PRESYS INTRUMENTOS E SISTEMAS LTDA

AUTORES:
ALEXANDER VILAÇA RODRIGUES
ANDRÉ VÍTOR PEREIRA CINI
CLAUS ALBERTO BIENEMANN
GUILHERME DIAS LIMA TURTERA
LUCAS ANTÔNIO MENEGUELLI
MATHEUS DE NOVAIS SOUZA
NATHAN VILELA DE SOUZA

Apostila C# - Manual do Estagiário

SÃO PAULO, SP
2020

Sumário

1. CRIPTOGRAFIA

Criptografar algo é basicamente tentar esconder seu real significado, para que apenas pessoas específicas consigam ver o real conteúdo, a criptografia é algo muito comum desde as sociedades antigas e vem se modernizando até hoje.

Como exemplo de criptografia de antigamente, temos cartas de guerra que se fossem apreendidas pelo exército inimigo, revelariam informações importantes que poderia determinar o rumo da guerra, então as cartas eram escritas seguindo algum padrão escondido, ou com símbolos, criados apenas para que as pessoas do mesmo exército consigam entender o conteúdo.

Hoje em dia, ela é extremamente utilizada no nosso cotidiano, onde todos os dados que são inseridos em sites como Facebook, Google, Twitter, etc, precisam ser criptografados, para que ninguém consiga ter acesso às suas informações pessoais.

Existem diversos tipos de criptografia, como:

- **Criptografia de chave Simétrica:**

A Criptografia de chave simétrica é o modelo mais básico de criptografia, nesse tipo de criptografia, tanto o emissor do código cifrado, quanto o receptor que vai descifrar o código, utilizam a mesma chave, ou seja, um mesmo padrão para escrever e ler a mensagem.

Uma das aplicações dessa criptografia, como já dito anteriormente, em guerras de civilizações mais antigas.

- **DES (Data Encryption Standart):**

Esse modelo foi criado pela IBM em 1977, fornecendo uma proteção básica de 56bits, oferecendo até 72 quadrilhões de combinações.

Porém, seu lado negativo, é que através de um ataque de força bruta, sua chave pode ser quebrada. Um ataque de força bruta, basicamente significa testar todas as possibilidades de chave, até encontrar uma que condiz com a utilizada.

- **IDEA (International Data Encryption Algorithm)**

Esse modelo foi criado em 1991 e é muito semelhante ao DES, mas opera em blocos de informação de 64bits, mas utilizando chaves de 128bits, causando uma confusão para cifrar o texto, o que impede o realinhamento para a leitura das informações.

- **AES (Advanced Encryption Standard)**

É um dos mais seguros algoritmos de criptografia da atualidade, sua criptografia é feita em blocos de 128 bits, mas suas chaves podem ser aplicadas em 192 e 256 bits, o que a torna muito difícil de ser quebrada.

Sua aplicação está presente até mesmo no Governo dos Estados Unidos.

Sua aplicação pode ser feita no C#, adicionando a biblioteca System.Security.Cryptography;

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Security.Cryptography;
using System.IO;
```

O código ao lado simplesmente adiciona a biblioteca System.Security.Cryptography e a System.IO;

```
public static byte[] Key = new byte[32];
public static byte[] IV = new byte[16];
public static void GenerateKeys()
{
    using (AesManaged aes = new AesManaged())
    {
        Key = aes.Key;
        IV = aes.IV;
    }
}
```

Aqui Instanciamos dois vetores, que serão a chave e o IV, o IV ou vetor de inicialização, é utilizado para o primeiro bloco de criptografia seja sempre diferente, é possivel não utilizá-lo, mas o código ficaria menos seguro.
Já a classe GenerateKeys, gera a chave e o IV para a sua criptografia.

```
public static byte[] Encrypt(string text)
{
    byte[] encrypted;
    using (AesManaged aes = new AesManaged())
    {
        ICryptoTransform encryptor = aes.CreateEncryptor(Key, IV);
        using (MemoryStream ms = new MemoryStream())
        {
            using (CryptoStream cs = new CryptoStream(ms, encryptor, CryptoStreamMode.Write))
            {
                using (StreamWriter sw = new StreamWriter(cs))
                    sw.Write(text);
                encrypted = ms.ToArray();
            }
        }
    }
    return encrypted;
}
```

O código acima, cifra seu texto de acordo com as chaves geradas e te retorna um vetor de byte, que será seu texto cifrado.

```
public static string Decrypt(byte[] cipherText)
{
    string text = null;
    using (AesManaged aes = new AesManaged())
    {
        ICryptoTransform decryptor = aes.CreateDecryptor(Key, IV);
        using (MemoryStream ms = new MemoryStream(cipherText))
        {
            using (CryptoStream cs = new CryptoStream(ms, decryptor, CryptoStreamMode.Read))
            {
                using (StreamReader reader = new StreamReader(cs))
                    text = reader.ReadToEnd();
            }
        }
    }
    return text;
}
```

O código acima descifra o seu vetor de byte para string.

- **Criptografia de chave Assimétrica:**

Essa chave trabalha em modo privado e em modo público, para o modo privado, a chave é secreta, já para o modo público, o usuário terá que criar uma chave e encaminhá-la para o receptor, para ter acesso ao conteúdo.

2. THREAD E MULTITHREAD

2.1. Thread

Thread são por tradução linha de execução, sequência, segmentos, encadeamento e etc.

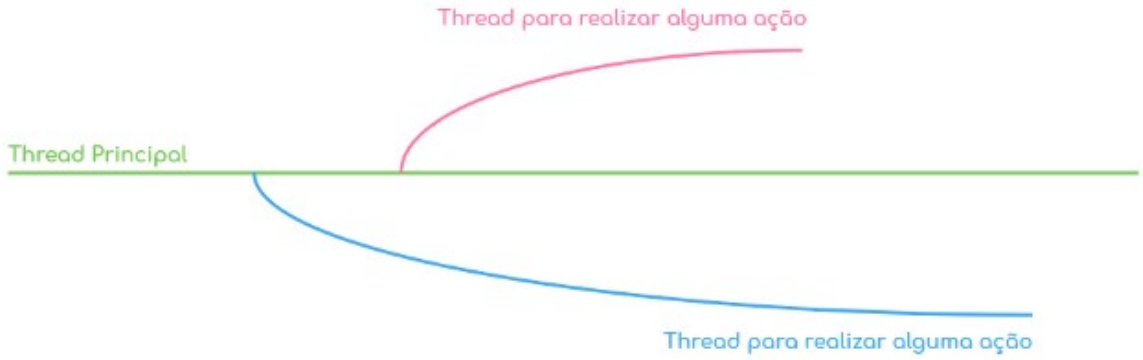
E esse recurso vem para dar mais performance e velocidade à execução do código, facilitando assim a execução de multitarefas. Deve – se tomar cuidados também ao utilizar threads em um código, isso se deve ao fato de threads funcionarem de maneira paralela ou de forma concorrente, mas o que seria isso?

- **Paralelo:**

Uma execução de threads em paralelo são threads que são executadas de uma forma **“simultânea”*** mas que não interferem no processo do outro, sendo assim não oferecendo muitos riscos ao código, por meio que esse fator utilizar mais de uma thread, isso à torna uma Multithread, mas que será abordado mais abaixo.

- **Concorrente:**

Uma execução de threads em concorrência são threads que são executadas no mesmo processo e assim podem interferir no processo uma da outra, gerando o problema do **impasse***, onde a partir de um impasse pode haver uma relação de erros crescentes, sendo assim, deve-se utilizar com cuidado e com uma boa organização.



2.2. Multithread

Multithread nada mais é que mais de uma thread num código, ou seja, subdivisões na hora de execuções de certos processos, onde cada thread está demandada para uma ou mais funções nas quais trabalham em conjunto com o sistema global. Os princípios anteriores se aplicam ao conceito de threads e também devem ser levadas em conta.

2.3. Cuidados na Utilização(Esclarecimentos)

Deve – se entender que as threads não executam simultaneamente de fato, mas sim uma intercalação entre threads de maneira rápida de modo onde o usuário não percebe esta troca, dando a entender o efeito simultâneo.

O impasse ocorre quando uma thread atrapalha um processo de uma outra thread executando antes ou depois do esperado, dificultando assim a execução do código de maneira “saudável”, ou de maneira esperada.

Códigos e Referências

Construtores	
Thread(ParameterizedThreadStart)	Inicializa uma nova instância da classe Thread, especificando um delegado que permite que um objeto seja passado para o thread quando o thread for iniciado.
Thread(ParameterizedThreadStart, Int32)	Inicializa uma nova instância da classe Thread, especificando um delegado que permite que um objeto seja passado para o thread quando o thread é iniciado e especificando o tamanho máximo da pilha para o thread.
Thread(ThreadStart)	Inicializa uma nova instância da classe Thread.
Thread(ThreadStart, Int32)	Inicializa uma nova instância da classe Thread, especificando o tamanho máximo da pilha do thread.
Propriedades	
ApartmentState	Obtém ou define o estado de apartment desse thread.
CurrentCulture	Obtém ou define a cultura do thread atual.
CurrentPrincipal	Obtém ou define a entidade de segurança atual do thread (para segurança baseada em função).
CurrentThread	Obtém o thread em execução no momento.
CurrentUICulture	Obtém ou define a cultura atual usada pelo Gerenciador de Recursos para procurar recursos específicos da cultura em tempo de execução.
ExecutionContext	Obtém um objeto ExecutionContext que contém informações sobre os diversos contextos do thread

	atual.
IsAlive	Obtém um valor que indica o status de execução do thread atual.
IsBackground	Obtém ou define um valor que indica se um thread é ou não um thread de segundo plano.
IsThreadPoolThread	Obtém um valor que indica se um thread pertence ao pool de threads gerenciados ou não.
ManagedThreadId	Obtém um identificador exclusivo para o thread gerenciado atual.
Name	Obtém ou define o nome do thread.
Priority	Obtém ou define um valor que indica a prioridade de agendamento de um thread.
ThreadState	Obtém um valor que contém os estados do thread atual.
Métodos	
Abort()	Gera um ThreadAbortException no thread no qual ele é invocado, para iniciar o processo de encerramento do thread. Geralmente, a chamada a esse método termina o thread.
Abort(Object)	Gera um ThreadAbortException no thread no qual ele é invocado, para iniciar o processo de término do thread e ao mesmo tempo fornecer informações de exceção sobre o término do thread. Geralmente, a chamada a esse método termina o thread.
AllocateDataSlot()	Aloca um slot de dados sem nome em todos os threads. Para melhorar o desempenho, use os campos marcados com o atributo ThreadStaticAttribute.
AllocateNameDatasSlot(String)	Aloca um slot de dados nomeado em todos os threads. Para melhorar o desempenho, use os campos marcados com o atributo ThreadStaticAttribute.
BeginCriticalRegion()	Notifica um host que a execução está prestes a entrar em uma região de código em que os efeitos de uma exceção sem tratamento ou anulação de thread podem comprometer outras tarefas no domínio do aplicativo.

BeginThreadAffinity()	Notifica um host de que o código gerenciado está prestes a executar instruções que dependem da identidade do thread atual do sistema operacional físico.
DisableComObjectEagerCleanup()	Desativa a limpeza automática de RCWs (Runtime Callable Wrappers) para o thread atual.
EndThreadCriticalRegion()	Notifica um host de que a execução está prestes a entrar em uma região de código na qual os efeitos de uma exceção sem tratamento ou anulação de thread estão limitados à tarefa atual.
EndThreadAffinity()	Notifica um host que o código gerenciado terminou de executar as instruções que dependem da identidade do thread do sistema operacional físico atual.
Equals(Object)	Determina se o objeto especificado é igual ao objeto atual. (Herdado de Object)
Finalize()	Garante que os recursos são liberados e outras operações de limpeza são executadas quando o coletor de lixo recupera o objeto Thread.
FreeNamedDataSlot(string)	Elimina a associação entre um nome e um slot em todos os threads do processo. Para melhorar o desempenho, use os campos marcados com o atributo ThreadStaticAttribute.
GetApartmentState()	Retorna um valor ApartmentState que indica o estado do apartment.
GetCompressedStack()	Retorna um objeto CompressedStack que pode ser usado para capturar a pilha do thread atual.
GetCurrentProcessorId()	Obtém uma ID usada para indicar em qual processador o thread atual está sendo executado.
GetData(LocalDataStoreSlot)	Recupera o valor do slot especificado no thread atual, no domínio atual do thread atual. Para melhorar o desempenho, use os campos marcados com o atributo ThreadStaticAttribute.
GetDomain()	Retorna o domínio atual no qual o thread atual está em execução.
GetDomainID()	Retorna o domínio atual no qual o thread atual está

	em execução.
GetHashCode()	Retorna um código hash para o thread atual.
GetNamedDataSlot(String)	Pesquisa um slot de dados nomeado. Para melhorar o desempenho, use os campos marcados com o atributo ThreadStaticAttribute.
GetType()	Obtém o Type da instância atual. (Herdado de Object)
Interrupt()	Interrompe um thread que está no estado de thread WaitSleepJoin.
Join()	Bloqueia o thread de chamada até que o thread representado por essa instância termine, enquanto continua a executar COM padrão e o bombeamento de SendMessage.
Join(Int32)	Bloqueia o thread de chamada até que o thread representado por essa instância termine ou até que o tempo especificado tenha decorrido, enquanto continua executando o COM padrão e o bombeamento de SendMessage.
Join(TimeSpan)	Bloqueia o thread de chamada até que o thread representado por essa instância termine ou até que o tempo especificado tenha decorrido, enquanto continua executando o COM padrão e o bombeamento de SendMessage.
MemberwiseClone()	Cria uma cópia superficial do Object atual. (Herdado de Object)
MemoryBarrier()	Sincroniza o acesso à memória da seguinte maneira: o processador que executa o thread atual não pode reorganizar as instruções de forma que os acessos à memória antes da chamada a MemoryBarrier() sejam executados após os acessos de memória que seguem a chamada a MemoryBarrier().
ResetAbort()	Cancela um Abort(Object) solicitado para o thread atual.
Resume()	Retoma um thread que foi suspenso.
SetApartmentState(ApartmentState)	Define o estado do apartment de um thread antes

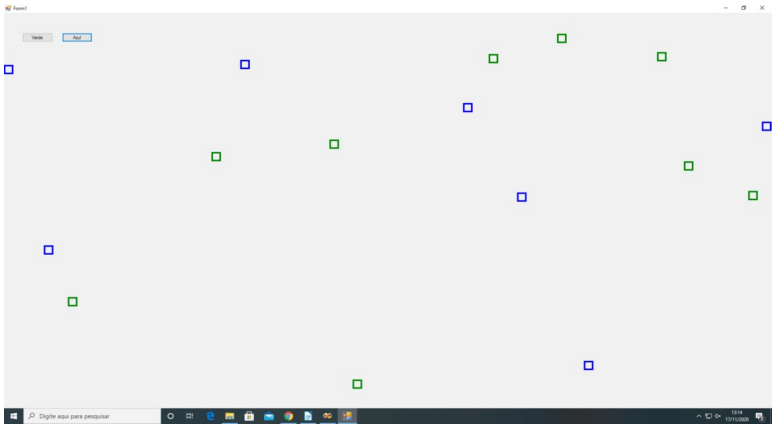
	que ele seja iniciado.
SetCompressedStack(CompressedStack)	Aplica uma CompressedStack capturada ao thread atual.
SetData(LocalDataStoreSlot, Object)	Define os dados no slot especificado no thread em execução no momento, para o domínio atual do thread. Para melhorar o desempenho, use os campos marcados com o atributo ThreadStaticAttribute.
Sleep(Int32)	Suspende o thread atual no número especificado de milissegundos.
Sleep(TimeSpan)	Suspende o thread atual para o período de tempo especificado.
SpinWait(Int32)	Faz com que um thread aguarde o número de vezes definido pelo parâmetro iterations.
Start()	Faz com que o sistema operacional altere o estado da instância atual para Running.
Start(Object)	Faz com que o sistema operacional altere o estado da instância atual para Running e, opcionalmente, fornece um objeto que contém dados a serem usados pelo método executado pelo thread.
Suspend()	Suspende o thread ou, se o thread já está suspenso, não tem efeito.
ToString()	Retorna uma cadeia de caracteres que representa o objeto atual. (Herdado de Object)
TrySetApartmentState(ApartmentState)	Define o estado do apartment de um thread antes que ele seja iniciado.
Yield()	Faz com que o thread de chamada conceda a execução para outro thread que está pronto para ser executado no processador atual. O sistema operacional seleciona o thread de recebimento.

Referências estas retiradas do próprio site da Microsoft para um maior compreendimento de algumas funções, algumas mais utilizadas e outras de uso mais específico.

Código Exemplo

```
namespace MultiThread2
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        Thread timer;
        Thread timer2;
        Random rdm;

        private void button1_Click(object sender, EventArgs e)
        {
            timer = new Thread(thread);
            timer.Start();
        }

        public void thread()
        {
            for (int i = 0; i < 100; i++)
            {
                this.CreateGraphics().DrawRectangle(new Pen(Brushes.Green, 4), new
Rectangle(rdm.Next(0, this.Width), rdm.Next(0, this.Height), 20, 20));
                Thread.Sleep(1000);
            }
        }
        public void thread2()
        {
            for (int i = 0; i < 100; i++)
            {
                this.CreateGraphics().DrawRectangle(new Pen(Brushes.Blue, 4), new
Rectangle(rdm.Next(0, this.Width), rdm.Next(0, this.Height), 20, 20));
                Thread.Sleep(1000);
            }
        }

        private void button2_Click(object sender, EventArgs e)
        {
            timer2 = new Thread(thread2);
            timer2.Start();
        }

        private void Form1_Load(object sender, EventArgs e)
        {
            rdm = new Random();
        }
    }
}
```

Esse código através de duas threads geram quadrados de maneira intercalada sem correr risco de perder performance.



3. INTERFACES

Uma interface define um contrato, ou seja, a classe “pai” definirá quais métodos deverão ser obrigatoriamente implementados nas classes “filhas” que utilizem esta interface. As interfaces são usadas justamente pelo fato de classes não poderem ter dois “pais” mas ainda sim devido a nossa aplicação necessitamos que certos métodos da classe “pai” sejam implementados, contornamos essa restrição usando contratos.

Prática:

Declarar uma interface é praticamente igual a declarar uma classe porém no lugar de "class" usamos a palavra-chave “interface”. E dentro dela colocamos apenas a assinatura dos métodos.

Exemplo:

```
public interface INomeDaInterface
{
    //código aqui
}
```

Quando devo usar uma interface?

- Quando houver a necessidade de implementar uma funcionalidade comum para classes não relacionadas,
- Agrupar objetos como características em comum (Ex: Criar vários objetos “pessoa” e todas elas devem ter como atributos como “cidade de nascimento”, “data de nascimento”, “nome” etc).
- Comportamento polimórfico as classes (Ex: métodos que possuem mesma assinatura porém argumentos diferentes).
- Componentes que são independentes

Exemplo uso de interface:

Suponhamos que eu tenha um programa que permite a entrada de várias formas geométricas e devemos retornar sua área. Para isso criamos primeiramente uma interface.

