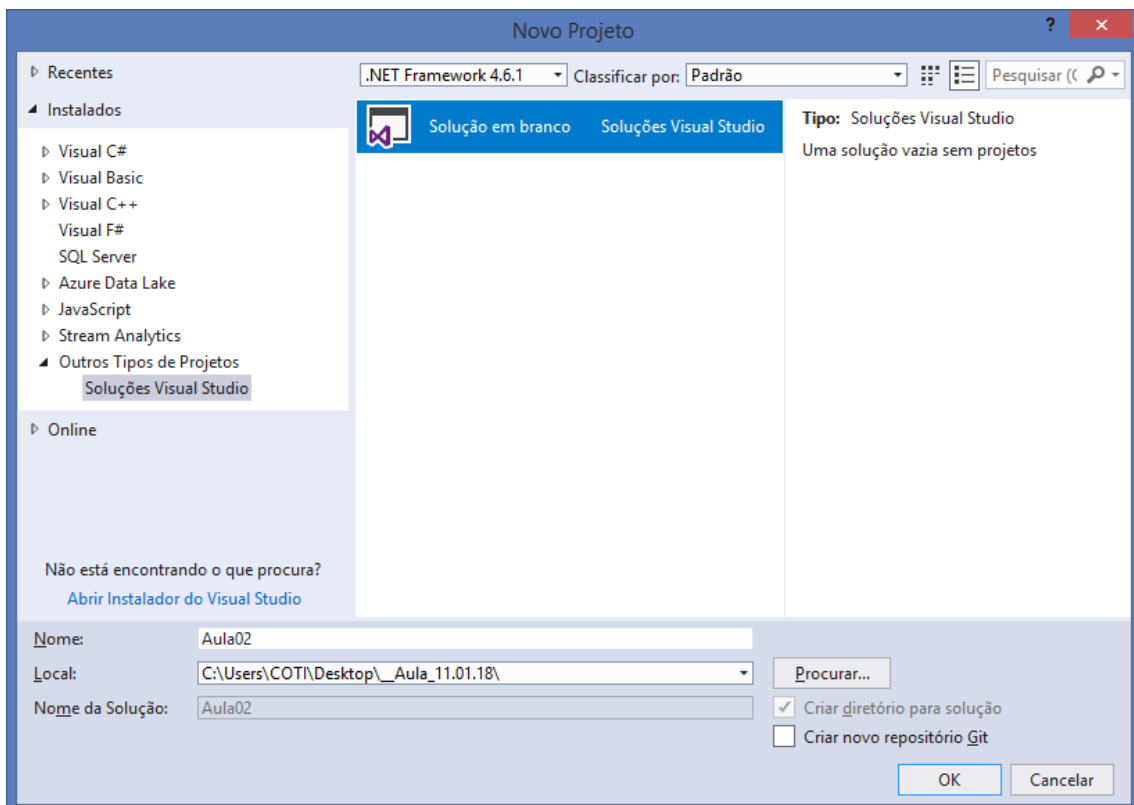
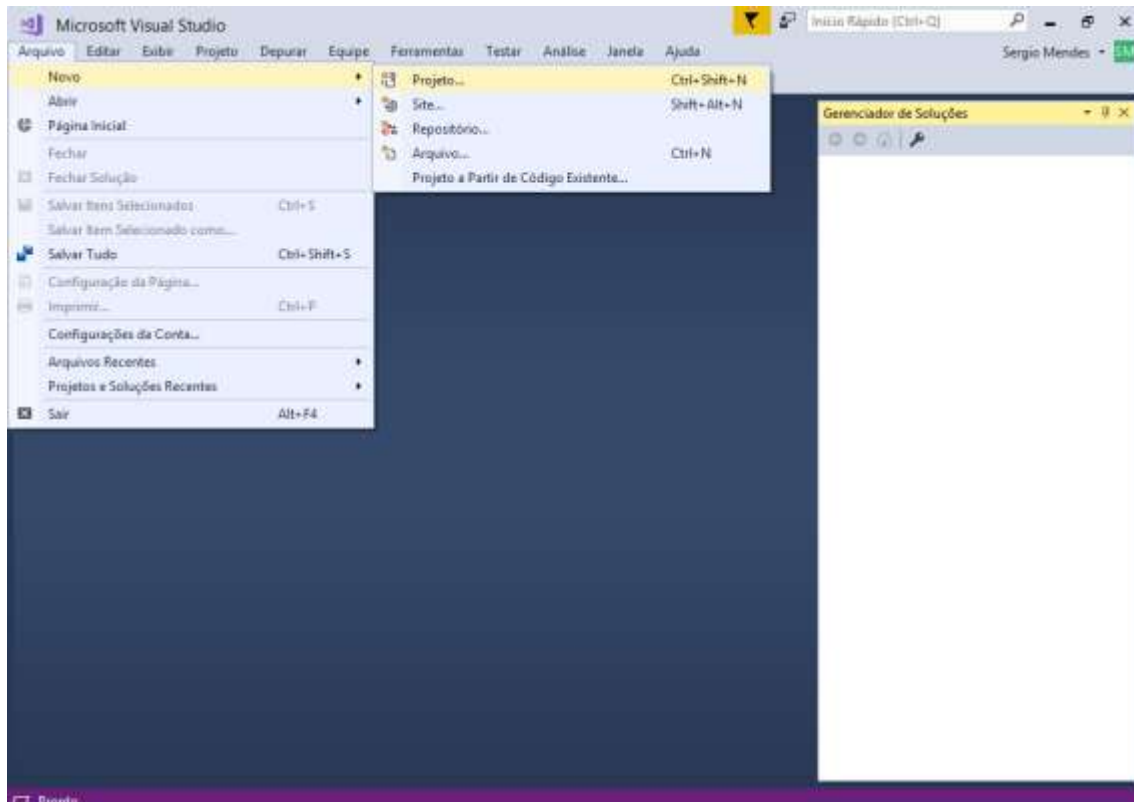


Criando uma nova solution em branco:



Criando um primeiro projeto **Console Application**

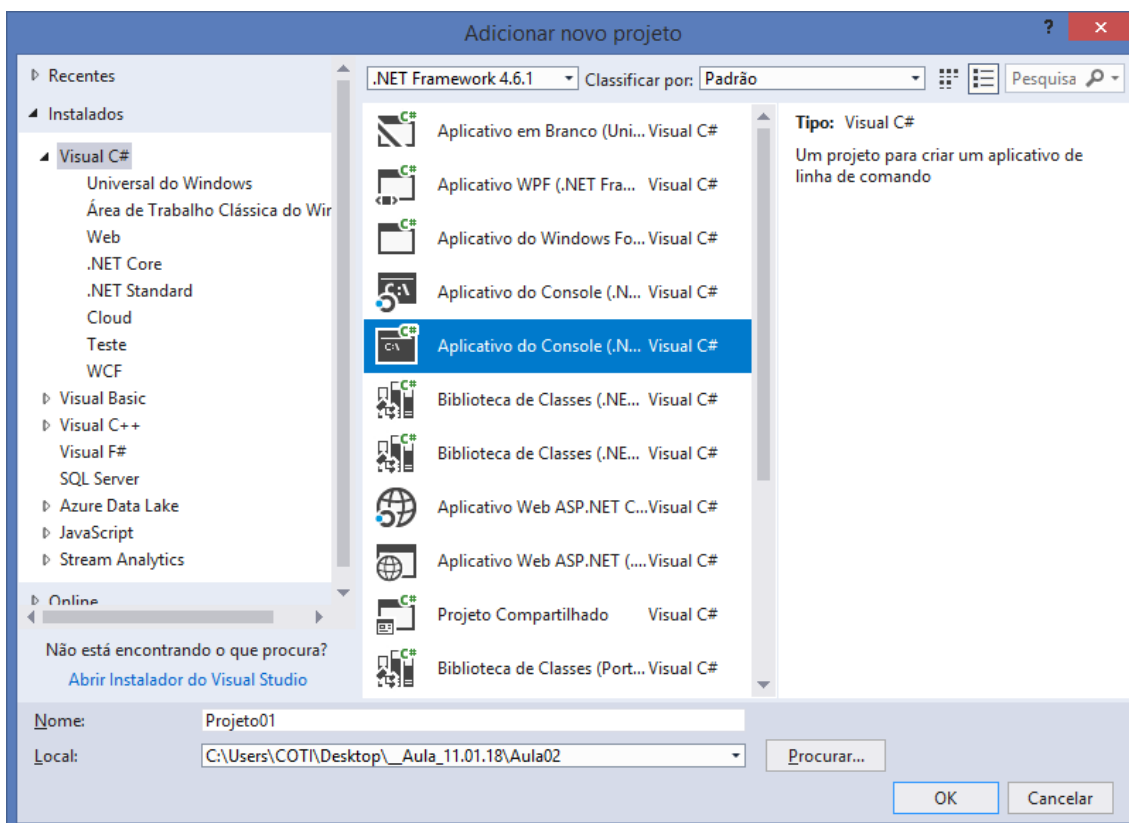
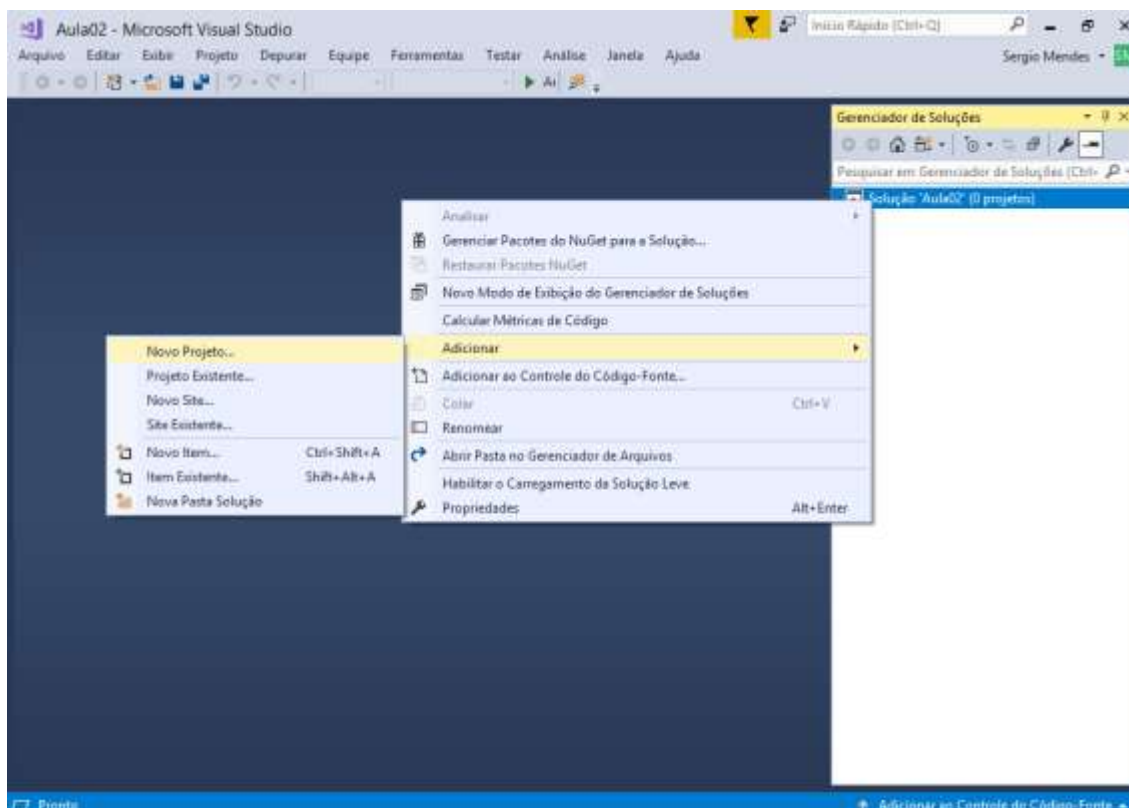
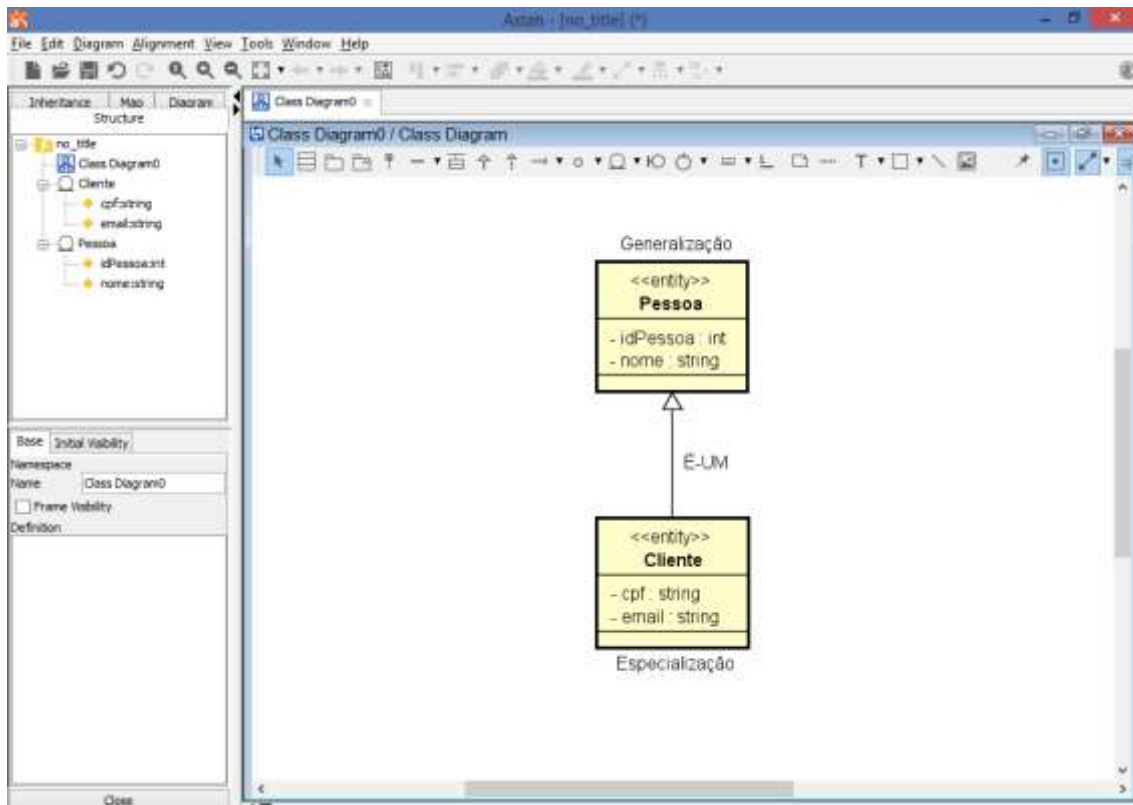


Diagrama de Classes

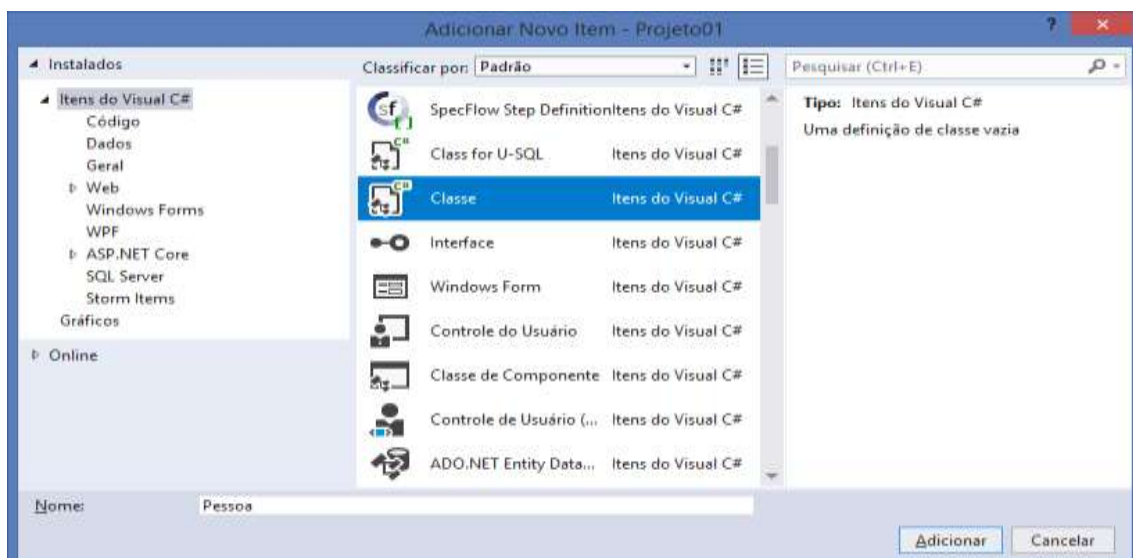
Modelagem Orientada a Objetos

Herança (SER)

Tipo de relacionamento entre superclasse e subclasses, ou seja, define uma relação de hierarquia (generalização / especialização)



Criando a classe Pessoa:



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Projeto01.Entidades
{
    public class Pessoa
    {
        #region Atributos

        private int idPessoa;
        private string nome;

        #endregion

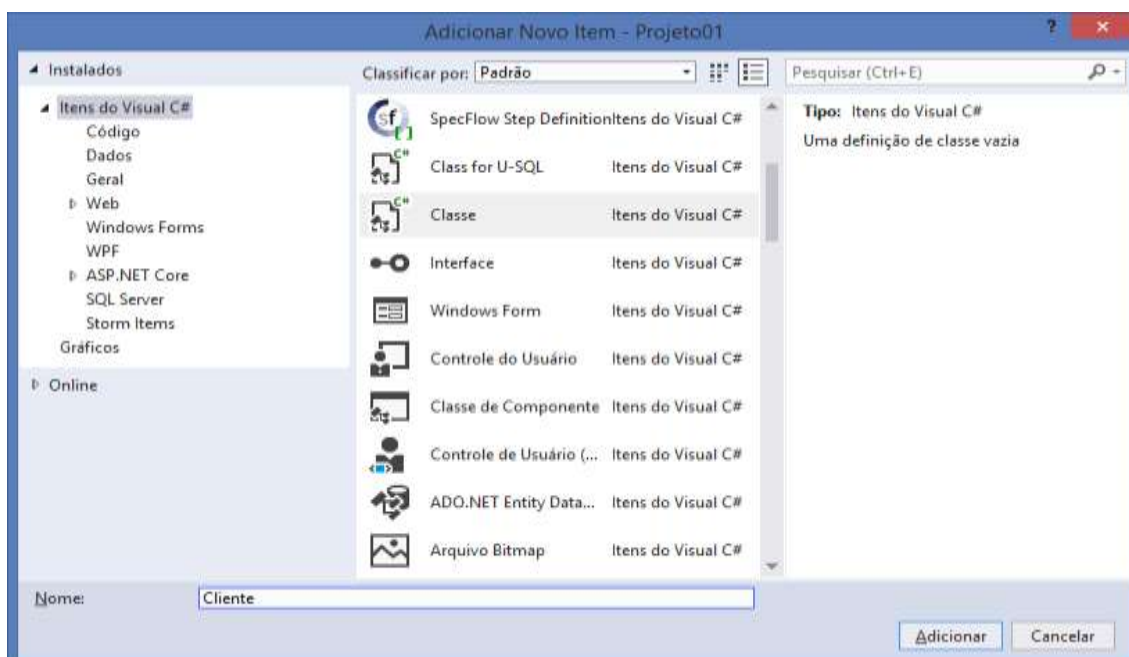
        #region Métodos de Encapsulamento

        public int IdPessoa
        {
            set { idPessoa = value; } //entrada
            get { return idPessoa; } //saida
        }

        public string Nome
        {
            set { nome = value; } //entrada
            get { return nome; } //saida
        }

        #endregion
    }
}
```

Criando a subclasse "Cliente":



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Projeto01.Entidades
{
    //Herança: Cliente É-UMA Pessoa
    public class Cliente : Pessoa
    {
        #region Atributos

        private string cpf;
        private string email;

        #endregion

        #region Métodos de Encapsulamento

        public string Cpf
        {
            set { cpf = value; }
            get { return cpf; }
        }

        public string Email
        {
            set { email = value; }
            get { return email; }
        }

        #endregion
    }
}
```

Regras sobre Herança

1) Em C#, não é permitido herança múltipla entre classes.

Exemplo:

```
public class A
{
}

public class B
{
}

public class C : A, B
{
}
```

2) Em C#, se uma classe é declarada como **sealed**, esta não poderá ser herdada (não poderá ter filhos).

Exemplo:

```
public class A
{
}

public sealed class B : A
{
}

public class C :-B
{
}
```

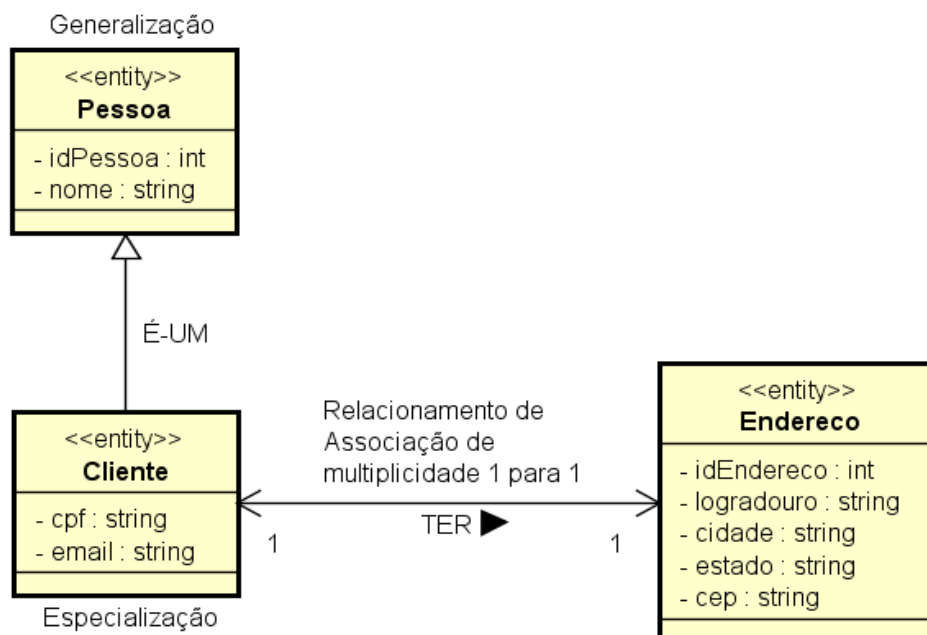
Associação (TER)

Tipo de relacionamento entre classes que define uma dependência de TODO / PARTE, por exemplo: Cliente TEM Endereco, Funcionario TEM Dependentes, etc...