```
public interface IArea
{
    double CalcularArea();
}
```

Agora vamos definir duas classes a Retângulo e o Círculo.

```
public class Retangulo : IArea
{
	private double base;
	private double altura;

	public Retangulo(double base, double altura)
	{
		this.base = base;
		this.altura = altura;
	}

	public double CalculaArea()
	{
		return base*altura;
	}
}
```

```
public class Circulo : IArea
{
	private double raio;

	public Circulo(double raio)
	{
		this.raio = raio;
	}

	public double CalculaArea()
	{
		return Math.PI*Math.Pow(raio, 2.0);
	}
}
```

Vemos que cada classe tem sua própria definição de área, apesar delas serem figuram geométricas diferentes.

```
private static void Main(string[] args)
{
	var formasGeometricas = new List<IArea>() { new Retangulo(2, 3), new Circulo(4) };
    CalcularArea(formasGeometricas);

    Console.ReadKey();
}
private static void CalculaArea(IEnumerable<IAreas> formasGeometricas)
{
    foreach(forma in formasGeometricas)
        Console.WriteLine("Area: {0}",forma.CalculaArea());
}
```

4. Delegate

O que é?

Delegate pode ser entendido como um substituto de algum(uns) método(s), ele faz referência a um método específico ou seja poder ser entendido como um ponteiro de métodos.

As classes delegate oferecem uma interface pública que permite inicializar, adicionar, remover e invocar delegates.

Quando usar?

Se houver a necessidade de enviar um método como parâmetro de um outro método.

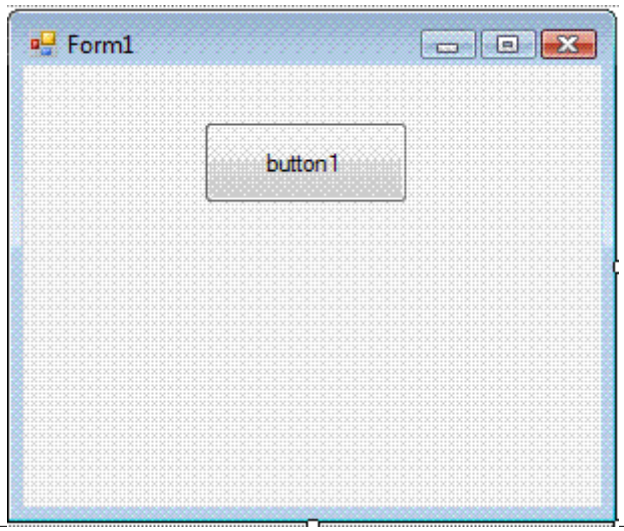
Exemplo 1:

```
namespace UmDelegateSimples
{
    public delegate void SimplesDelegate(); //declarando um delegate

    class ExemploDeDelegate
    {
        public static void minhaFuncao()
        {
            Console.WriteLine("Eu fui chamada por um delegate ...");
        }

        public static void Main()
        {
            SimplesDelegate simplesDelegate = new SimplesDelegate(minhaFuncao);
            //instanciando um delegate

            simplesDelegate(); //invocando indiretamente um delegate
            sismplesDelegate.Invoke(); //invocando diretamente um delegate
            Console.ReadKey();
        }
    }
}
```

Delegate e Eventos

Exemplo:

Quando criamos um controle, por exemplo um botão numa aplicação Windows Forms, automaticamente criamos o evento Click que ficará associado a ele.

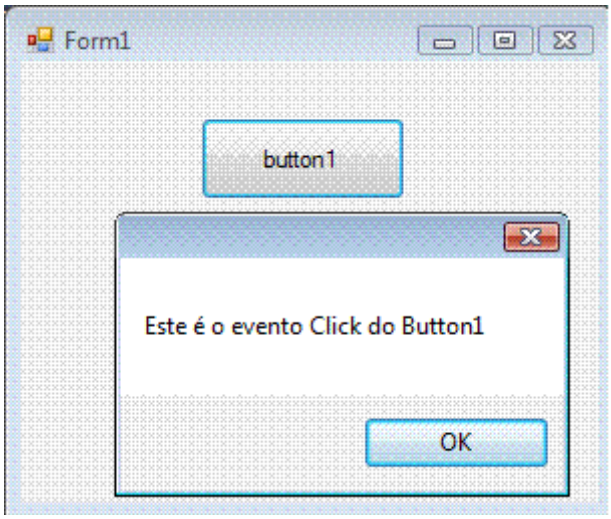



```
//
// button1
//
this.button1.Location = new System.Drawing.Point(90, 28);
this.button1.Name = "button1";
this.button1.Size = new System.Drawing.Size(102, 41);
this.button1.TabIndex = 0;
this.button1.Text = "button1";
this.button1.UseVisualStyleBackColor = true;
this.button1.Click += new System.EventHandler(this.button1_Click);
```

No código principal colocamos:

```
using System.Windows.Forms;

namespace C_Delegates_Events
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void button1_Click(object sender, EventArgs e)
        {
            MessageBox.Show("Este é o evento Click do Button1");
        }

        public void Mensagem(object sender, System.EventArgs e)
        {
            MessageBox.Show("Funciona  - método Mensagem");
        }
    }
}
```

E quando clicarmos no botão será exibido:



Se alterarmos a definição do evento click, colocando:

```
this.button1.Click += new System.EventHandler(Mensagem);
```

Ao colocar o operador += adicionamos o método Mensagem à lista de invocação.

Métodos Anônimos

Métodos anônimos são métodos que não possuem nome e são definidos através de um delegate e não estão ligados a uma função ou um método.

Exemplo:

```
private delegate void exemploDelegate();
private exemploDelegate objInstancia = delegate;
{
      Console.WriteLine("Exemplo de método anônimo");
}
```

Sua principal vantagem é poder ser executado de qualquer parte do programa e um objeto instância pode assumir outro endereço no decorrer da execução e também poder ser passado como parâmetro para outro método.

Quando um método anônimo não tem assinatura, por definição não obedece a nenhum delegate, mas também não pode acessar os parâmetros do delegate.

Exemplo 2 – Soma e multiplicação:

```
delegate double Calculo(double numero1, double numero2);
public static class Calculadora
{
	public static double Soma(double numero1, double numero2) => numero1 +
numero2;

	public static double Multiplicacao(doule numero1, double numero2) =>
numero1*numero2;
}
```

Agora vamos chamar o delegate, e para isso temos podemos usar ou não a palavra reservada "invoke".

```
double resultado = Soma.Invoke(2, 2)
double resultado = Multiplica(2, 3);
```

5. LINQ e Lambda

Consultas LINQ:

As consultas são, de forma rápida e objetiva, maneiras de recuperar dados de uma fonte de dados e são feitas em linguagens específicas, como SQL, Xquery, etc. O LINQ, surgiu com o intuito de simplificar isso, oferecendo uma maneira de trabalhar com vários tipos e formatos de banco de dados, para usar uma consulta LINQ, você precisa trabalhar com objetos, usando o mesmo padrão básico de codificação para consultar e transformar dados de documentos XML, de bancos de dados SQL, coleções .NET, etc.

Expressões Lambda:

Uma expressão lambda, basicamente, é uma função sem nome, podendo realizar cálculos, filtros e retornar valores. Sua representação é feita pelo uso do operador Lambda "=>", onde à esquerda do operador são colocados os parâmetros e a direita as expressões ou instruções.

Utilizando Consultas LINQ e Expressões Lambda:

O código abaixo foi escrito com a finalidade de filtrar todos os valores maiores do que 100 de uma lista, para isso podemos usar um foreach, mas também é possível o uso de consultas LINQ e Expressões Lambda:

```
var lista = new List<int>();
lista.Add(…);
//Adiciona valores para a lista
var filtrados = new List<int>();
foreach(int valor in lista)
{
    if(valor>100)
    {
        filtrados.Add(valor);
    }
}
```

Aqui temos um exemplo do código sem utilizar as consultas LINQ e as Expressões Lambda.

Agora, o código abaixo faz exatamente a mesma coisa que o anterior, só que utilizando LINQ e Lambda:

```
var lista = new List<int>();
lista.Add(…);
//Adiciona valores para a lista
var filtradas = lista.Where((int valor) => { return valor > 100; });
```

Neste código, temos uma lista de inteiros sendo analisada, cuja análise é: onde na lista tiver um valor acima de 100, retornar o valor.

Mas é possível simplificar ainda mais o código, pois as expressões Lambda, se possuírem apenas uma linha, podem ser simplificadas, como no exemplo abaixo:

```
var lista = new List<int>();
lista.Add(…);
//Adiciona valores para a lista
var filtradas = lista.Where(valor => valor > 100);
```

Consultas LINQ mais comuns:

Além da consulta “Where”, podemos utilizar várias outras consultas LINQ, como por exemplo:

- Sum

Serve basicamente para somar os elementos retornados:

```
var soma = lista.Sum(valor => valor);
```

- Average

Serve para calcular a média dos elementos retornados:

```
var media = lista.Average(valor => valor);
```

- Count

Serve para contar quantos elementos foram retornados:

```
var quantidade = lista.Count(valor => valor);
```

- Min

Retorna o menor valor:

```
var menor = lista.Min(valor => valor);
```

- Max

Retorna o maior valor:

```
var maior = lista.Max(valor => valor);
```

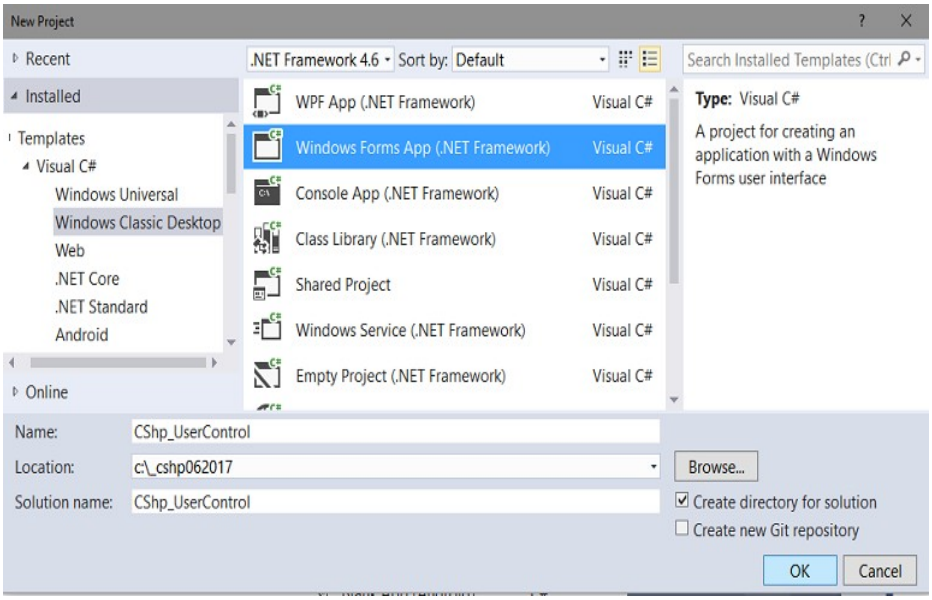
Com isso, percebemos que é possível simplificar bastante o código com o uso de métodos LINQ (Language Integrated Query), por padrão, a biblioteca LINQ já está inclusa em projetos do Visual Studio, mas caso não esteja, é possível adicionar a biblioteca manualmente, adicionando o "`using System.Linq;`" no inicio do seu programa.

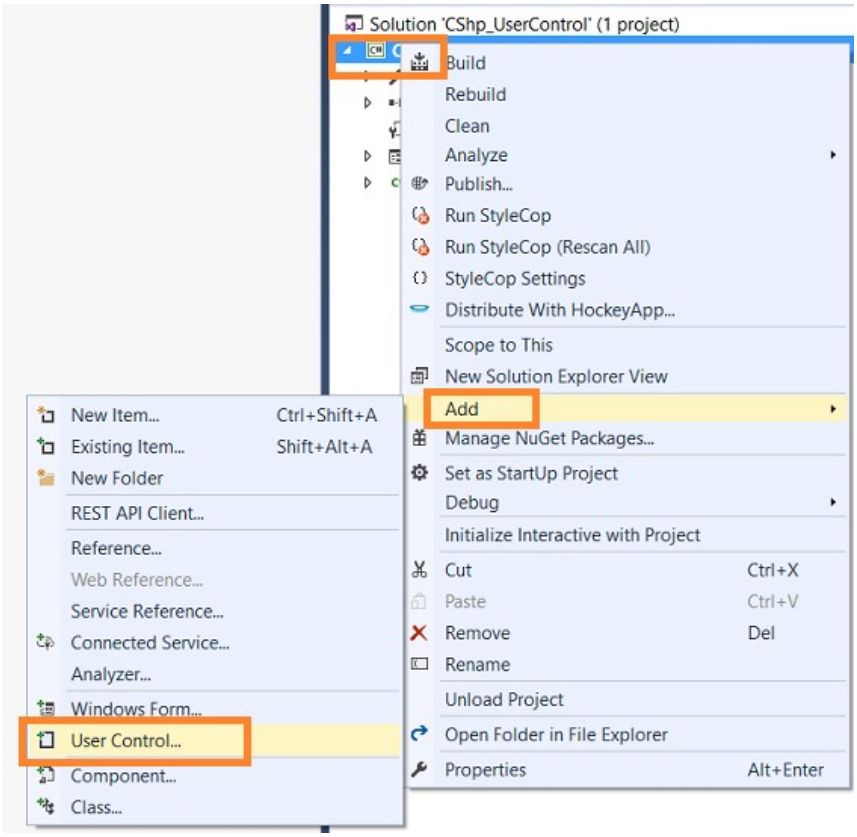
6. User Control

Ao desenvolver um projeto em Windows Forms, muitas vezes nos deparamos com padrões de designers. Para auxiliar e evitar excessivas repetições, o User Control veio para ajudar o desenvolvedor, servindo como uma ferramenta que cria um novo componente, que pode ser facilmente acessado na ToolBox.

Isso ajuda na repetição excessiva de códigos e interface gráfica.

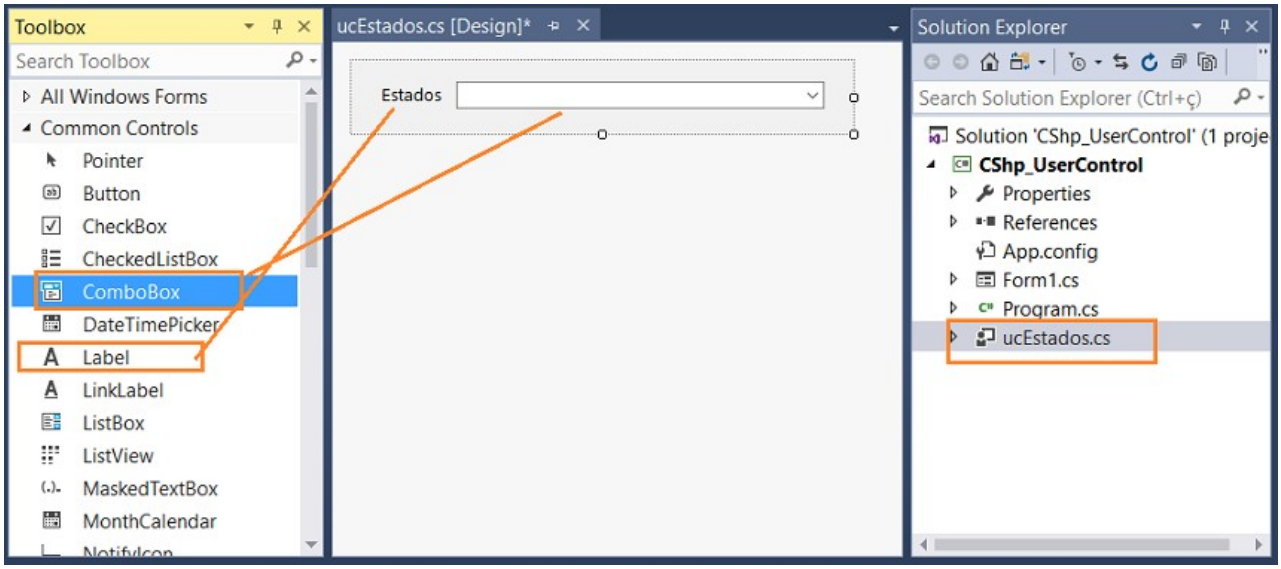
Aqui está um passo a passo de como utilizá-lo:





Após criar um User Control, podemos construí-lo e usá-lo como um Forms. Ao desenvolver o projeto, o C# cria um componente na ToolBox exatamente igual ao que você programou, com o nome do UserControl.

O exemplo a seguir exemplifica uma utilização básica dele:



Respectivamente, o UC possui lblEstados e cboEstados.

Adicione o seguinte código em uma classe Estado

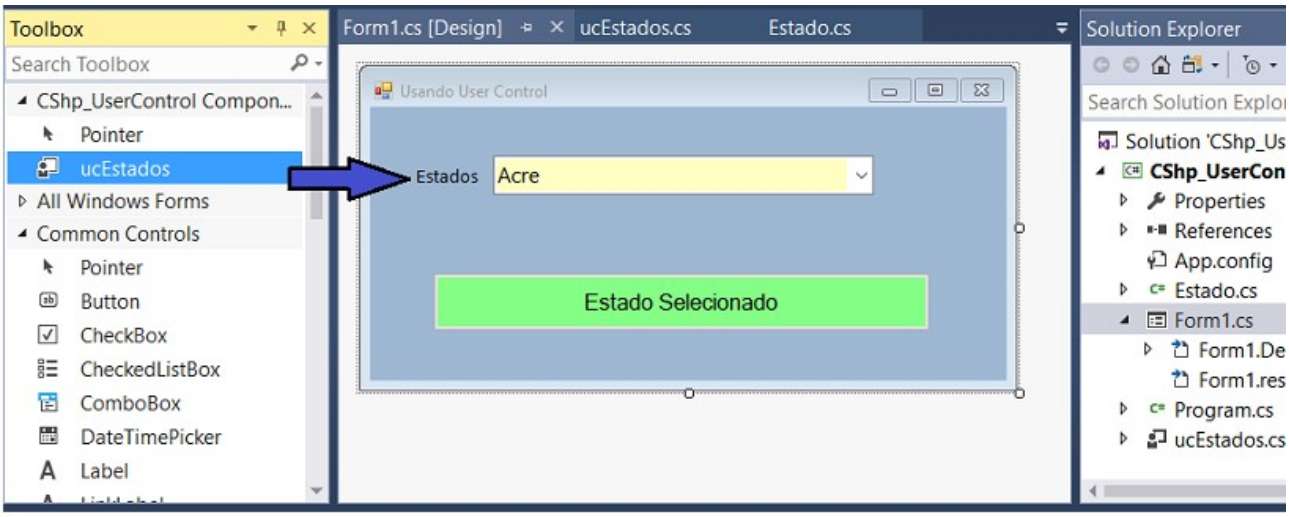
```
namespace CShp_UserControl
{
    public class Estado
    {
        public int Id { get; set; }
        public string Nome { get; set; }
    }
}
```

Agora insira o seguinte código no UserControl