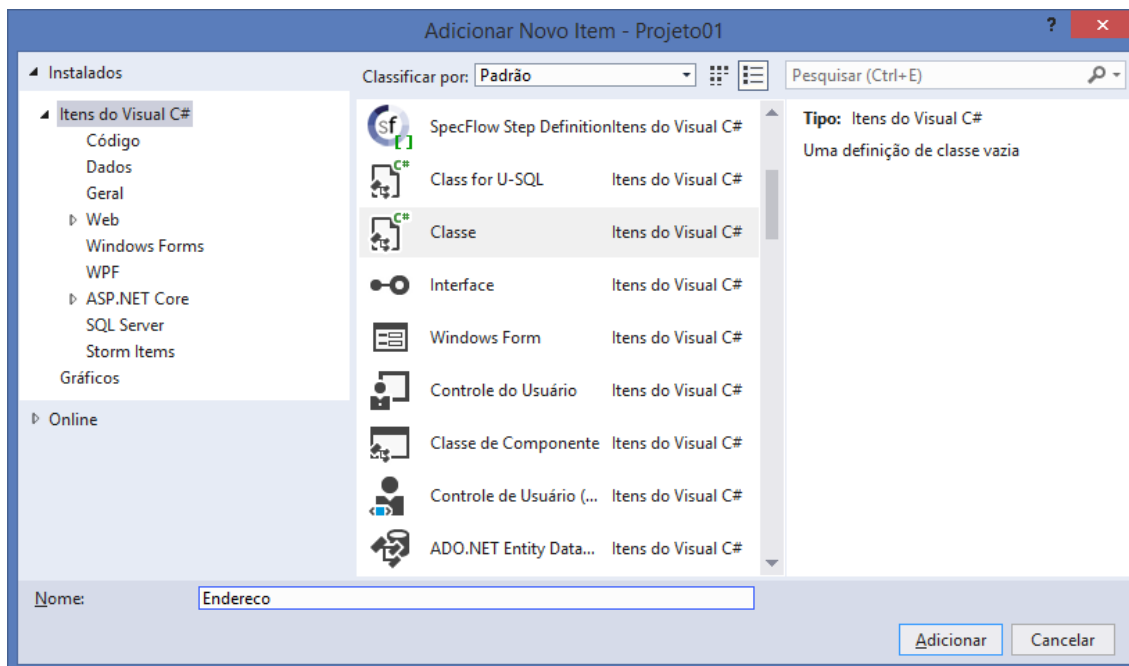
Toda relação de associação gera uma multiplicidade (cardinalidade) e pode ser de:

- **1 para 1**
- **1 para muitos**
- **muitos para 1**
- **muitos para muitos.**

Exemplo:



Criando a classe Endereco:



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Projeto01.Entidades
{
    public class Endereco
    {
        #region Atributos

        private int idEndereco;
        private string logradouro;
        private string cidade;
        private string estado;
        private string cep;

        #endregion

        #region Métodos de Encapsulamento

        public int IdEndereco
        {
            set { idEndereco = value; } //entrada
            get { return idEndereco; } //saida
        }

        public string Logradouro
        {
            set { logradouro = value; } //entrada
            get { return logradouro; } //saida
        }
    }
}
```

```

        public string Cidade
        {
            set { cidade = value; }
            get { return cidade; }
        }

        public string Estado
        {
            set { estado = value; }
            get { return estado; }
        }

        public string Cep
        {
            set { cep = value; }
            get { return cep; }
        }

        #endregion
    }
}

```

Relacionando Cliente com Endereco

Associação de multiplicidade 1 para 1

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Projeto01.Entidades
{
    //Herança: Cliente É-UMA Pessoa
    public class Cliente : Pessoa
    {
        #region Atributos

        private string cpf;
        private string email;
        private Endereco endereco; //Associação (TER-1)

        #endregion

        #region Métodos de Encapsulamento

        public string Cpf
        {
            set { cpf = value; }
            get { return cpf; }
        }

        public string Email
        {
            set { email = value; }
            get { return email; }
        }
    }
}

```



```

        public Endereco Endereco
        {
            set { endereco = value; }
            get { return endereco; }
        }

        #endregion
    }
}

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Projeto01.Entidades
{
    public class Endereco
    {
        #region Atributos

        private int idEndereco;
        private string logradouro;
        private string cidade;
        private string estado;
        private string cep;
        private Cliente cliente; //Associação (TER-1)

        #endregion

        #region Métodos de Encapsulamento

        public int IdEndereco
        {
            set { idEndereco = value; } //entrada
            get { return idEndereco; } //saida
        }

        public string Logradouro
        {
            set { logradouro = value; } //entrada
            get { return logradouro; } //saida
        }

        public string Cidade
        {
            set { cidade = value; }
            get { return cidade; }
        }

        public string Estado
        {
            set { estado = value; }
            get { return estado; }
        }

        public string Cep
        {

```

```

        set { cep = value; }
        get { return cep; }
    }

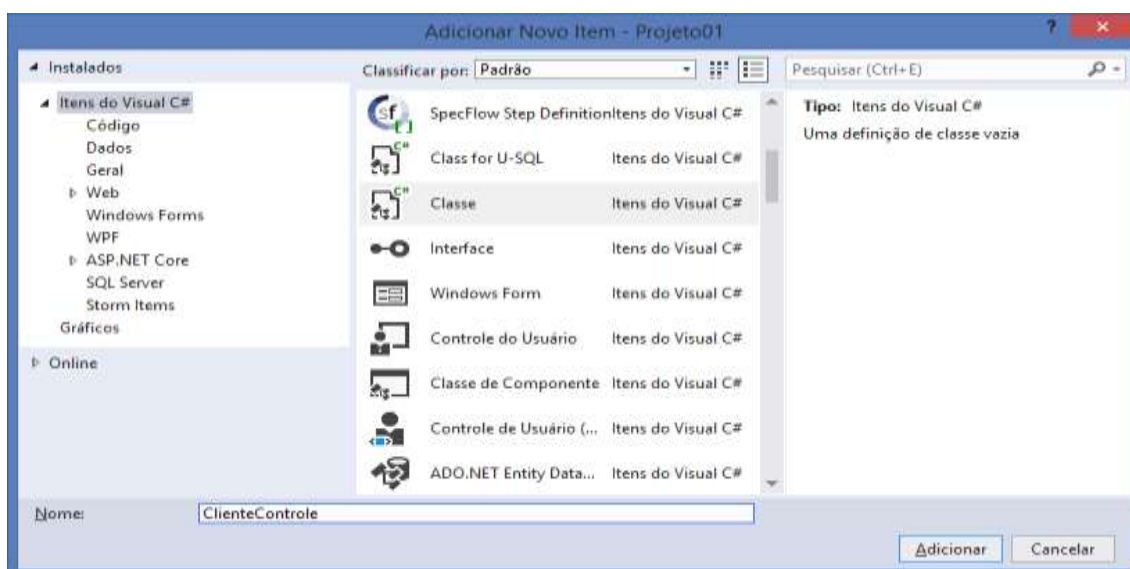
    public Cliente Cliente
    {
        set { cliente = value; }
        get { return cliente; }
    }

    #endregion
}

```

Classe para exportação de arquivos..

Formato: XML (eXtensible Markup Language)



XML (eXtensible Markup Language)

Metalinguagem baseado em HTML para armazenar dados em estruturas de tags. Como por exemplo:

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<cliente>
  <idpessoa>1</idpessoa>
  <nome>Sergio Mendes</nome>
  <email>sergio.coti@gmail.com</email>
  <cpf>123.456.789</cpf>
  <endereco>
    <idendereco>1</idendereco>
    <logradouro>Av Rio Branco 185, Centro</logradouro>
    <cidade>Rio de Janeiro</cidade>
    <estado>RJ</estado>
    <cep>25000-000</cep>
  </endereco>
</cliente>

```



```
<?xml version="1.0" encoding="ISO-8859-1"?>
- <cliente>
  <idpessoa>1</idpessoa>
  <nome>Sergio Mendes</nome>
  <email>sergio.coti@gmail.com</email>
  <cpf>123.456.789</cpf>
  - <endereco>
    <idendereco>1</idendereco>
    <logradouro>Av Rio Branco 185, Centro</logradouro>
    <cidade>Rio de Janeiro</cidade>
    <estado>RJ</estado>
    <cep>25000-000</cep>
  </endereco>
</cliente>
```

```
using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using Projeto01.Entidades; //importando..

using System.IO; //manipulação de arquivos..s

namespace Projeto01.Controles
{
    public class ClienteControle
    {
        //método para exportar os dados do cliente para
        //um arquivo de formato .XML
        public void ExportarParaXml(Cliente c)
        {
            //variavel para armazenar o nome do arquivo..
            //exemplo: [cliente_11012018191600.xml]
            string nomeArquivo = string.Format
                ("cliente_{0:ddMMyyyyHHmmss}.xml", DateTime.Now);
```