```
using System;
using System.Collections.Generic;
using System.Windows.Forms;

namespace CShp_UserControl
{
   public partial class ucEstados : UserControl
   {
      public ucEstados()
      {
         InitializeComponent();
      }

      public Estado estadoSelecionado
      {
         get
         {
            return (Estado)cboEstados.SelectedItem;
         }
      }

      private void ucEstados_Load(object sender, EventArgs e)
      {
         List<Estado> estados = new List<Estado>();
         estados.Add(new Estado() { Id = 1, Nome = "Acre" });
         estados.Add(new Estado() { Id = 2, Nome = "Alagoas" });
         estados.Add(new Estado() { Id = 3, Nome = "Amapá" });
         estados.Add(new Estado() { Id = 4, Nome = "Amazonas" });
         estados.Add(new Estado() { Id = 5, Nome = "Bahia" });
         estados.Add(new Estado() { Id = 6, Nome = "Ceará" });
         estados.Add(new Estado() { Id = 7, Nome = "Distrito Federal" });
         estados.Add(new Estado() { Id = 8, Nome = "Espírito Santo" });
         estados.Add(new Estado() { Id = 9, Nome = "Goiás" });
         estados.Add(new Estado() { Id = 10, Nome = "Maranhão" });
         cboEstados.DataSource = estados;
         cboEstados.ValueMember = "Id";
         cboEstados.DisplayMember = "Nome";
      }
   }
}
```

Este código basicamente cria uma lista de objetos Estado, que é declarada na linha de código. Os elementos da ComboBox são selecionados a partir de uma propriedade estadoSelecionado dentro do User control.

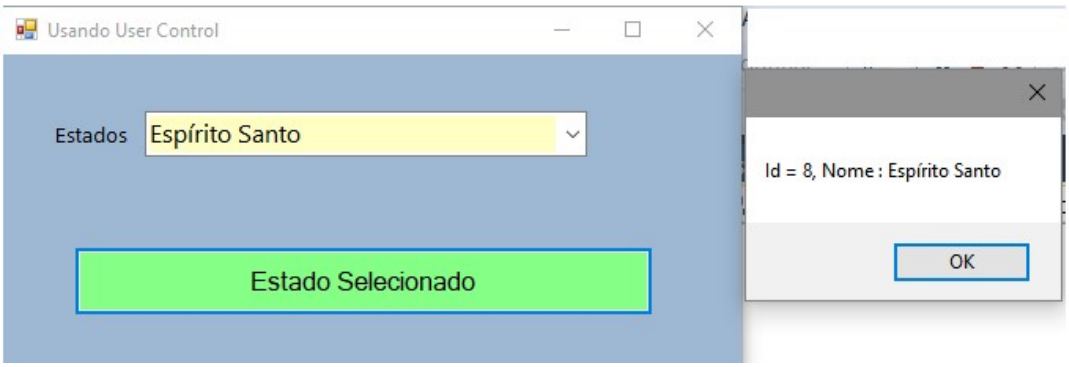
Após criado, podemos aplicá-lo em um Forms, e utilizá-lo.



Agora defina o seguinte código no evento **Click** do botão de comando:

```
 private void btnEstadoSelecionado_Click(object sender, EventArgs e)
  {
        MessageBox.Show(string.Format("Id = {0}, Nome : {1}",
ucEstados1.estadoSelecionado.Id.ToString(), ucEstados1.estadoSelecionado.Nome.ToString()));
  }
```

Executando o projeto iremos obter o seguinte resultado:



Na linha de código, ele simplesmente chama a ucEstado1, a propriedade da ucEstado1 estadoSelecionado, que retornará um objeto Estado. Depois disso, ele faz uma lógica para aparecer os atributos deste objeto, como mostrado na última imagem.

7. DLLs

O que é uma DLL:

Uma DLL (Dynamic Link Library) é uma biblioteca dinâmica que contém dados que podem ser acessados por mais de um programa, e até mesmo podendo ser utilizada ao mesmo tempo por vários programas, elas são basicamente sub-rotinas que executam determinadas ações quando algum programa requisita o seu acesso.

Vantagens das DLLs:

DLLs podem conter diversas coisas, como dados, códigos, recursos, etc. Umas das vantagens de utilizar uma DLL, é quando é necessário atualizar um programa, mas determinada rotina não é alterada, então, é possível criar DLLs responsáveis por essas rotinas, e deixando a atualização mais compacta e fácil de ser feita.

Outra vantagem de utilizar DLLs está na economia de recursos, pois quando há mais de um programa utilizando a mesma biblioteca de funções, o uso das DLLs impede a duplicação do código que será carregado no disco, reduzindo o uso de memória, o que consequentemente melhora o desempenho de aplicativos tanto em primeiro, quanto em segundo plano.

Problemas com as DLLs:

Alguns problemas com o uso das DLLs são quando:

- A DLL é atualizada para uma versão incompatível com algum programa;
- A DLL sofre modificações;
- Uma DLL é sobrescrita com uma versão mais antiga;
- A DLL compartilhada é removida por algum motivo;

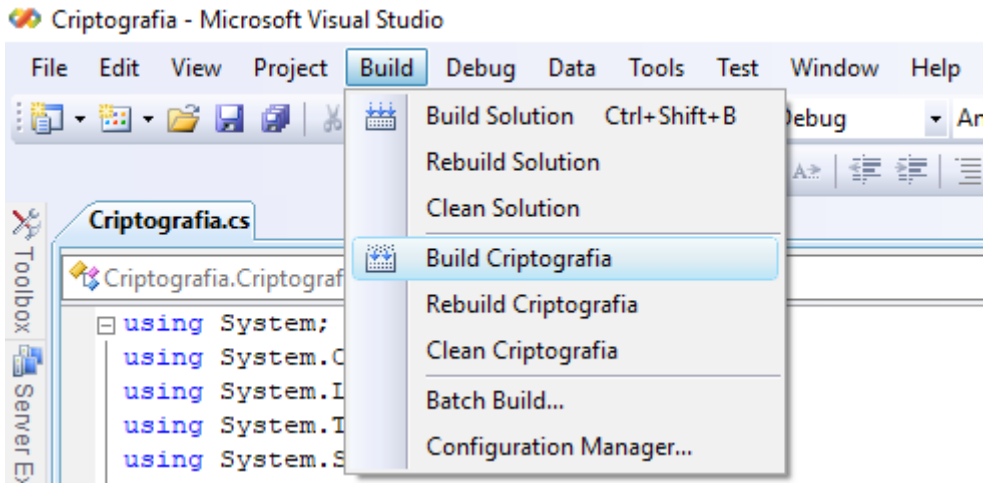
Isso pode gerar conflitos, fazendo com que alguns aplicativos apresentem mal funcionamento ou parem de funcionar.

Como criar DLLs no C#:

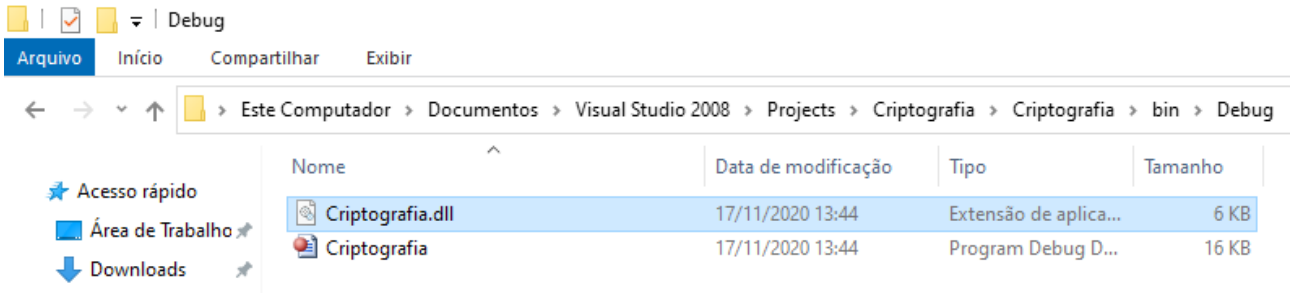
Criar DLLs no C# é um processo relativamente simples, onde basicamente é necessário criar uma classe, com as ações que você deseja que sua DLL faça, como exemplo, será utilizado uma classe para criptografia de dados.

Primeiramente crie uma classe e coloque o código que você quer que sua DLL seja responsável por executar.

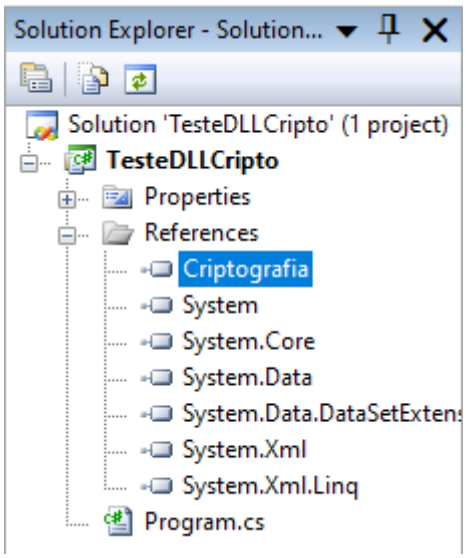
Após isso, vá no menu do Visual Studio → *Build* → *Build Nome-da-sua-classe*, como no exemplo a seguir:



Pronto, sua DLL está criada e localizada na pasta *debug* da sua solução:



Prontinho, agora para utilizar sua DLL em um projeto, basta adicioná-la nas referências, e Escrever seu nome para usar as funcionalidades.




```
Criptografia.Criptografia.GenerateKeys();
foreach (byte a in Criptografia.Criptografia.Key)
{
    Console.WriteLine(a.ToString());
}
Console.ReadKey();
```

8. Serialização e Desserialização

Dados JSON e XML são formatos comum passado entre aplicações hoje em dia. Serialização é o processo de conversão de objetos .NET em Strings no formato JSON ou XML e Desserializar é o processo contrário em que um dado JSON OU XML são convertidos para objetos .NET.

JSON tem os seguintes estilos:

1. Objeto

Um assembly "name / value" não ordenado. Um objeto começa com "{" e termina com "}". Atrás de cada "nome", há dois pontos. E a vírgula é usada para separar muito "nome / valor". Por exemplo:

```
var user = { "name" : "Manas" , "gender" : "Male" , "birthday" : "1987-8-8" }
```

2. Array

Ordem de valor definida. Uma matriz começa com "[" e termina com "]". E os valores são separados por vírgulas. Por exemplo:

```
var userlist = [{"user":{"name":"Manas","gender":"Male","birthday":"1987-8-8"}},
{"user":{"name":"Mohapatra","Male":"Female","birthday":"1987-7-7"}}]
```

3. String

Qualquer quantidade de conjunto de caracteres Unicode que esteja entre aspas. Ele usa barra invertida para escapar

```
var userlist = "{\"ID\":1,\"Name\":\"Manas\",\"Address\":\"India\"}"
```

A Serialização e Desserialização JSON pode ser implementado de três formas:

- Usando a classe JavaScriptSerializer
- Usando a classe DataContractJsonSerializer
- Ussando a biblioteca JSON.NET

Usando a classe DataContractJsonSerializer

A classe DataContractJsonSerializer possibilita a serialização e desserialização JSON. Está presente no namespace System.Runtime.Serialization.Json. Usando a classe, podemos serializar um objeto em dados JSON e desserializar os dados JSON em um objeto atráves de contratos de dados equivalentes.

Classe de exemplo:

```
[DataContract]
class Pessoa
{
	[DataMember]
	public string Name { get; set; }

	[DataMember]
	public string Description { get; set; }
}
```

Serialização

Na serialização, ele converte um objeto .NET em uma String JSON. Na serialização, ele converte um objeto .Net personalizado em uma string JSON. No código a seguir, ele cria uma instância de Pessoa e atribui valores a suas propriedades. Em seguida, criamos uma instância da classe DataContractJsonSerializer passando o parâmetro a classe Pessoa e criamos uma instância da classe MemoryStream para escrever o objeto (Pessoa). Por último, ele cria uma instância da classe StreamReader para ler dados JSON do objeto MemorySteam.

```
Pessoa psObj = new Pessoa()
{
	Name = "Matheus",
	Description = "Estudante"
};

DataContractJsonSerializer js = new DataContractJsonSerializer(typeof(Pessoa));
MemoryStream msObj = new MemoryStream();
js.WriteObject(msObj, psObj);
msObj.Position = 0;
StreamReader sr = new StreamReader(msObj);

// "{\"Description\":\"Estudante\",\"Name\":\"Matheus\"}"
string json = sr.ReadToEnd();

sr.Close();
msObj.Close();
```

Desserialização

Na desserialização, ele faz o oposto da serialização, o que significa que converte a string JSON em um objeto .Net personalizado. No código a seguir, ele cria uma instância da classe Pessoa e atribui valores a suas propriedades. Em seguida, criamos uma instância da classe DataContractJsonSerializer passando o parâmetro classe Pessoa e criando uma instância da classe MemoryStream para escrever um objeto (Pessoa). Por último, ele cria uma instância da classe StreamReader para ler dados JSON do objeto MemorySteam.

```
string json = "{\"Description\":\"Estudante\",\"Name\":\"Matheus\"}";

using (var ms = new MemoryStream(Encoding.Unicode.GetBytes(json)))
{
      // Deserialization from JSON
      DataContractJsonSerializer deserializer = new
      DataContractJsonSerializer(typeof(Pessoa));
      Pessoa psObj2 = (Pessoa)deserializer.ReadObject(ms);
      Response.Write("Name: " + psObj2.Name); // Name: Matheus
      Response.Write("Description: " +
      psObj2.Description); // Description: Estudante
}
```

Usando JavaScriptJsonSerializer

Ele está presente no namespace System.Web.Script.Serialization. Para serializar um objeto .Net para uma string JSON, use o método Serialize. É possível desserializar a string JSON para o objeto .Net usando os métodos Deserialize <T> ou DeserializeObject.

Classe exemplo:

```
class Pessoa
{
      public string Name { get; set; }
      public string Description { get; set; }
}
```

Serialização

Cria uma instância de JavaScriptSerializer e chamamos o método Serialize () passando o objeto (Pessoa). Ele retorna dados JSON em formato de string.