```
//criando um arquivo XML..
StreamWriter sw = new StreamWriter("c:\\temp\\" + nomeArquivo);

sw.WriteLine("<?xml version='1.0' encoding='ISO-8859-1'?>");
sw.WriteLine("<cliente>");
    sw.WriteLine("<idpessoa>{0}</idpessoa>", c.IdPessoa);
    sw.WriteLine("<nome>{0}</nome>", c.Nome);
    sw.WriteLine("<email>{0}</email>", c.Email);
    sw.WriteLine("<cpf>{0}</cpf>", c.Cpf);
    sw.WriteLine("<endereco>");
        sw.WriteLine("<idendereco>{0}</idendereco>",
            c.Endereco.IdEndereco);
        sw.WriteLine("<logradouro>{0}</logradouro>",
            c.Endereco.Logradouro);
        sw.WriteLine("<cidade>{0}</cidade>", c.Endereco.Cidade);
        sw.WriteLine("<estado>{0}</estado>", c.Endereco.Estado);
        sw.WriteLine("<cep>{0}</cep>", c.Endereco.Cep);
    sw.WriteLine("</endereco>");
sw.WriteLine("</cliente>");

//fechando o arquivo..
sw.Close();
    }
}
```

Tratamento de Exceções

Erros que ocorrem não em tempo de compilação mas sim em tempo de execução, ou seja, quando "rodamos" e testamos uma rotina do sistema.

Para que possamos evitar e tratar estes tipos de erros, podemos utilizar um bloco de programação denominado **try** e **catch**

Exception

Nome da classe em C# que captura qualquer tipo de erro ocorrido em tempo de execução.

```
try //tentativa
{
    //...
}
catch(Exception e) //captura da exceção
{
    //...
}
```

Instanciando a classe Cliente (Espaço de memória) e declarando um objeto

Cliente c = new Cliente();

[Classe] [Objeto] [Construindo espaço de memória (Instância)]

É necessário também instanciar as classes que estão relacionadas a Cliente, como por exemplo: Endereco.

c.Endereco = new Endereco();

[Relacionamento] [Construindo espaço de memória (Instância)]

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Projeto01.Entidades; //importando..
using Projeto01.Controles; //importando..

namespace Projeto01
{
    class Program
    {
        static void Main(string[] args)
        {
            //instanciar um objeto da classe Cliente..
            Cliente c = new Cliente();
            c.Endereco = new Endereco();

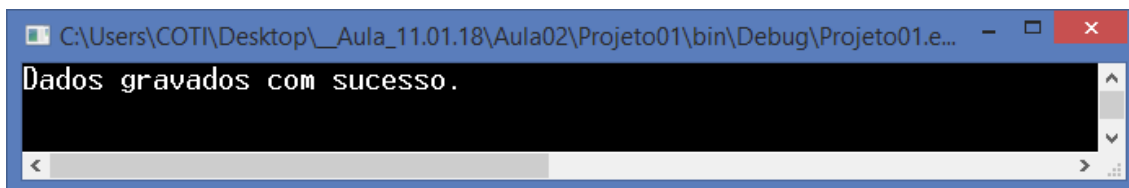
            c.IdPessoa = 1;
            c.Nome = "Sergio Mendes";
            c.Email = "sergio.coti@gmail.com";
            c.Cpf = "123.456.789-00";
            c.Endereco.IdEndereco = 1;
            c.Endereco.Logradouro = "Av Rio Branco, 185 Centro";
            c.Endereco.Cidade = "Rio de Janeiro";
            c.Endereco.Estado = "RJ";
            c.Endereco.Cep = "25000-000";

            try //tentativa
            {
                ClienteControle cc = new ClienteControle();
                cc.ExportarParaXml(c); //gravando..

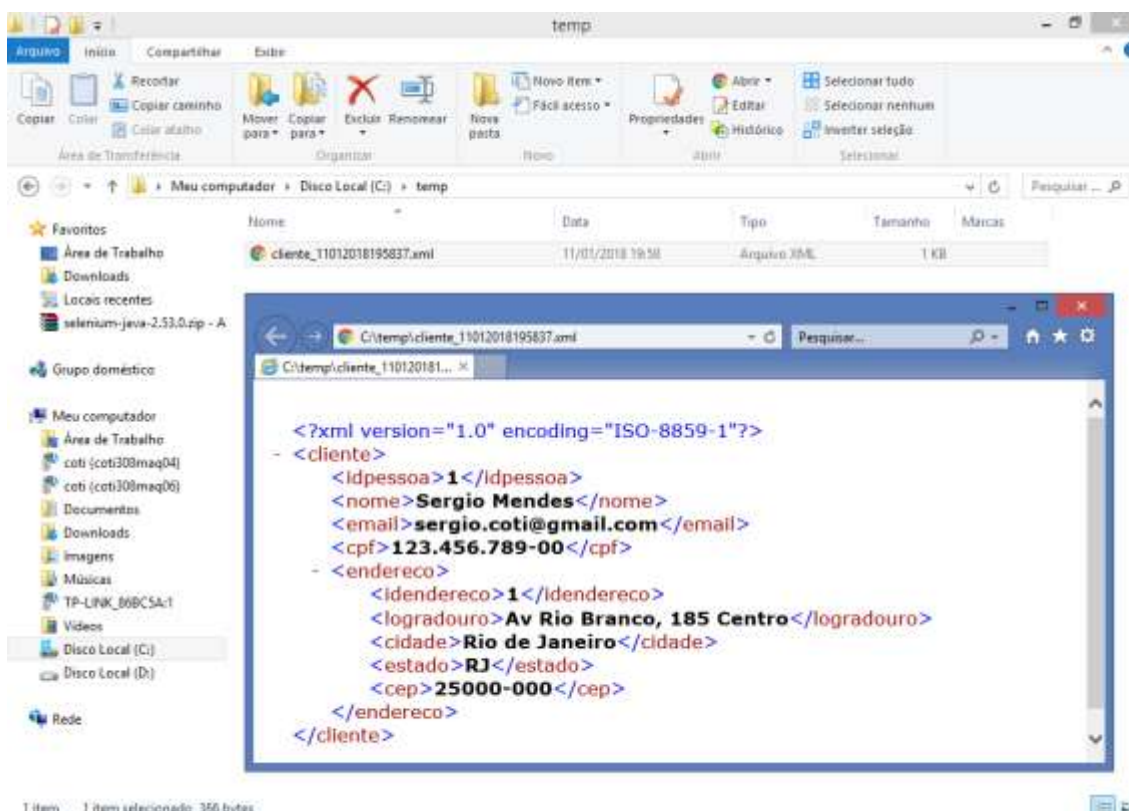
                Console.WriteLine("Dados gravados com sucesso.");
            }
            catch(Exception e) //captura da exceção
            {
                Console.WriteLine("Erro ao gravar dados: " + e.Message);
            }

            Console.ReadKey(); //pausar..
        }
    }
}
```

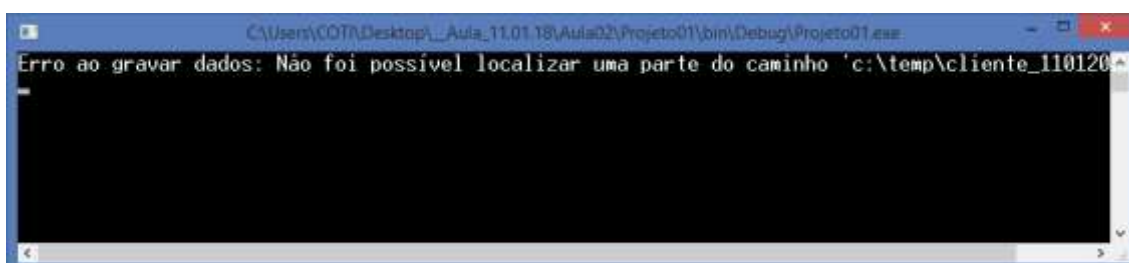
Executando:



Arquivo gerado:



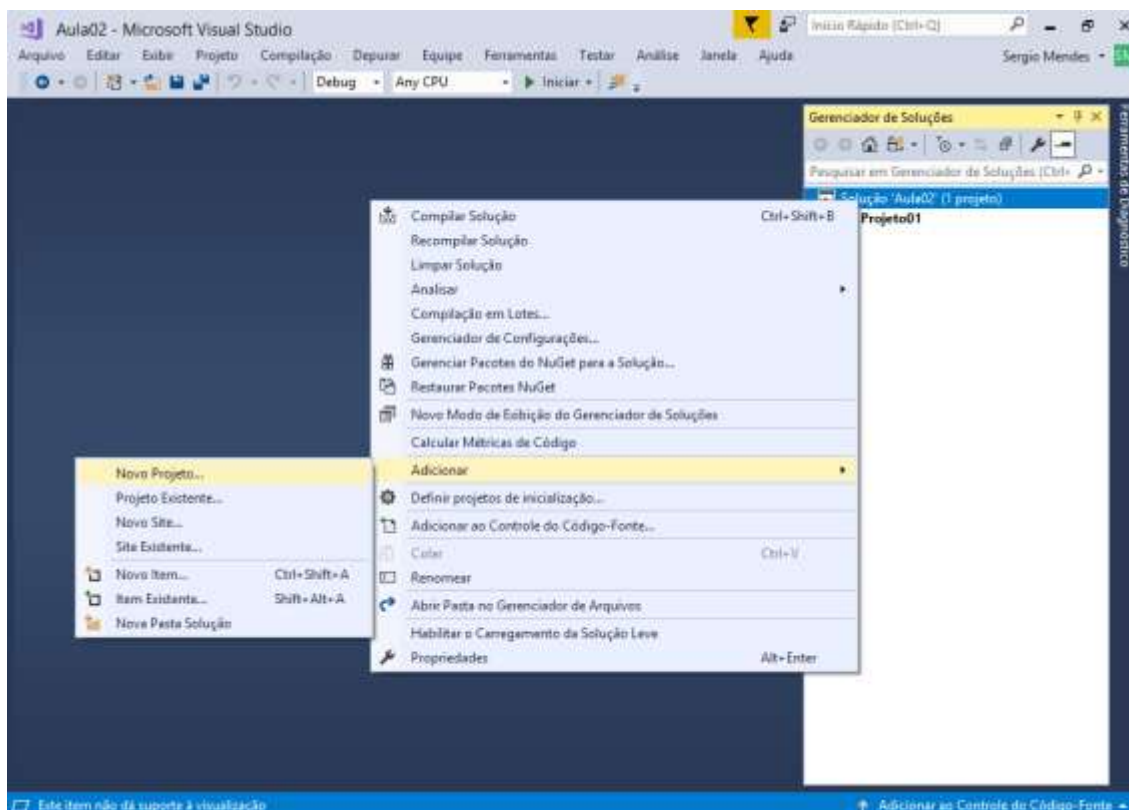
Obtendo uma exceção ao executar:



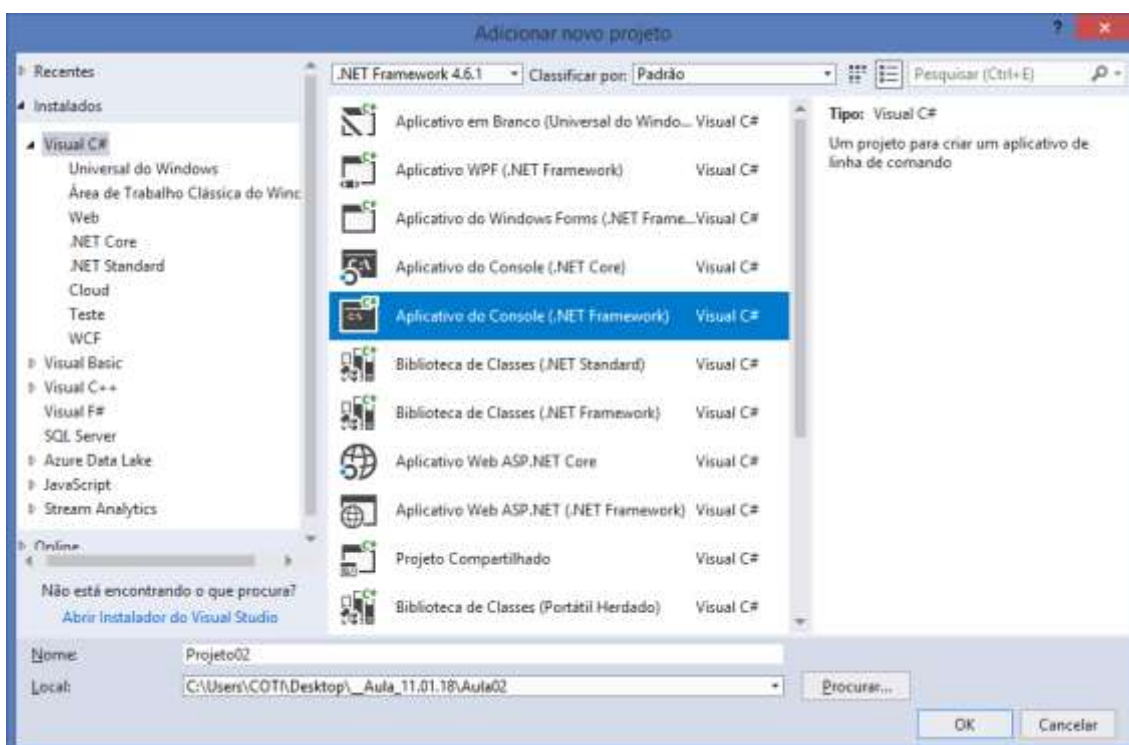
Erro ao gravar dados: Não foi possível localizar uma parte do caminho 'c:\temp\cliente_1101201820002.xml'

Criando um novo projeto:

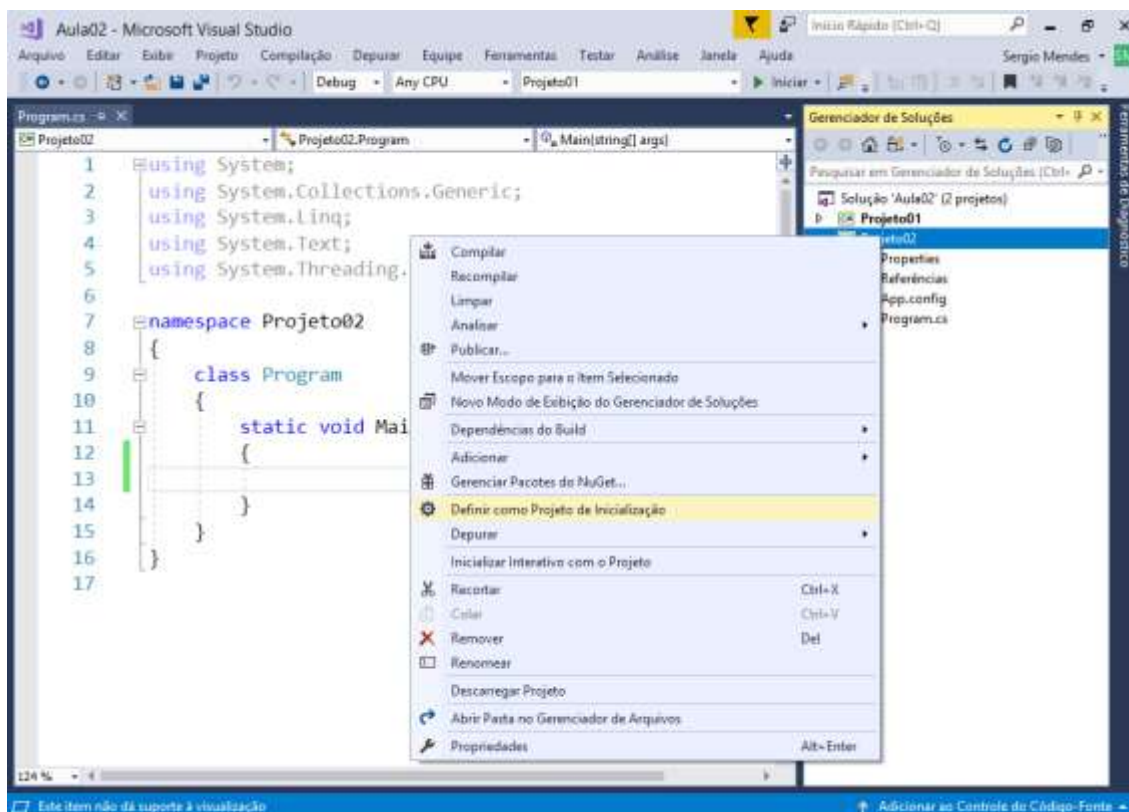
Console Application (.NET Framework)



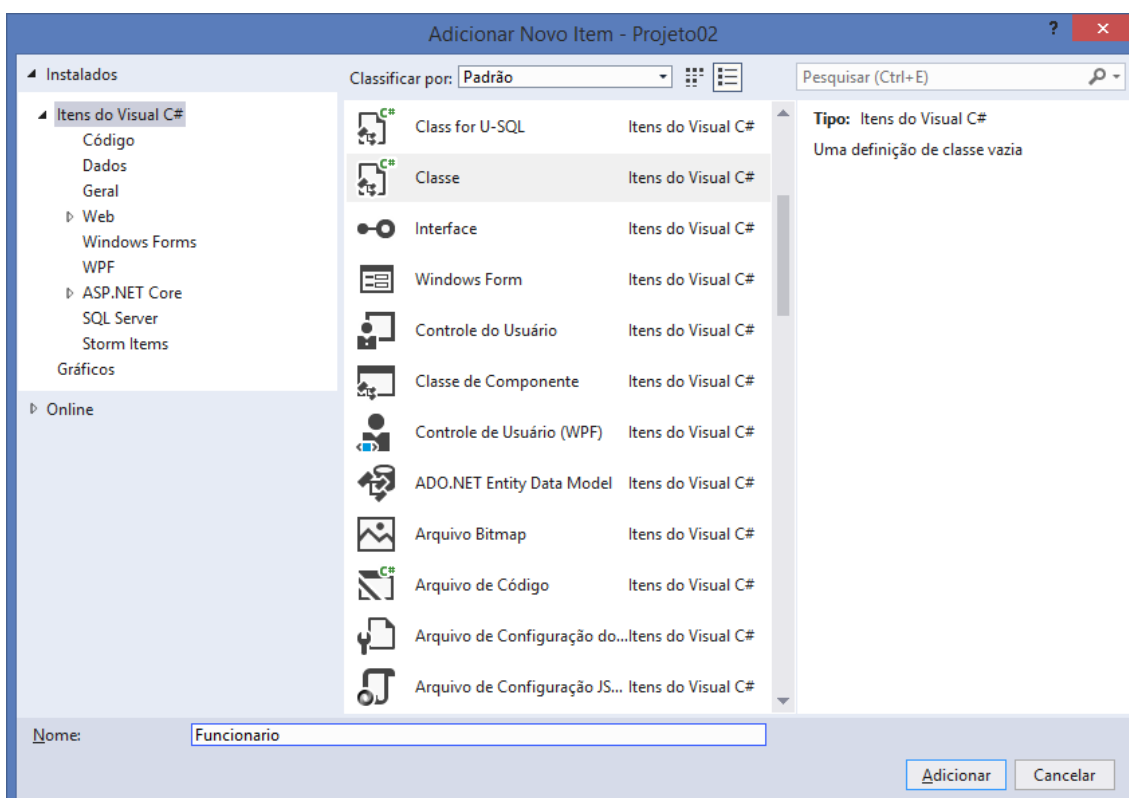
Nome: **Projeto02**



Definindo o projeto principal da solution:



Criando uma nova classe de entidade:



Encapsulamento implícito

Ocorre quando declaramos apenas os métodos `set` e `get` fazendo com que o compilador já assuma que a classe possui atributos privados para cada método `set` e `get`.

Encapsulamento "padrão":

```
private int idFuncionario;

public int IdFuncionario
{
    set { idFuncionario = value; }
    get { return idFuncionario; }
}
```

Encapsulamento "implícito":

```
public int IdFuncionario { get; set; }
```

```
-----

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Projeto02.Entidades
{
    public class Funcionario
    {
        //propriedade implicitamente encapsulada..
        //[prop] + 2x[tab]
        public int IdFuncionario { get; set; }
        public string Nome { get; set; }
        public double Salario { get; set; }
        public DateTime DataAdmissao { get; set; }
    }
}
```

Construtor

Método utilizado para inicializar os objetos criados para uma classe. É através do construtor que um objeto recebe espaço de memória para uma classe (instância).

Exemplo:

Funcionario f = new Funcionario();
[Classe - Tipo] [Objeto] [Método Construtor]

Para o compilador, toda classe em C# já possui um construtor implícito, que não precisa estar escrito no código. Podemos escrever o construtor de uma classer, ou seja, declarar este metodo de maneira que ele fique explicito no código.

Funcionario f = new Funcionario();

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Projeto02.Entidades
{
    public class Funcionario
    {
        //propriedade implicitamente encapsulada..
        //[prop] + 2x[tab]
        public int IdFuncionario { get; set; }
        public string Nome { get; set; }
        public double Salario { get; set; }
        public DateTime DataAdmissao { get; set; }

        //criando o método construtor da classe..
        //[ctor] + 2x[tab]
        public Funcionario()
        {
            //default..
        }
    }
}
```

Sobrecarga de Métodos (**Overloading**)

Ocorre quando declaramos em uma classe métodos com o mesmo nome, porem com entrada de argumentos diferentes.

Exemplo:

```
public class ControlePagamento
{
    public double CalcularPgto(double valor)
    {
        return 0.0;
    }

    public double CalcularPgto(double valor, int parcelas)
    {
        return 0.0;
    }

    public double CalcularPgto(double valor, int parcelas, double juros)
    {
        return 0.0;
    }
}
```

Podemos utilizar a sobrecarga de métodos também na declaração dos **construtores** de uma classe, ou seja, podemos criar em uma classe varios construtores sendo cada um com entrada de argumentos diferentes.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Projeto02.Entidades
{
    public class Funcionario
    {
        //propriedade implicitamente encapsulada..
        //[prop] + 2x[tab]
        public int IdFuncionario { get; set; }
        public string Nome { get; set; }
        public double Salario { get; set; }
        public DateTime DataAdmissao { get; set; }

        //criando o método construtor da classe..
        //[ctor] + 2x[tab]
        public Funcionario()
        {
            //default..
        }
    }
}
```

```
//sobrecarga (overloading) de método construtor..
public Funcionario(int idFuncionario, string nome,
                   double salario, DateTime dataAdmissao)
{
    IdFuncionario = idFuncionario;
    Nome = nome;
    Salario = salario;
    DataAdmissao = dataAdmissao;
}
}
```

Executando o construtor default:

Funcionario f = new Funcionario();

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Projeto02.Entidades
{
    public class Funcionario
    {
        //propriedade implicitamente encapsulada..
        //[prop] + 2x[tab]
        public int IdFuncionario { get; set; }
        public string Nome { get; set; }
        public double Salario { get; set; }
        public DateTime DataAdmissao { get; set; }

        //criando o método construtor da classe..
        //[ctor] + 2x[tab]
        public Funcionario()
        {
            //default..
        }

        //sobrecarga (overloading) de método construtor..
        public Funcionario(int idFuncionario, string nome,
                           double salario, DateTime dataAdmissao)
        {
            IdFuncionario = idFuncionario;
            Nome = nome;
            Salario = salario;
            DataAdmissao = dataAdmissao;
        }
    }
}
```

Executando o construtor com entrada de argumentos:

Funcionario f = new Funcionario(1, "Ana", 1000.0, DateTime.Now);

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Projeto02.Entidades
{
    public class Funcionario
    {
        //propriedade implicitamente encapsulada..
        //[prop] + 2x[tab]
        public int IdFuncionario { get; set; }
        public string Nome { get; set; }
        public double Salario { get; set; }
        public DateTime DataAdmissao { get; set; }

        //criando o método construtor da classe..
        //[ctor] + 2x[tab]
        public Funcionario()
        {
            //default..
        }

        //sobrecarga (overloading) de método construtor..
        public Funcionario(int idFuncionario, string nome,
                           double salario, DateTime dataAdmissao)
        {
            IdFuncionario = idFuncionario;
            Nome = nome;
            Salario = salario;
            DataAdmissao = dataAdmissao;
        }
    }
}
```