```
// Criando objeto Pessoa

Pessoa psObj = new Pessoa()
{
      Name = "Matheus",
      Description = "Estudante"
};

// Serializando objeto para JSON
JavaScriptSerializer js = new JavaScriptSerializer();
string jsonData = js.Serialize(psObj);
//{"Name":"Matheus","Description":"Estudante"}
```

Desserialização

Cria uma instância JavaScriptSerializer e chama Deserialize () passando dados JSON. Ele retorna um objeto personalizado (Pessoa) de dados JSON.

```
// Desserializando dados json para o objeto
JavaScriptSerializer js = new JavaScriptSerializer();
Pessoa pessoaObject = js.Deserialize<Pessoa>(jsonData);
string name = pessoaObject.Name;
string description = pessoaObject.Description;

// Outra maneira de sem ajuda da classe Pessoa
dynamic pessoaObject = js.Deserialize<dynamic>(jsonData);
string name = pessoaObject["Name"];
string description = pessoaObject["Description"];
```

Usando Json.Net

Json.NET é uma biblioteca de terceiros que ajuda a conversão entre texto JSON e objeto .NET usando o JsonSerializer.

Obs: Suporta .NET 2, .NET 3.5, .NET 4, Silverlight and Windows Phone.

Serialização

chama o método estático SerializeObject () da classe JsonConvert passando o objeto (Pessoa). Ele retorna dados JSON em formato de string.

```
// Criando objeto Pessoa
Pessoa psObj = new Pessoa()
{
      Name = "Matheus",
      Description = "Estudante"
};

// Converter o objeto Pessoa para o formato de string JOSN
string jsonData = JsonConvert.SerializeObject(psObj);

Response.Write(jsonData);
```

Desserialização

Chama o método estático DeserializeObject () da classe JsonConvert passando dados JSON. Ele retorna um objeto personalizado (Pessoa) de dados JSON.

XML em .NET

Usando o namespace System.Xml.Serialization

Classe exemplo, usando contrato:

```
[DataContract()]
public class Pessoa
{

      public Pessoa(string nome, string sobrenome, DateTime nascimento)
      {
            Nome = nome;
            Sobrenome = sobrenome;
            Nascimento = nascimento;
      }
      [DataMember()]
      public string Nome { get; set; }
      [DataMember()]
      public string Sobrenome { get; set; }
      [DataMember()]
      public DateTime Nascimento { get; set; }
}
```

Classe onde acontece a serialização e desserialização:

```
public class XmlHelper
{

        public static string Serializar<T>(T obj)
      {
            DataContractSerializer serializer = new
DataContractSerializer(typeof(T));
            MemoryStream ms = new MemoryStream();
            serializer.WriteObject(ms, obj);
            return Encoding.UTF8.GetString(ms.ToArray());
      }

      public static T DeSerializar<T>(string json)
      {
            DataContractSerializer ser = new DataContractSerializer(typeof(T));
            MemoryStream ms = new MemoryStream(Encoding.UTF8.GetBytes(json));
            return (T)ser.ReadObject(ms);
      }
}
```

Formulário principal:

```
Pessoa p;
private void btn_Serializar_Click(object sender, EventArgs e)
{
      try
      {
            txt_Exibir.Text
            Serializar.XmlHelper.Serializar<Pessoa>(lista[i]);
      }

      catch { }
}


private void btn_Desserializar_Click(object sender, EventArgs e)
{
      try
      {
            propertyGrid1.SelectedObject =
            Serializar.XmlHelper.DeSerializar<Pessoa>(txt_Exibir.Text);
      }

      catch { }
}
```

9. SQL

A sigla SQL significa “Structured Query Language”, em português é traduzida como “Linguagem Estrutura de Dados”, trata-se de uma linguagem de programação, voltada para gerenciamento de dados em SGBDs (Sistemas de Gerenciamento de Bancos de Dados).

Para um melhor entendimento dessa linguagem deve-se entender o que seria um “Banco de Dados”, ele é basicamente uma fonte onde se registra informações de vários tipos, normalmente são constituídos de colunas e linhas para armazenar cada dado em seu local pré-determinado, além disso esses dados podem ou não ser relacionados e únicos, ou seja, um dado pode ou não ter ligação com outro.

Um exemplo disso seria de, uma pessoa qualquer é definida pelo seu CPF, ou seja, aquela pessoa tem um número que sempre irá a identificar para qualquer tipo de caso, por outro lado podemos ter uma pessoa que tenha uma certa idade, porém essa idade não é única e nem mesmo relaciona explicitamente essa pessoa, pois podem haver outras pessoas com a mesma idade que a que está sendo procurada.

Vamos agora entender como funciona os comandos da linguagem SQL, essa linguagem está ligada ao conceito de CRUD que significa Create, Read, Update, Delete, em português é traduzido como Criar, Ler, Atualizar e Deletar, onde esse conceito define que todo e qualquer dado pode ser criado, lido, atualizado e deletado na hora que for necessário.

Sabendo disso vamos ver como cada um funciona nessa linguagem:

- **CREATE**

Em SQL, esse comando pode ser dividido em dois comandos separados, mas que no final acabam criando algo no banco de dados, sendo eles o comando **CREATE** que é utilizado para se criar Banco de Dados ou Tabelas no SQL.

Para se criar um banco de dados o comando é: **CREATE DATABASE CLIENTES ,** já para as tabelas veremos abaixo como fazer.

Exemplo: **CREATE TABLE USUARIOS(ID INT, NOME VARCHAR(40))**

Esse comando cria uma tabela USUARIOS com as colunas ID do tipo inteiro e NOME do tipo varchar com 40 caracteres.

E o comando **INSERT** que consiste em inserir um novo dado em uma tabela, logo esse dado está sendo criado e armazenado na tabela, por isso se encaixa nesse conceito.

Exemplo: **INSERT INTO USUARIOS(ID, NOME) VALUES (1,’LUCAS’)**

Esse comando cria um novo dado na tabela USUARIOS com o ID igual a 1 e o NOME igual a LUCAS.

- **READ**

Em SQL, para se ler um ou mais dados de um banco de dados, se utiliza o comando **SELECT** que consiste em selecionar uma tabela e os dados, e ler os que foram especificados no corpo do comando.

Exemplo: S**ELECT * FROM USUARIOS**

Esse comando seleciona todos os dados da tabela USUARIOS, pois o carácter (*) implica em selecionar todos as colunas da tabela, caso deseja-se selecionar específicos bastaria colocar o nome da coluna a ser selecionada como: **SELECT ID FROM USUARIOS**, onde selecionaria só a coluna ID.

- **UPDATE**

Em SQL, para se atualizar um ou mais dados em um banco de dados, se utiliza o comando **UPDATE** que consiste em selecionar um ou mais dados que já estão armazenados e alterar os campos especificados no corpo do comando.

Exemplo: **UPDATE USUARIOS SET ID = ‘2’, NOME= ‘CLAUS’**

Esse comando atualizaria todos os dados da tabela para ID igual a 2 e NOME igual a CLAUS, para se atualizar dados específicos da tabela, deve-se usar o comando **WHERE** junto ao **UPDATE** para definir quais dados serão modificados.

Além disso também é possível se atualizar uma tabela do banco, utilizando o comando **ALTER**, é possível se alterar os campos da tabela com base no que for inserido no corpo do comando.

Exemplo: **ALTER TABLE USUARIOS ADD Email varchar(50)**

Esse comando adiciona na tabela USUARIOS uma nova coluna chamada E-mail do tipo varchar com 50 caracteres.

- **DELETE**

Em SQL, para se deletar um ou mais dados em um banco de dados, se utiliza o comando **DELETE** que consiste em selecionar um ou mais dados e os excluir do banco de dados, os dados a serem excluídos são especificados no corpo do comando.

Exemplo: **DELETE * FROM USUARIOS**

Esse comando deletaria todos os dados da tabela USUARIOS pois não foi especificado o dado a ser deletado, para isso deve-se usar o comando WHERE junto ao DELETE para definir quais dados serão deletados.

Além disso também é possível se deletar uma tabela do banco, utilizando o comando **DROP**, onde se especifica o nome da tabela a qual será deletada do banco de dados.

Exemplo: **DROP TABLE USUARIOS**

Esse comando deleta a tabela USUARIOS do banco de dados.

Os comandos citados em cada uma das siglas de CRUD, constituem os principais comandos de SQL, porém existem outros comandos que auxiliam cada um deles para ser possível manipular como quiser cada dado ou tabela do banco de dados, alguns exemplos são:

- **FROM:** Utilizado para definir de qual tabela os dados serão selecionados.
  Exemplo: **SELECT ID, NOME FROM USUARIOS**
  Nesse exemplo estamos selecionando os dados ID e Nome da tabela Usuários pelo comando FROM.
- **JOIN:** Existem diversos tipos de JOIN, mas, em geral, seu efeito é de juntar duas ou mais tabelas em apenas uma para relacionar cada uma e mostrar seus vínculos entre si.
  Exemplo: **SELECT ID FROM Finança as A JOIN Clientes as B on a.ID = b.ID WHERE B.CPF = X**
  Nesse exemplo estamos selecionando um ID da tabela finanças e porém estamos buscando esse ID por um CPF que está na tabela Clientes logo precisamos juntas as duas para procurar esse dado.
  O comando **as** faz com que demos um apelido para uma tabela para ser mais rápido escrever no código e o deixá-lo mais simples porém não é necessário.
- **WHERE:** Define quais os dados serão selecionados com base em uma condição.
- **INTO:** Completa o comando INSERT porém não é necessário especificar esse comando para se inserir na tabela mas para melhorar a abstração humana ele é útil.
- **ORDER BY:** Ordena os valores a serem selecionados com base no dado a ser mandado por esse comando.
  Exemplo: **SELECT ID, NOME FROM USUARIOS ORDER BY ID**
  Nesse exemplo estamos selecionando todos os IDs e Nomes de uma tabela de usuários e exibindo na tela na ordem de ID, ou seja, do primeiro para o último ID.
- **GROUP BY:** Agrupa valores repetidos em um único valor para melhor identificação do que está acontecendo.
  Exemplo: **SELECT COUNT(Compras), Cliente FROM Usuarios GROUP BY Cliente**
  Nesse exemplo estamos agrupando e contando cada compra de cada cliente para saber quanto cada cliente comprou.
- **LIKE:** Define o dado a ser procurado por carácter especificado em vez do conjunto completo de caracteres.
  Exemplo: **SELECT * FROM USUARIOS WHERE NOME LIKE '%a%'**
  Esse comando procura no banco de dados todos os nomes que tenham a letra a e os seleciona.
- **SET:** Define um valor para um dado.

Exemplo: **UPDATE USUARIOS SET NOME=’Guilherme’**

Esse comando define o nome para guilherme para todos os dados nessa tabela.

Existem diversos outros comandos em SQL para as mais diversas situações, então dependendo da situação que se encontra pode ser que exista um comando que resolva o seu problema, mas caso não exista, ai seria necessário desenvolver uma lógica para resolver a situação.

Os comandos estudados acima foram todos sobre realizar ações no banco de dados, porém existem outros comandos que mudam algumas propriedades do banco de dados, como por exemplo:

- Comandos de Permissão:

  São comandos que implicam nas permissões dos usuários.

GRANT: Faz com que você permita um usuário acessar e modificar as informações do banco de dados.

REVOKE: Faz com que você proíba ou impeça o usuário de acessar e modificar as informações do banco de dados.

- Comandos de Salvamento:

São comandos que quando ativados podem garantir maior segurança ao seu banco de dados, pois adicionam um nível de proteção para caso se execute uma ação errada no banco, podendo assim reverter o que foi feito.

COMMIT: Salva todas as alterações no banco de dados

ROLLBACK: Restaura o banco de dados para o último **COMMIT** feito.

Código Exemplo

Vamos criar um banco de dados do zero, realizar algumas alterações e após tudo isso excluir tudo que foi feito.

```
--Criando um banco de dados com nome de CLIENTES
CREATE DATABASE CLIENTES

USE CLIENTES

--Criando uma tabela com nome de DADOS
CREATE TABLE DADOS(
ID INT PRIMARY KEY IDENTITY,
NOME VARCHAR(40),
EMAIL VARCHAR(50),
IDADE INT
)
```

Com isso criamos o banco de dados CLIENTES, estamos usando ele por causa do USE e após isso criamos uma tabela chamada DADOS, porém acabamos criando com E-mail e na verdade queríamos ter criado com CPF, em vez de recriar a tabela podemos alterá-la.

```
--Modificando a tabela
ALTER TABLE DADOS DROP COLUMN EMAIL

ALTER TABLE DADOS ADD CPF VARCHAR(11)
```

Com isso excluímos a coluna E-mail da tabela e inserimos a coluna CPF.

```
--Inserindo um novo dado na tabela
INSERT INTO DADOS(NOME,IDADE,CPF) VALUES ('Lucas','22','12345678900')
INSERT INTO DADOS(NOME,IDADE,CPF) VALUES ('Claus','19','00987654321')

--Selecionando todos os dados da tabela DADOS
SELECT * FROM DADOS
```

	ID	NOME	IDADE	CPF
1	1	Lucas	22	12345678900
2	2	Claus	19	00987654321

Utilizando o INSERT e o SELECT, colocamos os novos dados na tabela e estamos os visualizando em video.

```
--Atualizando os dados da tabela da maneira errada
UPDATE DADOS SET IDADE=30
```

	ID	NOME	IDADE	CPF
1	1	Lucas	30	12345678900
2	2	Claus	30	00987654321

```
--Excluindo todos os dados da tabela
DELETE FROM DADOS
```

Utilizando o comando UPDATE da forma errada acabou alterando todos os dados da tabela de uma vez, por isso como não estamos utilizando comandos de salvamento vamos excluir todos os dados e colocá-los de novo, ou poderíamos ter alterado os dados que forem atualizados incorretamente porém se a quantidade fosse muito grande seria inviável em comparação com inseri-los novamente.

```
--Atualizando os dados da tabela da maneira certa
UPDATE DADOS SET IDADE=30 WHERE NOME='Claus'
```

	ID	NOME	IDADE	CPF
1	3	Lucas	22	12345678900
2	4	Claus	30	00987654321

Agora atualizamos os dados da forma certa, e como resultado vemos que apenas o dado que foi designado sofreu a alteração.

```
--Excluindo um dado da tabela
DELETE FROM DADOS WHERE NOME = 'Lucas'
```

	ID	NOME	IDADE	CPF
1	4	Claus	30	00987654321

Excluindo um dos dados da tabela, utilizando o conceito certo para designar o dado a ser excluído.

```
--Selecionando os dados que deseja da tabela
SELECT NOME,IDADE FROM DADOS
```

	NOME	IDADE
1	Claus	30

Com esse comando selecionamos somente algumas colunas da tabela para exibir em vídeo, para não pegar informações desnecessárias para a situação.

```
--Excluindo a tabela DADOS
DROP TABLE DADOS

--Excluindo o banco de dados CLIENTES
DROP DATABASE CLIENTES
```

E por fim excluímos a tabela DADOS junto com o banco de dados CLIENTES.

10. Design Patterns – Padrões de projeto

Introdução

Quando estamos falando de um software, não estamos somente nos referindo a como ele funciona na prática, e sim em todos os componentes e conceitos utilizados para montá-lo.

O que isso significa?

De forma direta, significa que não basta um código funcionar, caso ele não seja bem escrito, ele se tornará obsoleto no futuro, por motivos diversos,.como:

- Difícil manutenção;
- Difícil aplicação de novas funcionalidades;
- Resultando em menor entrega;

Por esse motivo que as linguagens vem melhorando a abstração ao mundo real, de forma a ficar mais fácil e claro o entendimento de um código. Com o advento da POO (Programação Orientada a Objeto), códigos mais limpos foram se tornando possíveis, com padrões se moldando ao longo do tempo.

São justamente estes padrões que serão abordados neste manual.

Definição

A primeira definição que aparece no Google de Design Pattern é:

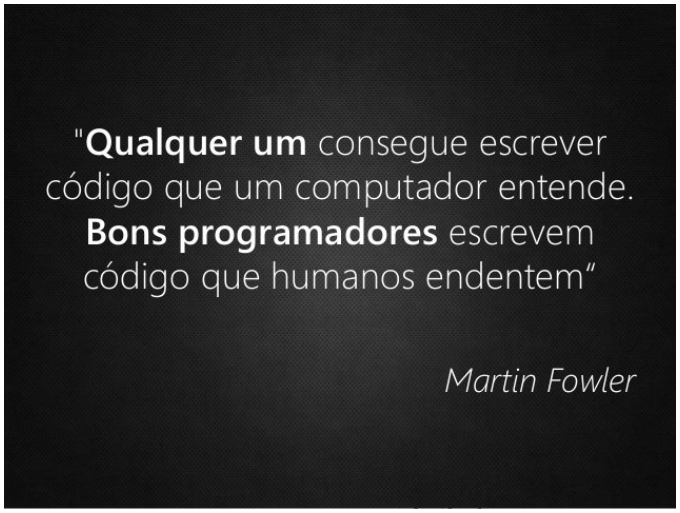
Design Patterns ou padrões de projetos são soluções generalistas para problemas recorrentes durante o desenvolvimento de um software. Não se trata de um framework ou um código pronto, mas de uma definição de alto nível de como um problema comum pode ser solucionado.

Isto significa que, Design Patterns nada mais são do que formas de organizar seu código de forma mais clara, conforme padrões estudados por especialistas da área.

Por que devo utilizá-los?

Como dito anteriormente, códigos sujos atrasam a vida de programadores, acabam com sua eficiência e podem até inutilizar um código.

Com a vinda de Design Patterns, fica mas claro e objetiva a forma em que se pode otimizar e claramente deixar um código mais inteligível e “LIMPO”.



São classificados em 3 modelos:

. Padrões criacionais → Ajudam na criação/instanciação de objetos, aumentam a flexibilização e melhoram a reutilização do código:

- *Factory Method;*
- *Abstract Factory;*
- *Builder;*
- *Prototype;*
- *Singleton.*

. Padrões estruturais → Formas de montar objetos e classes em estruturas maiores, ainda com um escopo flexível e eficiente:

- *Adapter;*
- *Bridge;*
- *Composite;*
- *Decorator;*
- *Facade;*
- *Flyweight;*
- *Proxy.*

. Padrões comportamentais → Modelos de comunicação e definição responsabilidades entre objetos

- *Chain of Responsibility;*
- *Command;*
- *Iterator;*
- *Mediator;*
- *Observer;*
- *State;*

- *Strategy;*
- *Template Method;*
- *Visitor.*

Todos os padrões citados acima possuem soluções para problemas cotidianos de um software. No manual, abordaremos poucos temas, porém os que são mais utilizados. Caso se interesse, recomendamos o site ***Refactoring Guru*** para pesquisas sobre o assunto, possuindo uma biblioteca repleta de exemplos, explicações e exemplos de código em diversas linguagens.

Os códigos utilizados aqui foram embasados fortemente neste site.

Singleton

Esta aqui um exemplo de Design Pattern muito utilizado. Embora viole um dos princípios SOLID de Responsabilidade Única, ela funciona como um pé na roda do desenvolvedor.

Ele se baseia em uma classe normal, em que só possui uma, e somente uma instância de classe.

Mas para que isso serve? Por que não utilizar uma classe estática?

Classes estáticas, além de não poderem ser passadas como parâmetros, não podem ser usadas em princípios de herança.

Um Singleton pode ser uma ótima pedida para variáveis de configuração e para aplicações Multi-Threading.

A seguir, um código de um Singleton Simples para Multi-Threading, que não possui a melhor performance, porém é inteligível e confiável.

```
public sealed class Singleton
{
	private static Singleton instance = null;
	private static readonly object padlock = new object();

	Singleton() { }

	public static SingletonInstance
	{
		get
		{
			lock (padlock)
			{
				if (instance == null)
				{
					instance = new Singleton();
				}
			return instance;

			}
		}
	}
}
```

Observamos um padrão:

- Um objeto estático que é nulo no começo.
- Um lock para o threadsafe
- Um construtor/método que retorna a instância anterior ou, caso nula, uma nova instância que será a única do projeto.

Existem algumas variações do Singleton, cada um com seus benefícios e malefícios, porém, este é o mais comumente utilizado.

Builder

Outro padrão muito utilizado é o BUILDER.

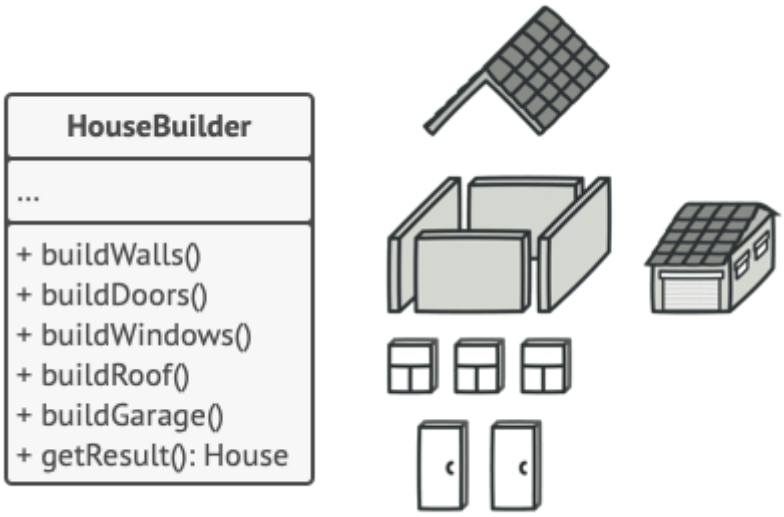
Imagine que você precisa construir um computador. Uma solução, seria criar uma classe Computador, que é herdada pelas variações de Computador. Mas uma hora ou outra iríamos observar inúmeras subclasses.

Este Design Pattern dá uma ótima solução para isso.

Ele diz para criar Classes Builder, que possuem uma interface em comum.

Esses Builders possuem métodos, e o objeto é criado da forma como foi programado.

O exemplo a seguir aborda o assunto de uma forma ilustrativa:



Ex:

```
class Computador
atributo componentes
atributo cores
      Interface Ibuilder
- buildMotherBoard();
- buildSSD();

      class DesktopBuilder : Ibuilder
private produto = novo Computador
public DesktopBuilder()
{
      produto.Reset();
}

public void Reset()
{
      produto = new Computador();
}
- buildMotherBoard();
- buildSSD();
public Computador retornaProduto()
{
      Computador aux = produto;
      produto.Reset();
      return aux;
}
```

Também pode haver uma classe diretora, em que possui métodos prontos para a contrução ordenada. Como se fosse um “pacote pronto”.

```
Public class ComputerDirector
- private Ibuilder construtor
- public Ibuilder Construtor
{
      set {construtor = value;}
}

- buildSimple()
{
      buildMotherBoard();
}
- buildAll()
{
      buildMotherBoard();
      buildSSD();
}
```

Com isso, a modulação e flexibilização do código são garantidas.

11. Coleções Genéricas

Coleções Genéricas são baseadas em recursos no C# que não são genéricas para assim gerar um tipo de recurso de uso como o próprio nome diz, genérico. Os genéricos permitem que você personalize um método, uma classe, uma estrutura ou uma interface para o tipo exato de dados no qual ele atua. Por exemplo, em vez de usar a classe Hashtable, que permite que as chaves e os valores sejam de qualquer tipo, você pode usar a classe genérica Dictionary<TKey,TValue> e especificar o tipo permitido para a chave e o tipo permitido para o valor. Entre os benefícios de genéricos estão maior reutilização de códigos e segurança de tipos.

Genéricos são classes, estruturas, interfaces e métodos que possuem espaços reservados (parâmetros de tipo) para um ou mais dos tipos que eles armazenam ou usam. Uma classe de coleção genérica pode usar um parâmetro de tipo como um espaço reservado para o tipo de objetos que ela armazena; os parâmetros de tipo aparecem como os tipos de seus campos e os tipos de parâmetro de seus métodos. Um método genérico pode usar seu parâmetro de tipo como o tipo de seu valor de retorno ou como o tipo de um de seus parâmetros formais. O código a seguir ilustra uma definição de classe genérica simples.

Coleções GENÉRICAS mais utilizadas

Mutável	Amortizado	Pior caso	Imutável	Complexidade
Stack<T>.Push	O (1)	O (n)	ImmutableStack<T>.Push	O (1)
Queue<T>.Enqueue	O (1)	O (n)	ImmutableQueue<T>.Enqueue	O (1)
List<T>.Add	O (1)	O (n)	ImmutableList<T>.Add	O (log n)
List<T>.Item[Int32]	O (1)	O (1)	ImmutableList<T>.Item[Int32]	O (log n)
List<T>.Enumerator	O (n)	O (n)	ImmutableList<T>.Enumerator	O (n)
HashSet<T>.Add, pesquisa	O (1)	O (n)	ImmutableHashSet<T>.Add	O (log n)
SortedSet<T>.Add	O (log n)	O (n)	ImmutableSortedSet<T>.Add	O (log n)
Dictionary<T>.Add	O (1)	O (n)	ImmutableDictionary<T>.Add	O (log n)
Dictionary<T> busca	O (1)	O (1) – ou estritamente O (n)	ImmutableDictionary<T> busca	O (log n)
SortedDictionary<T>.Add	O (log n)	O (n log n)	ImmutableSortedDictionary<T>.Add	O (log n)

List é muito parecido com a classe ArrayList, que era a opção de lista de acesso antes de listas genéricas compatíveis com C#. Portanto, você também verá que o List pode fazer muitas das mesmas coisas que um Array (que também implementa a interface IList), mas em muitas situações, List é mais simples e fácil trabalhar. Por exemplo, você não precisa criar uma lista com um tamanho específico; em vez disso, basta criá-la e o .NET a expandirá automaticamente para se ajustar à quantidade de itens à medida que você os adiciona.

```
private void btnList_Click(object sender, EventArgs e)
{
    Lista.Items.Clear();

    List<string> lista = new List<string>();

    lista.Add("André Cini");
    lista.Add("Nathan Vilela");
    lista.Add("Alexander Vilaça");
    lista.Add("Claus Alberto");
    lista.Add("Guilherme Turtera");
    lista.Add("Lucão Antônio");
    lista.Add("Ivan Zanutto");
    lista.Add("Matheus Novais");

    foreach (string aux in lista)
    {
        Lista.Items.Add(aux);
    }
}
```

O HashSet se baseia numa lista que não permite repetições em seus elementos nas quais podem ser de qualquer tipo, utilizando assim um recurso útil quando se quer uma coleção que evite o acúmulo de informações repetidas, mais as propriedades comuns da lista como a propriedade de elementos dentro da coleção e outras propriedades funcionais dentro dessa coleção genérica, outro fator importante que deve notar é que em relação a lista os HashSet tem um modo diferente de processamento que então poderia assim influenciar na escolha do tipo de coleção.

```
    HashSet<string> veiculos = new HashSet<string>()
        {
        "Carro",
        "Moto",
        "Avião",
        "Helicóptero",
        "Navio",
        "Elevador",
        "Bicicleta",
        "Em um HashSet não pode haver elementos repetidos"
        };


    if (veiculos.Add(texto)) //.Add retorna um bool
    {
        MessageBox.Show(string.Format("Item '{0}' Adicionado!", texto));
    }
    else
    {
        MessageBox.Show(string.Format("Item '{0}' não Adicionado!", texto));
    }

    foreach (string aux in veiculos)
    {
        Lista.Items.Add(aux);
    }

    teste = !teste;
}
```

Dictionary é um recurso que, de fato, veio do dicionário que nós conhecemos, onde uma palavra é única mas pode dispor de vários significados aplicados a mesma, dando uma funcionalidade ao código em certas condições nas quais o programador se sinta necessitado de uma solução como esta, sempre influenciado pelo impacto que aquele recurso causa no andamento do código.

```
    Dictionary<int, string> alunos = new Dictionary<int, string>()
    {
        {08120001, "André Cini"},
        {08120002, "Nathan Vilela"},
        {08120003, "Guilherme Turtera"},
        {08110001, "Claus Alberto"},
        {08100001, "Matheus Novais"},
        {08110002, "Lucas Antônio"},
        {08020001, "Ivan Zanutto"},
        {08120004, "Alexsander Vilaça"}

    };
    //alunos.Add(100, "GABRIEL");

    foreach (KeyValuePair<int, string> item in alunos)
    {
        Lista.Items.Add(item.Key + " -> " + item.Value);
    }

    Lista.Items.Add("");
    Lista.Items.Add(" Podemos resumir, de forma simplista, a definição ");
    Lista.Items.Add("do dictionary à uma lista indexada. Ele nos permite ");
    Lista.Items.Add("narmazenar pares de chave + valor, sendo que estes ");
    Lista.Items.Add("npodem ser de qualquer tipo.");
    Lista.Items.Add("");
    Lista.Items.Add(" O melhor é que esta lista é indexada pela chave ");
    Lista.Items.Add("que você define, se tornando muito fácil e rápido ");
    Lista.Items.Add("localizar itens dentro dela.");

}
```

A classe SortedList representa uma coleção de pares de chave/valor que são classificados pelas chaves e podem ser acessados por chave e por índice. Esta classe está incluída no namespace System.Collection.

Um objeto SortedList gerencia internamente dois arrays para armazenar os elementos da lista; ou seja, um array para as chaves e outro array para os valores. Cada elemento é um par de chave/valor que pode ser acessado como um objeto DictionaryEntry. Uma chave não pode ser null, mas um valor pode.

```
    SortedList<int, string> alunos = new SortedList<int, string>()
    {
        {08120001, "André Cini"},
        {08120002, "Nathan Vilela"},
        {08120003, "Guilherme Turtera"},
        {08110001, "Claus Alberto"},
        {08100001, "Matheus Novais"},
        {08110002, "Lucas Antônio"},
        {08020001, "Ivan Zanutto"},
        {08120004, "Alexsander Vilaça"}
    };

    foreach (KeyValuePair<int, string> aux in alunos)
    {
        Lista.Items.Add(aux.Key + "->" + aux.Value);
    }

}
```

A coleção SortedDictionary<TKey,TValue> é semelhante à classe Dictionary<TKey,TValue> na forma de usar mas muito diferente em sua execução.

A classe SortedDictionary<TKey,TValye> usa uma árvore binária para manter os itens em ordem pela chave. Como consequência desta ordenação, o tipo usado para a chave deve implementar a interface IComparable<TKey> de modo que as chaves possam ser corretamente ordenadas.

O dicionário ordenado gasta um pouco de tempo a mais na pesquisa devido a capacidade de manter os itens em ordem, assim as operações de inclusão/exclusão/pesquisa em um dicionário ordenado são logarítmicas : O(log n).

```
SortedDictionary<int, string> alunos = new SortedDictionary<int, string>()
{
     {08120001, "André Cini"},
    {08120002, "Nathan Vilela"},
    {08120003, "Guilherme Turtera"},
    {08110001, "Claus Alberto"},
    {08100001, "Matheus Novais"},
    {08110002, "Lucas Antônio"},
    {08020001, "Ivan Zanutto"},
    {08120004, "Alexsander Vilaça"}

};

foreach (KeyValuePair<int, string> item in alunos)
{
    Lista.Items.Add(item.Key + "->" + item.Value);
}

}
```

A Queue implementa o conceito “First-in, First-out”, ou “Primeiro a entrar, Primeiro a sair”. Ou seja, o primeiro elemento inserido na lista é também o primeiro a ser removido. Podemos pensar, por exemplo, em uma fila de banco. A primeira pessoa a entrar na fila será a primeira a ser atendida, logo, a primeira a sair da fila.

Também é muito comum se usar a expressão FIFO (abreviação de first-in, first-out) para definir a pilha.

```
        Queue<string> fila = new Queue<string>();
        fila.Enqueue("André");
        fila.Enqueue("Nathan");
        fila.Enqueue("Alexander");
        fila.Enqueue("Claus");
        fila.Enqueue("Guilherme");
        fila.Enqueue("Lucão");
        fila.Enqueue("Ivan");
        fila.Enqueue("Matheus");

        MessageBox.Show(fila.Count.ToString());

        foreach (var aux in fila)
        {
            Lista.Items.Add(aux);
        }

    }
```

O conceito implementado pela Stack é o de “Last-in, First-out” (também chamado LIFO ou FILO, de first-in, last-out, que tem o mesmo sentido prático) ou “Último a entrar, Primeiro a sair”.

Imagine, por exemplo, que você esteja organizando vários pratos. Inicialmente você organiza todos uns sobre os outros, formando uma pilha de pratos. Logo após, você irá retirar um a um para organizar no seu devido local. Perceba que o primeiro prato removido foi o último a ser inserido na pilha, aquele que ficou na parte superior. Enquanto isso, o primeiro prato inserido na pilha, aquele que está abaixo de todos, será o último removido.

```
    Stack<string> pilha = new Stack<string>();

    pilha.Push("André");
    pilha.Push("Nathan");
    pilha.Push("Alexander");
    pilha.Push("Claus");
    pilha.Push("Guilherme");
    pilha.Push("Lucão");
    pilha.Push("Ivan");
    pilha.Push("Matheus");

    foreach (string aux in pilha)
    {
        Lista.Items.Add(aux);
    }
}
```

12. PRINCIÍPIOS S.O.L.I.D.:

12.1 – INTRODUÇÃO

SOLID é um acrônimo dos cinco primeiros princípios da programação orientada a objetos e design de códigos identificados por Robert C. Martin (ou Uncle Bob) por volta do ano 2000. O acrônimo SOLID foi introduzido por Michael Feathers, após observar que os cinco princípios poderiam se encaixar nesta palavra.

Os princípios SOLID devem ser aplicados para se obter os benefícios da orientação a objetos, tais como: manter, adaptar e ajustar as alterações de escopo; seja testado e de fácil entendimento; que seja possível realizar alterações com o mínimo de esforço; que possa ser reutilizado a maior quantidade de vezes possíveis; e que permaneça o máximo tempo possível sendo utilizado.

Além disso, utilizando os princípios SOLID é possível evitar problemas muito comuns, como: dificuldade de realização de testes; código sem estrutura ou padrão; dificuldade de isolar as funcionalidades, necessidade de realizar a mesma alterações em diversos pontos; e fragilidade (códio pode ser quebrado após alguma alteração).

12.2 – OS PRINCÍPIOS S.O.L.I.D.:

[S] -> Princípio da Responsabilidade Única

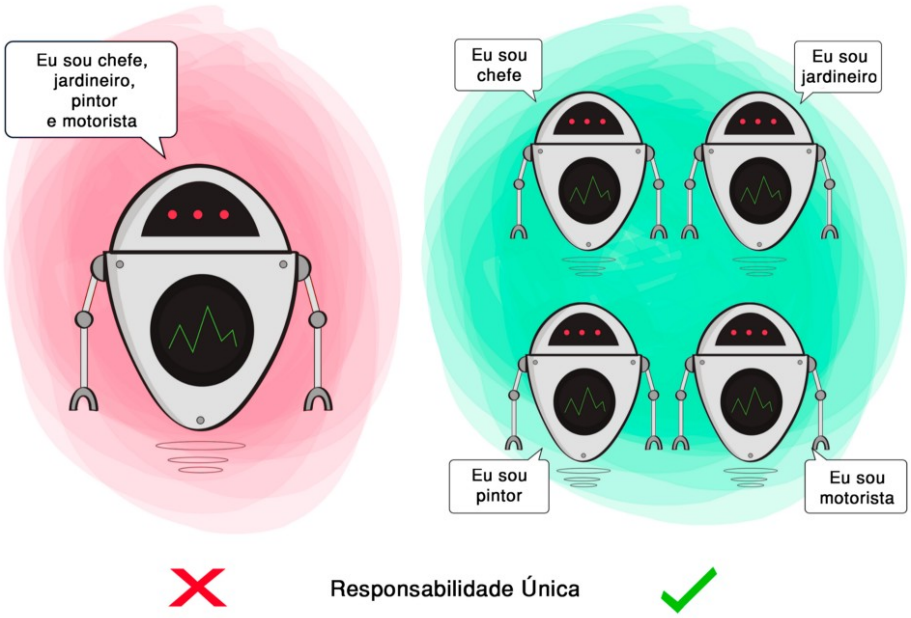
[O] -> Princípio do Aberto/Fechado

[L] -> Princípio da Substituição de Liskov

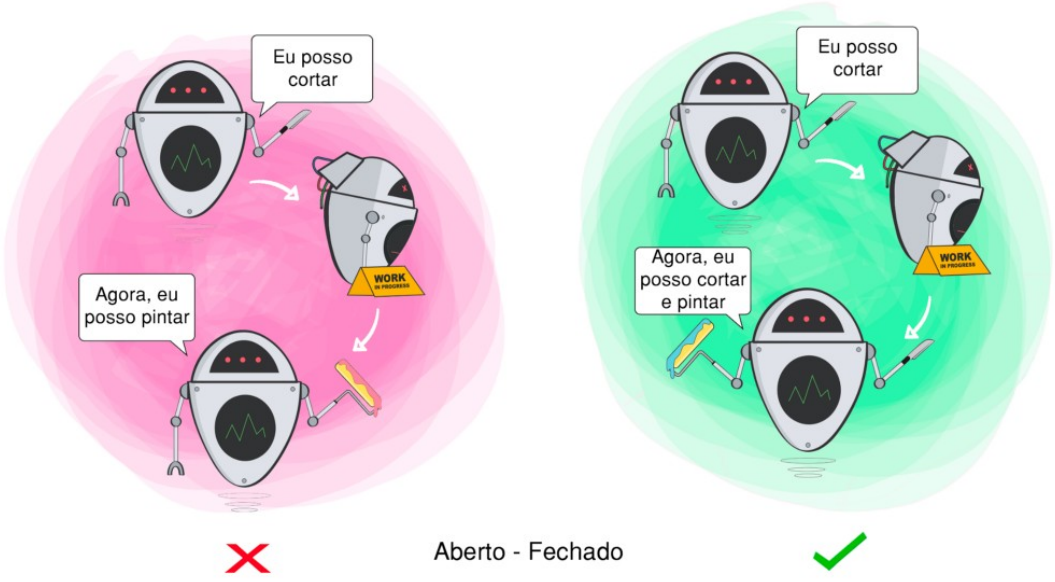
[I] -> Princípio da Segregação de Interfaces

[D] -> Princípio da Injeção de Dependência

[S] = “Uma classe deve ter apenas um motivo para mudar”, ou seja, deve ter uma única responsabilidade. Basicamente, esse princípio trata especificamente a coesão. A coesão é definida como a afinidade funcional dos elementos de um módulo. Se refere ao relacionamento que os membros desse módulo possuem, se possuem uma relação mais direta e importante. Dessa forma, quanto mais bem definido o que sua classe faz, mais coesa ela é.



[O] = “As entidades de software (classes, módulos, funções etc.) devem ser abertas para ampliação, mas fechadas para modificação”. De forma mais detalhada, diz que podemos estender o comportamento de uma classe, quando for necessário, por meio de herança, interface e composição, mas não podemos permitir a abertura dessa classe para fazer pequenas modificações.

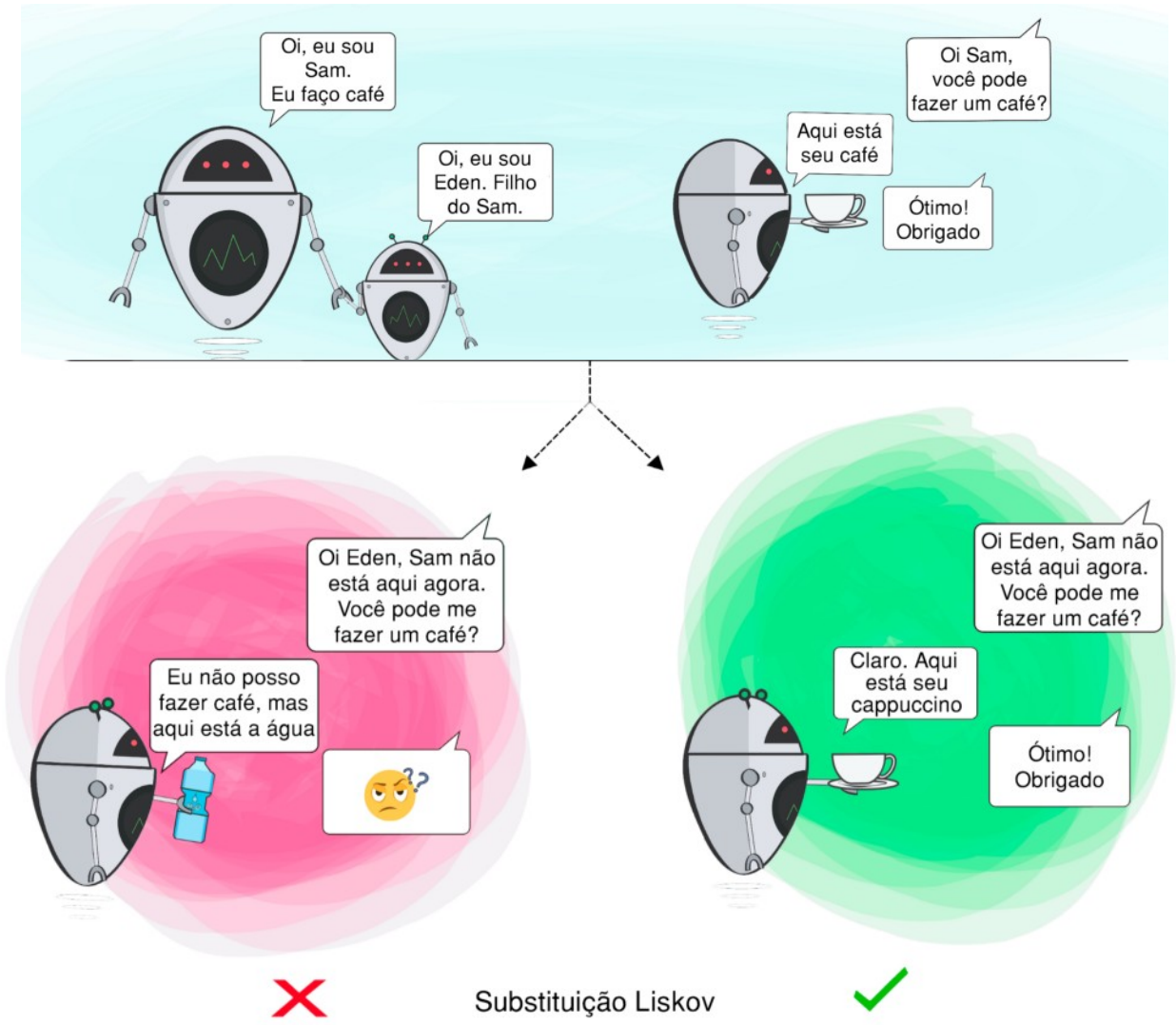


[L] = “Se S é um subtipo de T, os objetos do tipo T em um programa podem ser substituídos por objetos do tipo S sem alterar nenhuma das propriedades desejáveis desse programa.” Quando uma classe filho não pode executar as mesmas ações que sua classe pai, isso pode causar erros.

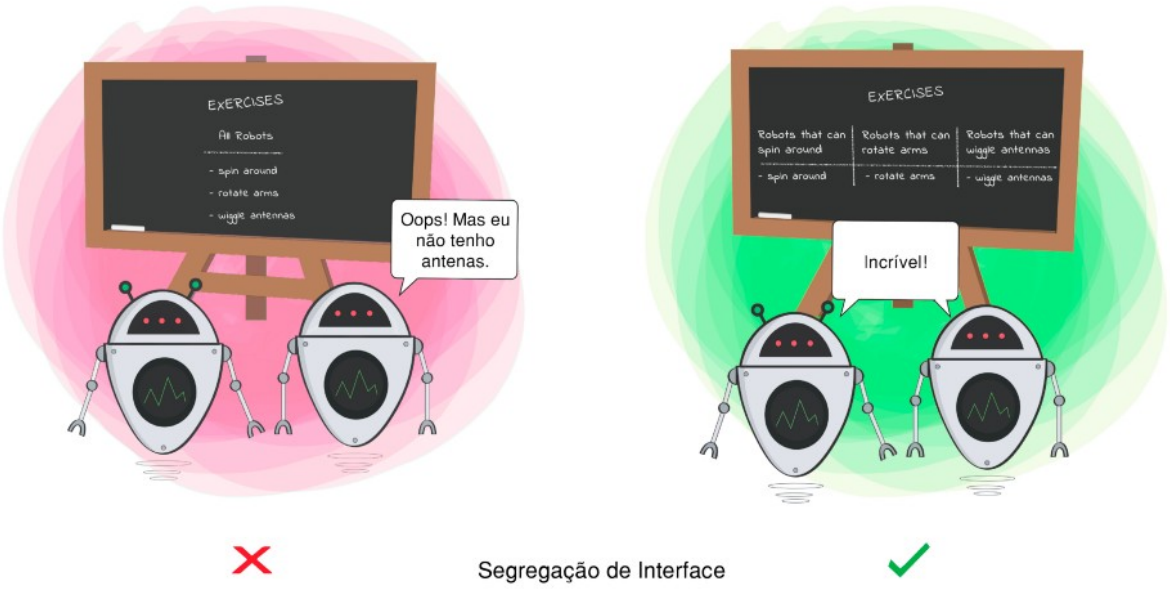
Se você tem uma classe e cria outra classe a partir dela, ela se torna um pai e a nova classe se torna um filho. A classe filho deve poder fazer tudo o que a classe pai pode fazer. Esse processo é chamado de herança.

A classe filho deve ser capaz de processar as mesmas solicitações e entregar o mesmo resultado que a classe pai, ou pode entregar um resultado do mesmo tipo.

Se a classe filho não atender a esses requisitos, significa que a classe filho foi alterada, violando esse princípio.



[I] = “Uma classe não deve ser forçada a implementar interfaces e métodos que não irão utilizar. ” Basicamente, este princípio estabelece que muitas interfaces específicas são melhores do que uma interface única que engloba funções que não serão utilizadas em outras classes, por exemplo.



Segregação de Interface

[D] - “Módulos de alto nível não devem depender de módulos de baixo nível. Ambos devem depender da abstração. Além disso, abstrações não devem depender de detalhes. Os detalhes devem depender de abstrações”.

Módulo de alto nível (ou classe): classe que executa uma ação com uma ferramenta.

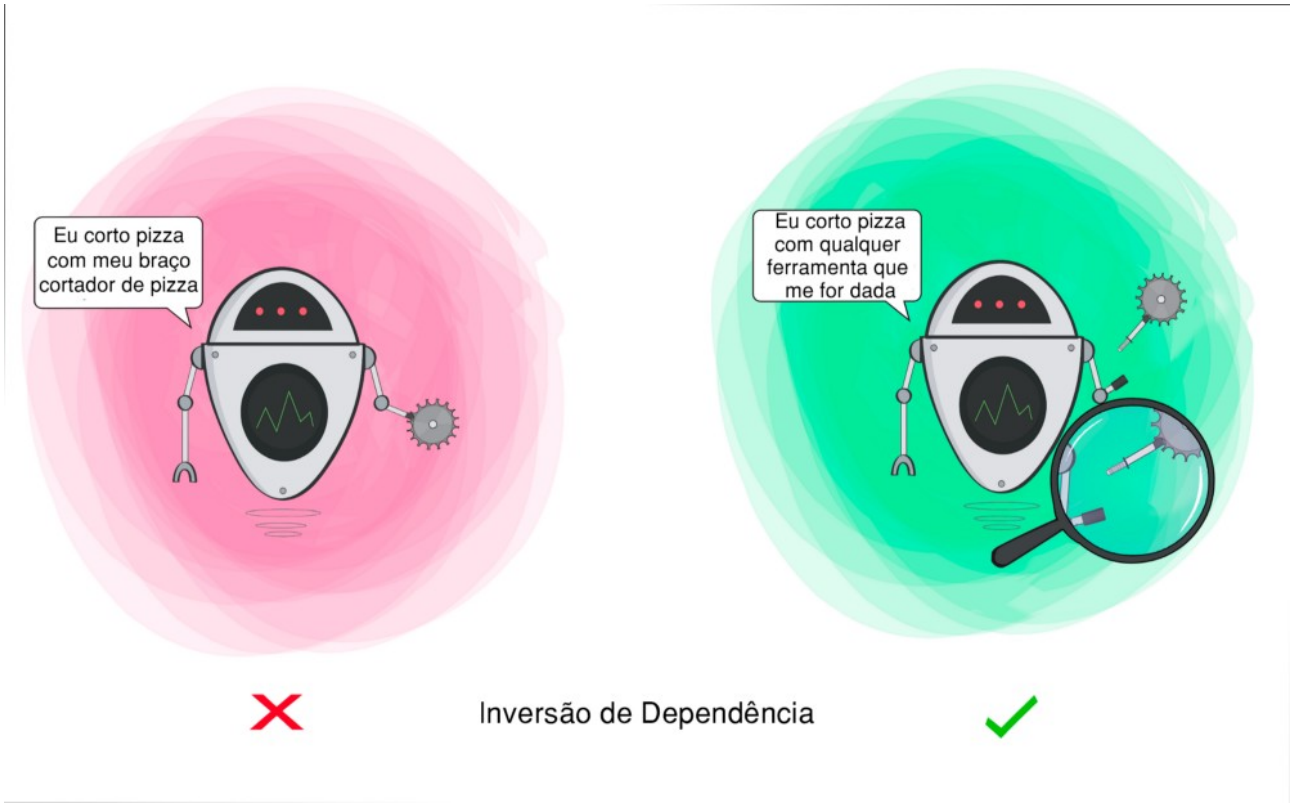
Módulo de baixo nível (ou classe): a ferramenta necessária para executar a ação.

Abstração: representa uma interface que conecta as duas classes.

Detalhes: como a ferramenta funciona.

Este princípio diz que uma classe não deve ser fundida com a ferramenta usada para executar uma ação. Em vez disso, ele deve ser fundido à interface que permitirá que a ferramenta se conecte à Classe.

Ele também diz que a classe e a interface não devem saber como a ferramenta funciona. No entanto, a ferramenta precisa atender às especificações da interface.



12.3 – RESUMO.:

Em suma, os princípios S.O.L.I.D. foram criados para tornar seu código mais limpo, mais fácil de ajustar, estender, testar além de deixá-lo mais inteligível.

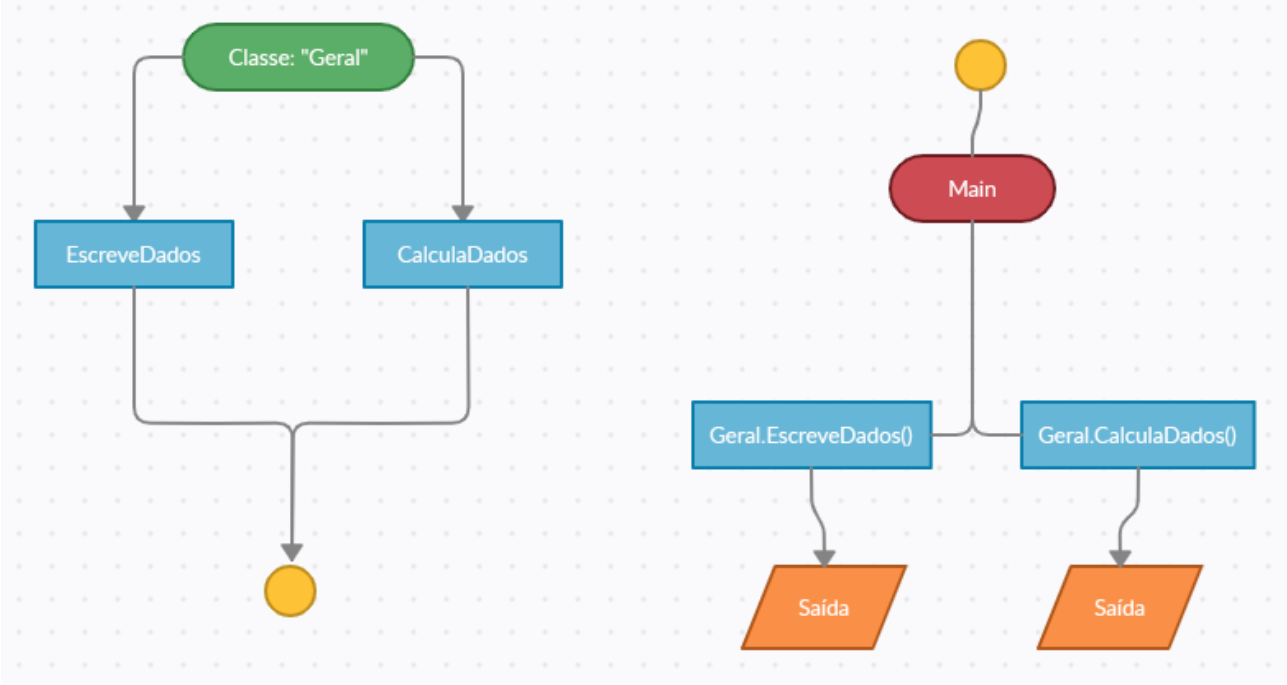
Para consulta rápida, estes princípios seguem possuem, respectivamente, estes objetivos:

[S] - Esse princípio visa separar comportamentos para que, se surgirem erros como resultado de sua alteração, não afetem outros comportamentos não relacionados.
[O] - Este princípio visa estender o comportamento de uma classe sem alterar o comportamento existente. Isso serve para evitar erros quando a classe estiver sendo usada.
[L] - Este princípio visa estender o comportamento de uma classe sem alterar o comportamento existente. Isso serve para evitar erros quando a classe estiver sendo usada.
[I] - Esse princípio visa impor consistência para que a classe pai ou sua classe filho possa ser usada da mesma maneira sem erros.
[D] - Este princípio visa reduzir a dependência de uma classe de alto nível na classe de baixo nível, introduzindo uma interface.

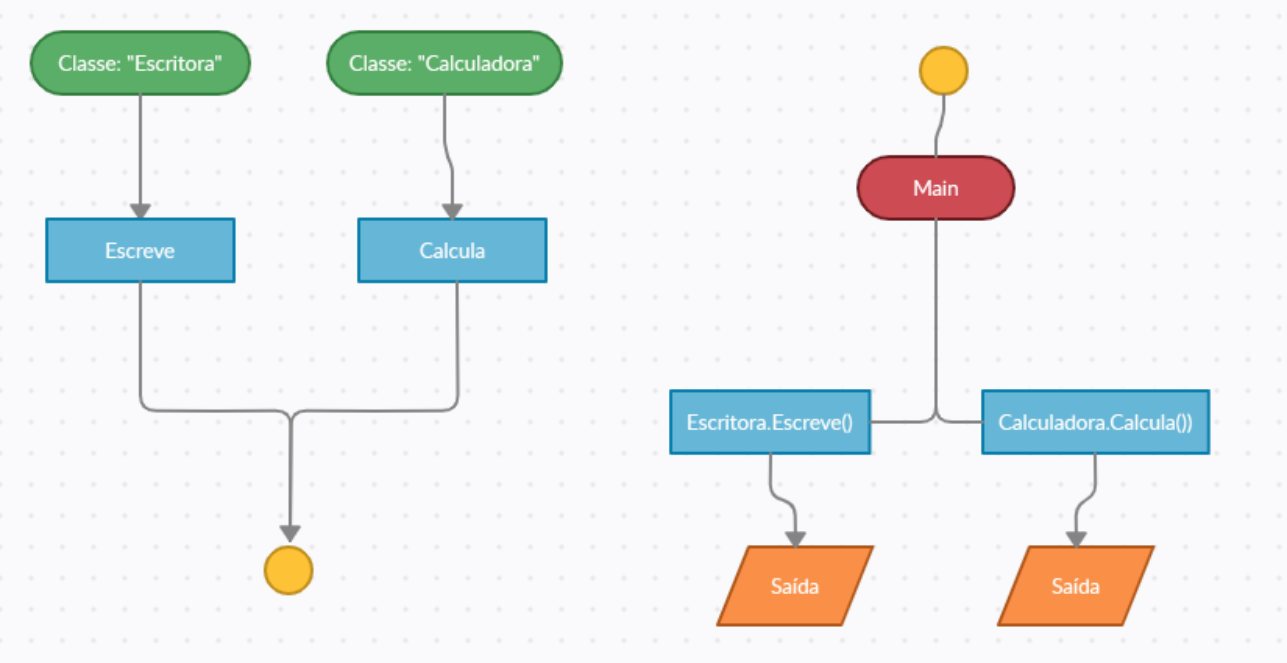
12.3 – EXEMPLOS.:

[S] - Princípio da Responsabilidade Única

ERRADO!

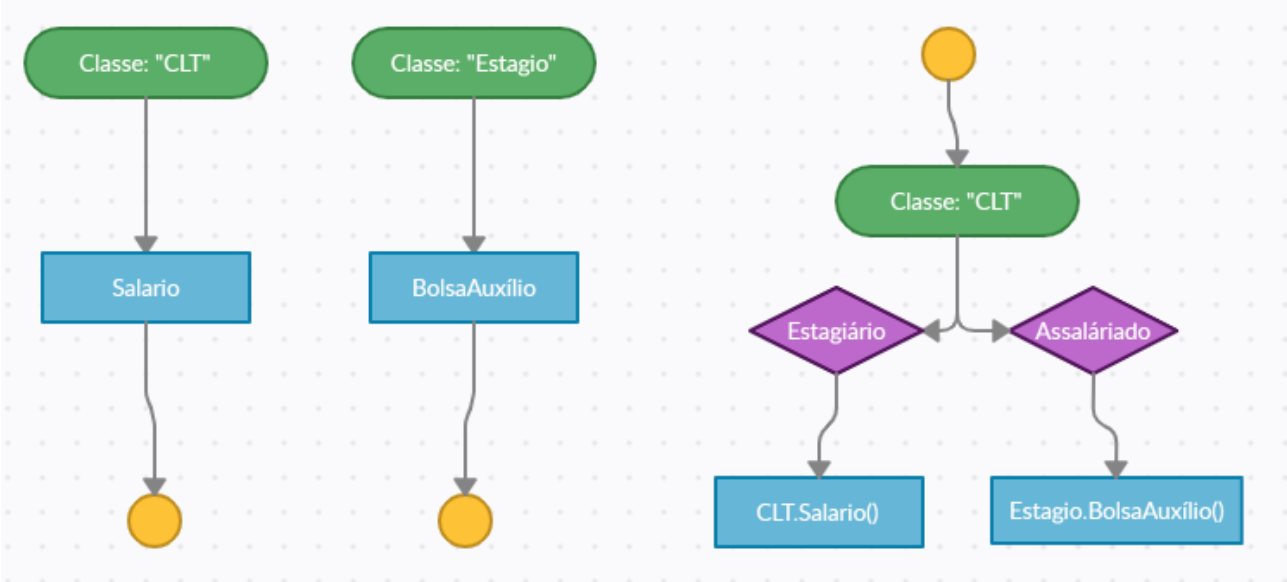


CERTO!

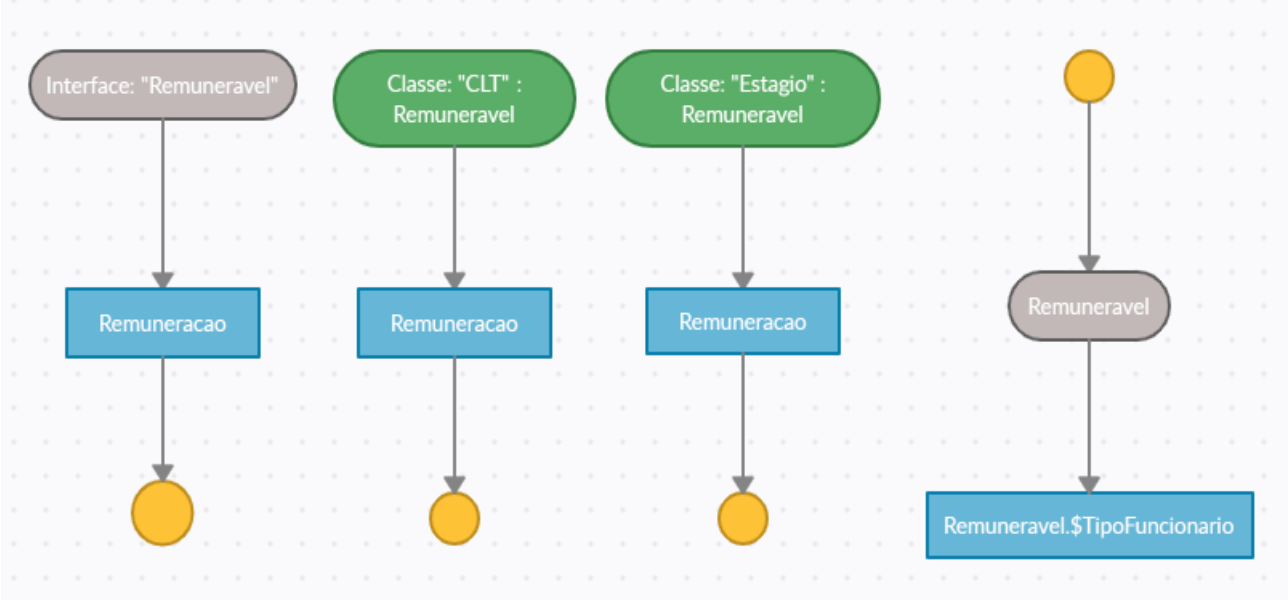


[O] - Princípio do Aberto/Fechado

ERRADO!

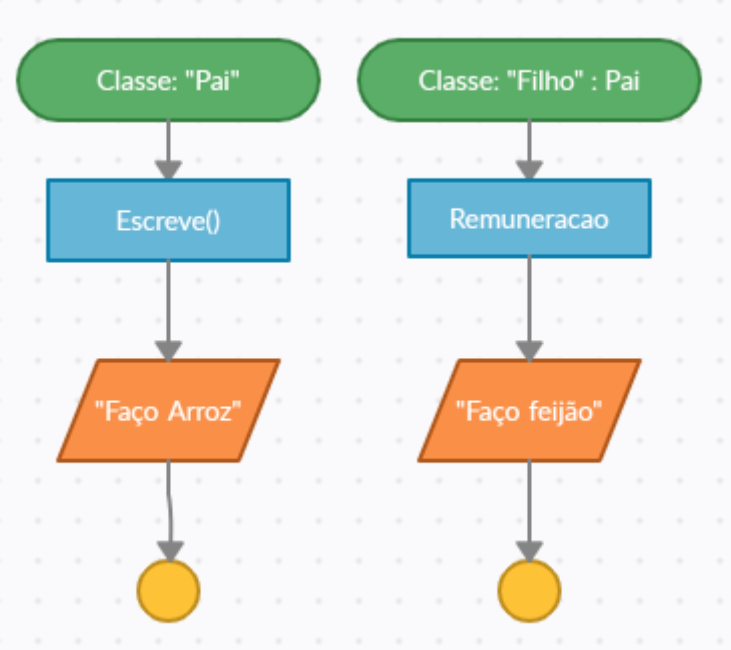


CERTO!

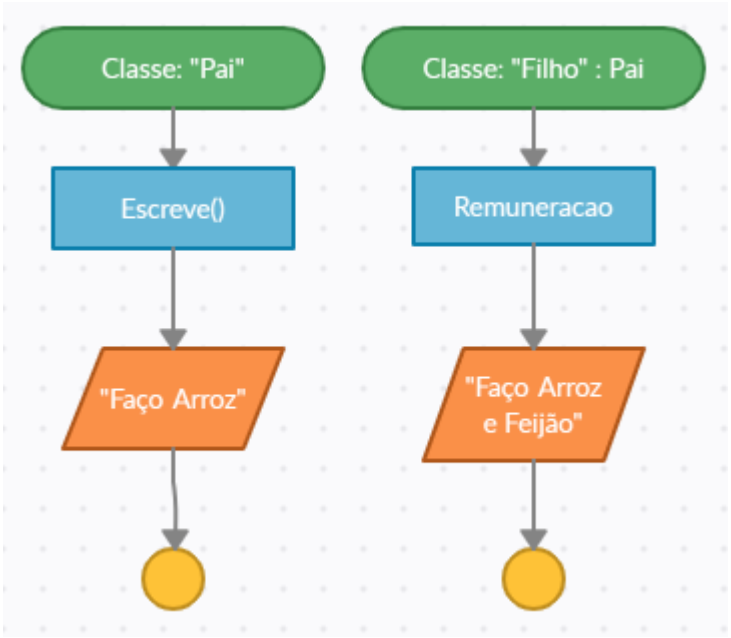


[L] - Princípio da Substituição de Liskov

ERRADO!

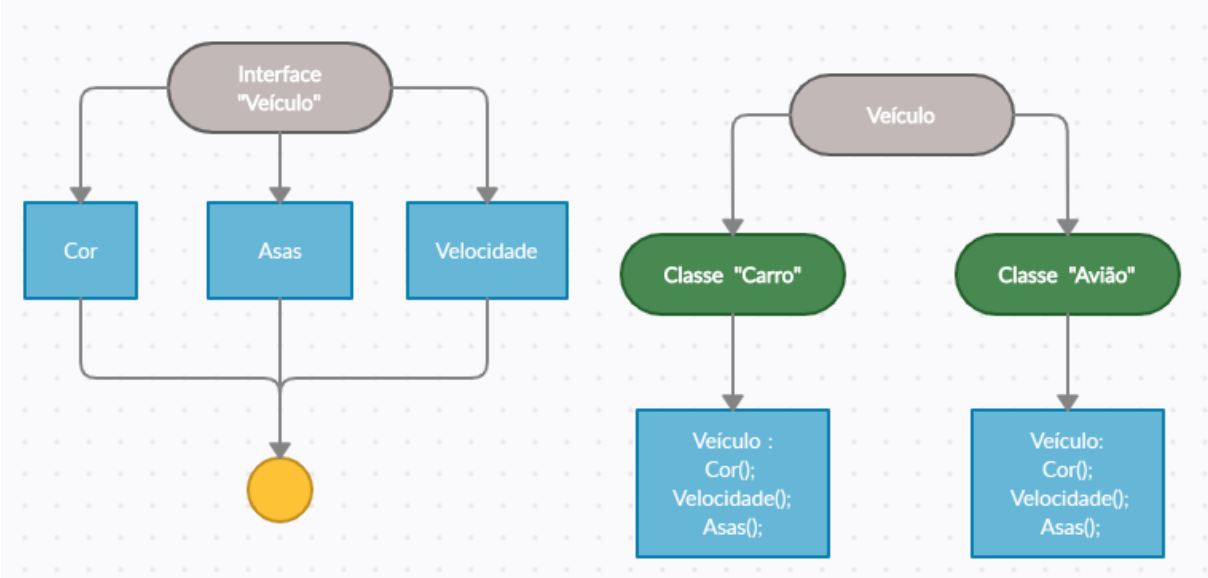


CERTO!

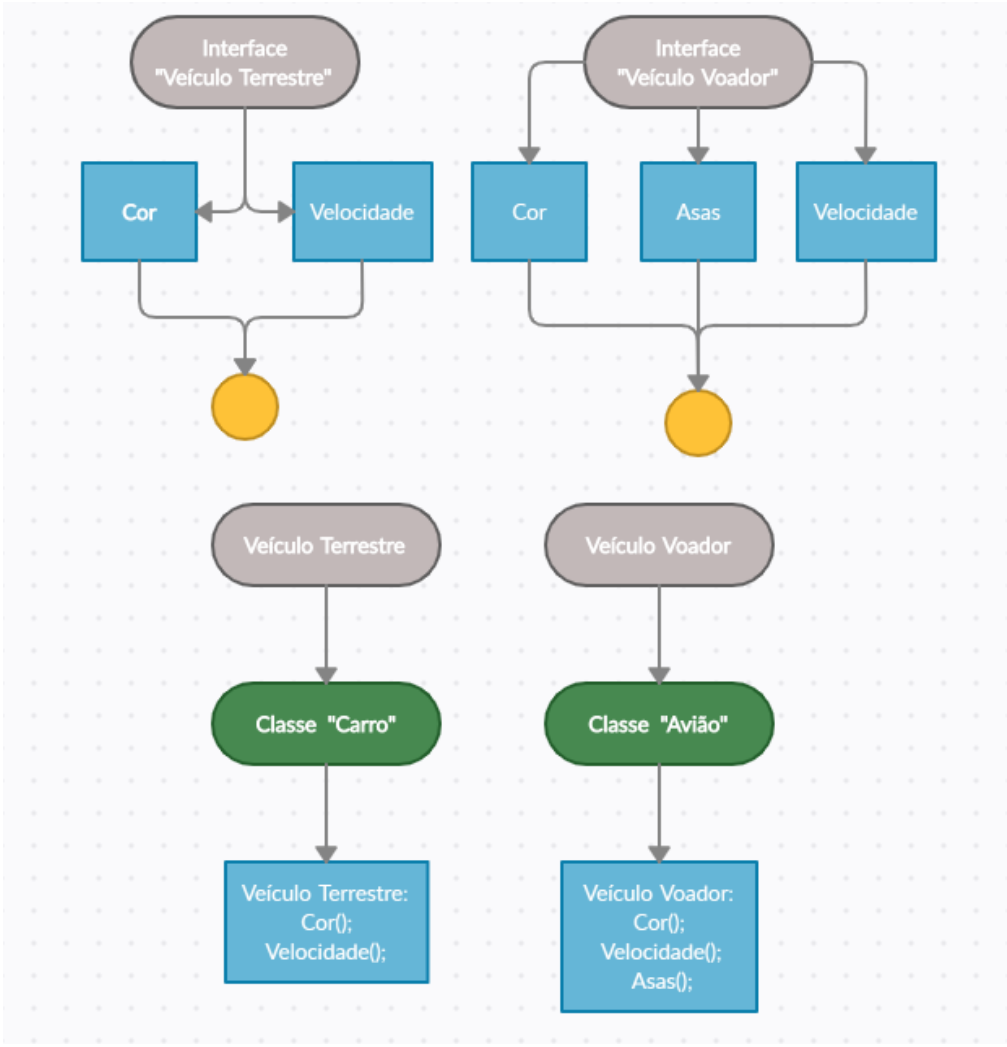


[I] - Princípio da Segregação de Interfaces

ERRADO!

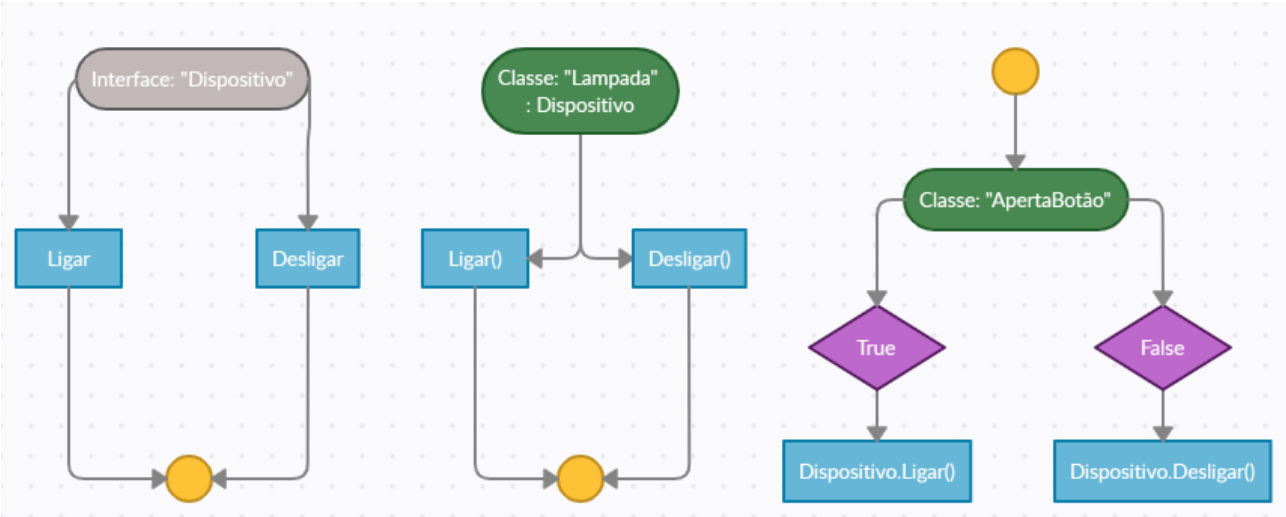


CERTO!

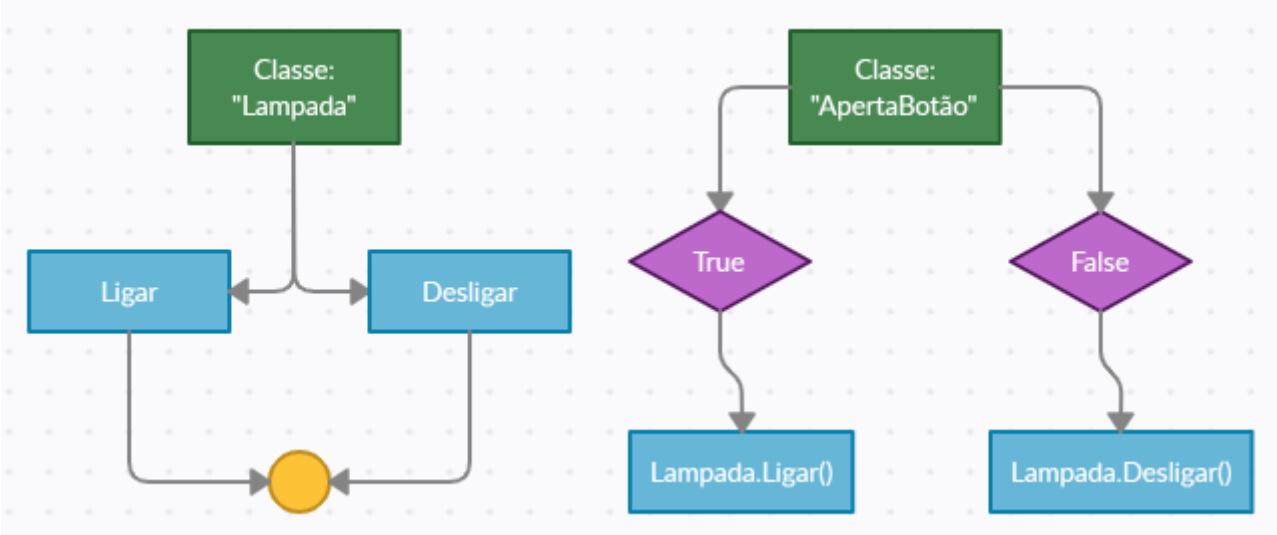


[D] - Princípio da Injeção de Dependência

ERRADO



CERTO


Classe:
"Lampada"
Ligar
Desligar
Classe:
"ApertaBotão"
True
False
Lampada.Ligar()
Lampada.Desligar()

13. *Garbage Collector* C#

No *Common Language Runtime (CLR), o Garbage Collector*, ou traduzindo diretamente, Coletor de Lixo, tem como função gerenciar a memória automaticamente, gerenciando a alocação e o lançamento da memória para um aplicativo, ou seja, não é necessário escrever um código para o gerenciamento de memória, pois ele é automático, impossibilitando o surgimento de alguns erros, como esquecer de liberar um objeto, ou a tentativa de acessar a memória de um objeto que já foi liberado.

13.1 Benefícios do *Garbage Collector*:

- Facilita o trabalho dos desenvolvedores, não necessitando a preocupação de liberar memória manualmente;
- Aloca objetos no *heap* gerenciado com eficiência;
- Recupera os objetos que não estão sendo utilizados, limpa a memória e mantém a memória disponível para futuras alocações;
- Proporciona segurança de memória, garantindo que um objeto não use o conteúdo de outro;

13.2 Condições para a coleta de lixo:

A coleta de lixo acontece quando:

- O sistema possuí pouca memória, detectado pelo próprio Sistema Operacional (SO);
- A memória usada por objetos no *heap* gerenciado ultrapassa um limite aceitável;
- o método GC.Collect é chamado;

13.3 Heap Gerenciado:

Após o Garbage Collector ser iniciado, ele aloca um segmento da memória para armazenar e gerenciar objetos, essa memória recebe o nome de ***heap* gerenciado**.

Existe um ***heap* gerenciado** para cada processo gerenciado, todos os threads no processo alocam memória no mesmo ***heap.***

13.4 Finalize() e Dispose()

O C# possui métodos especiais para liberar a instância de uma classe, a partir da memória, são eles os Finalize() e o Dispose().

Finalize()	Dispose()
É utilizado para liberar recursos não gerenciados (arquivos, conexões, recursos COM, etc), retidos por um objeto antes que ele seja destruído.	É utilizado para liberar recursos não gerenciados (arquivos, conexões, recursos COM, etc), a qualquer momento.
Ele é chamado internamente pelo G.C., e não pode ser chamado pelo usuário.	É chamado explicitamente pelo usuário, a classe que o define deve implementar a interface IDisposable.
Pertence a classe Object.	Pertence a Interface IDisposable.
Implementar quando se tem recursos não gerenciados, e é requisitado que esse recursos sejam coletados quando ocorrer a coleta de lixo.	Implementar quando se está escrevendo uma classe que será utilizada por outros usuários.
Há custo de desempenho.	Não há custo de desempenho.

- **Finalize():**

```
public class Classe : IDisposable
{
    public Classe() //Construtor
    {
    }

    ~Classe() //Finalize
    {
        this.Dispose();
    }

    public void Dispose()
    {
        //código p/ liberar os recursos não gerenciados
    }
}
```

- **Dispose():**

```
void Dispose()
{
    this.Close();
}
```

14. System Reflection

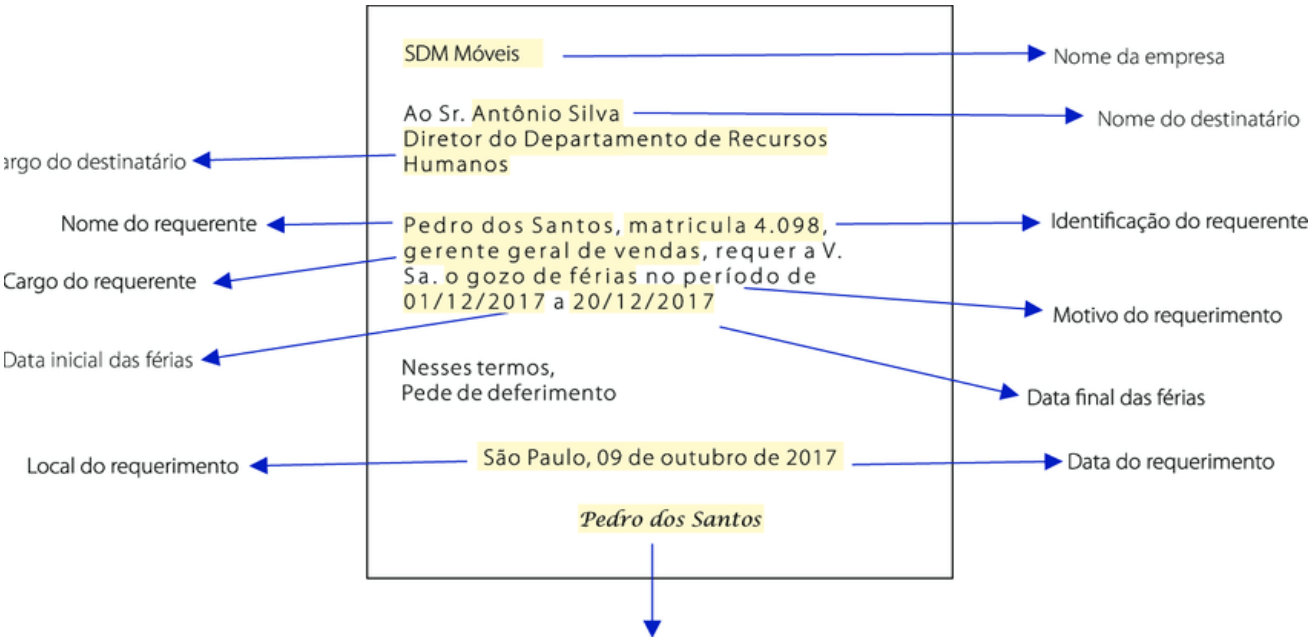
Introdução

- **O que são Metadados?**

Metadados são basicamente informações sobre informações.

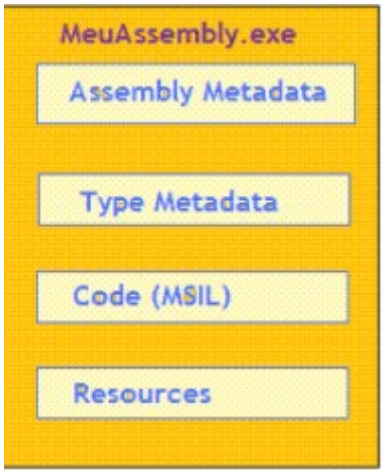
- **Mas o que isso significa?**

Significa simplesmente os dados que estão por trás de uma determinada informação. Através do exemplo a seguir ficará mais claro:



Vemos que até em um texto, temos informações por trás delas mesmas. Isto é METADADOS.

Na programação, estes Metadados seriam informações sobre atributos, como uma Classe, Interface, ou até mesmo um Assembly.



Na imagem acima, podemos ver os metadados.

Temos no Assembly, informações comuns para a compilação e execução de códgio pela máquina, assim como no Type, que possui informações sobre os tipos.

O System Reflection é uma ferramenta do .Net, que permite você a capturar informações em RunTime. Isto se dá muito útil quando se quer saber sobre informações do código, como Types em tempo de execução.

O System Reflection

Nada melhor para aprender sobre o assunto é vê-lo na prática:

O objetivo do código a seguir é implementar as informações sobre os produtos instanciados e listá-los. Mostraremos de forma comum, e através do System Reflection.

Primeiro, criaremos uma aplicação console no C# e criaremos as seguintes classes:

```
public class Isocal
{
    public Boolean Pressao { get; set; }
    public Boolean Frequencia { get; set; }
    public String Preco { get; set; }
    public String Modelo { get; set; }
}

public class Banho
{
    public Boolean Pressao { get; set; }
    public Boolean Frequencia { get; set; }
    public String Preco { get; set; }
    public String Modelo { get; set; }
    public String Temperatura { get; set; }
}
```

Segundo, criaremos uma classe com dois métodos, que têm por objetivo listar as propriedades dos objetos:

```
public class SemSystemReflection
{
    public static void InfoIsocal(Isocal isocal)
    {
        string auxiliar = "";
        auxiliar += "Equipamento : Isocal\n";
        auxiliar += "Pressão: " + isocal.Pressao.ToString()+"\n";
        auxiliar += "Frequência: " + isocal.Frequencia.ToString()+"\n";
        auxiliar += "Preço: " + isocal.Preco + "\n";
        auxiliar += "Modelo: " + isocal.Modelo + "\n";

        Console.WriteLine(auxiliar);
    }

    public static void InfoBanho(Banho banho)
    {
        string auxiliar = "";
        auxiliar += "Equipamento : Banho\n";
        auxiliar += "Pressão: " + banho.Pressao.ToString() + "\n";
        auxiliar += "Frequência: " + banho.Frequencia.ToString() + "\n";
        auxiliar += "Preço: " + banho.Preco + "\n";
        auxiliar += "Modelo: " + banho.Modelo + "\n";
        auxiliar += "Temperatura: " + banho.Temperatura + "\n";

        Console.WriteLine(auxiliar);
    }
}
```

Desta forma, podemos listar os produtos, porém precisaremos de um método para cada produto adicionado, além de infligir princípios SOLID.

Agora, criaremos a forma através do System Reflection:

```
public class ComSystemReflection
{
    public static void InfoEquipamento(object obj)
    {
        var tipo = obj.GetType();

        string aux = "";
        aux += "Equipamento : " + tipo.Name + "\n";

        foreach (var prop in tipo.GetProperties())
            aux += prop.Name + ": " + prop.GetValue(obj, null) + "\n";

        Console.WriteLine(aux);
    }

}
```

Podemos perceber neste código uma otimização.

Ele faz as mesmas coisas que o primeiro, porém de uma forma automatizada.

Para qualquer objeto recebido, ele listará o nome do atributo referente ao objeto, o valor referente a essas propriedades, junto com seus respectivos nomes.

Para ver na prática, adicione o seguinte código no Main:

```
static void Main(string[] args)
{
    Isocal IsocalColgate = new Isocal()
    {
        Pressao = true,
        Frequencia = true,
        Preco = "R$10000",
        Modelo = "MCSXV"
    };

    Banho BanhoColgate = new Banho()
    {
        Pressao = false,
        Frequencia = false,
        Preco = "R$50000",
        Modelo = "T-25N",
        Temperatura = "-25°C"
    };

    Console.WriteLine("Sem System Reflection\n");
    SemSystemReflection.InfoBanho(BanhoColgate);
    SemSystemReflection.InfoIsocal(IsocalColgate);


    Console.WriteLine("\nCom SystemReflection\n");
    ComSystemReflection.InfoEquipamento(BanhoColgate);
    ComSystemReflection.InfoEquipamento(IsocalColgate);
    Console.ReadKey();
}
```

Vejamos o programa rodando:

Vemos que o para funções de listagem o System Reflection funciona de maneira automatizada e eficiente. Este é apenas um exemplo simples de como utilizá-lo, porém temos que ele funciona para muitas utilidades além desta.

15. Ofuscamento de Código

15.1 Introdução

Para iniciar, entraremos com uma breve explicação de como um sistema de transformação de dados funciona.

Tipos de transformação de código de alto nível em baixo nível:

- ***Compilador:***

O compilador, basicamente pega o código fonte e o transforma em linguagem de máquina. Isto é bom para grandes aplicações e mais difícil de ser feita uma engenharia reversa.

- ***Interpretador:***

O interpretador, por sua vez, transforma códigos fontes em linguagem de máquina em tempo de execução, ou seja, enquanto o arquivo roda. Sua vantagem é que é mais fácil de se fazer atualizações e mais rápido de ser inicializado.

- ***Tradutor (Compilador + Interpretador)***

O tradutor basicamente mescla o benefício de cada um dos citados acima. Utilizado em linguagens como Java e o C#, o Tradutor transforma o código fonte em um código intermediário. No caso do C#, compila