Testando:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Projeto02.Entidades;

namespace Projeto02
{

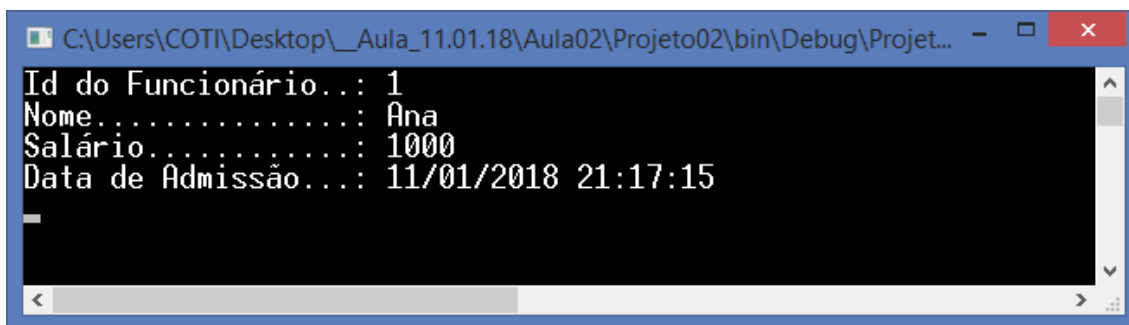
```

```
class Program
{
    static void Main(string[] args)
    {
        Funcionario f = new Funcionario(1, "Ana", 1000.0, DateTime.Now);

        //imprimindo..
        Console.WriteLine("Id do Funcionário...: " + f.IdFuncionario);
        Console.WriteLine("Nome.....: " + f.Nome);
        Console.WriteLine("Salário.....: " + f.Salario);
        Console.WriteLine("Data de Admissão....: " + f.DataAdmissao);

        Console.ReadKey();
    }
}
```

Saída do programa:



Sobrescrita de Métodos (Override)

A sobrescrita de métodos ocorre quando uma subclasse reprograma um método da sua superclasse, sem modificar a assinatura do método.

Para que uma subclasse possa sobrescrever um método de sua superclasse, a superclasse precisa declarar o método com a palavra reservada **virtual**

virtual

Faz com que um método de uma classe possa ser sobrescrito por alguma das suas subclasses.

Exemplo:

Considere a seguinte classe e seu método:

```
public class ControlePagamento
{
    public double CalcularPgto(double valor)
    {
        return 0.0;
    }
}
```

Caso a classe acima declare o seu método como "virtual", ela estará permitindo a uma subclasse sobrescrever o método.

Por exemplo:

Abaixo vemos a subclasse herdar um método de sua superclasse:

```
public class ControlePagamento
{
    public virtual double CalcularPgto(double valor)
    {
        return 0.0;
    }
}

public class ControlePagamentoDesconto : ControlePagamento
{
}

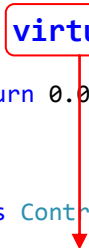
class Test
{
    void Main()
    {
        ControlePagamentoDesconto d = new ControlePagamentoDesconto();
        d.CalcularPgto(100.0);
    }
}
```

Com o método da superclasse é virtual, a subclasse poderia, se quisesse sobrescreve-lo e portanto alterar o seu resultado.

```
public class ControlePagamento
{
    public virtual double CalcularPgto(double valor)
    {
        return 0.0;
    }
}

public class ControlePagamentoDesconto : ControlePagamento
{
    public override double CalcularPgto(double valor)
    {
        return valor - 10;
    }
}

class Test
{
    void Main()
    {
        ControlePagamentoDesconto d = new ControlePagamentoDesconto();
        d.CalcularPgto(100.0);
    }
}
```



Exemplo:

```
public class Pai
{
    public virtual void Dirigir()
    {
        Console.WriteLine("Dirigindo com calma..");
    }
}

public class Filho : Pai
{
    public override void Dirigir()
    {
        Console.WriteLine("Dirigindo com emoção..");
    }
}

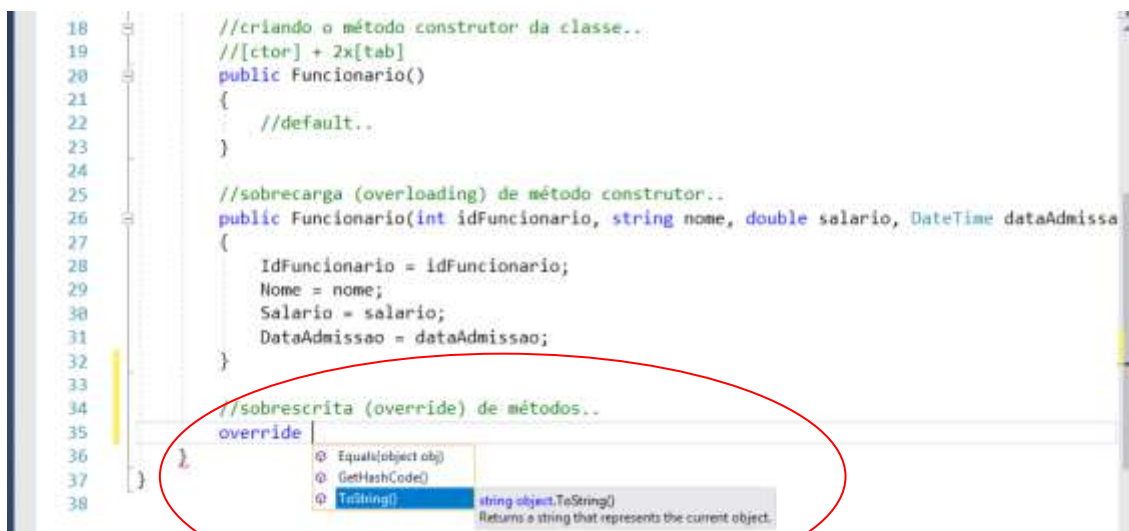
class Test
{
    void Main()
    {
        Filho f = new Filho();
        f.Dirigir();
    }
}
```

Toda classe em C# é Herança da Classe Object, portanto toda classe sempre herda os métodos: ToString, Equals, GetHashCode e GetType. Com exceção do método GetType, os demais são **virtual**, ou seja, se quisermos podemos sobrescreve-los.

Exemplo:

Fazendo uma sobrecrita do método ToString()

Método simples utilizado para retornar os dados de uma classe em uma unica linha de texto.




```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Projeto02.Entidades
{
    public class Funcionario
    {
        //propriedade implicitamente encapsulada..
        //[prop] + 2x[tab]
        public int IdFuncionario { get; set; }
        public string Nome { get; set; }
        public double Salario { get; set; }
        public DateTime DataAdmissao { get; set; }

        //criando o método construtor da classe..
        //[ctor] + 2x[tab]
        public Funcionario()
        {
            //default..
        }

        //sobrecarga (overloading) de método construtor..
        public Funcionario(int idFuncionario, string nome,
                           double salario, DateTime dataAdmissao)
        {
            IdFuncionario = idFuncionario;
            Nome = nome;
            Salario = salario;
            DataAdmissao = dataAdmissao;
        }

        //sobrescrita (override) de métodos..
        public override string ToString()
        {
            return string.Format("{0}, {1}, {2}, {3}",
                                  IdFuncionario, Nome, Salario, DataAdmissao);
        }
    }
}
```

Testando:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Projeto02.Entidades;

namespace Projeto02
{
    class Program
    {
```

```
static void Main(string[] args)
{
    Funcionario f = new Funcionario(1, "Ana", 1000.0, DateTime.Now);

    //imprimindo..
    Console.WriteLine("Id do Funcionário..: " + f.IdFuncionario);
    Console.WriteLine("Nome.....: " + f.Nome);
    Console.WriteLine("Salário.....: " + f.Salario);
    Console.WriteLine("Data de Admissão...: " + f.DataAdmissao);

    Console.WriteLine("Imprimindo com o método ToString()");
    Console.WriteLine("Funcionario: " + f.ToString());

    Console.ReadKey();
}
}
```

Executando:

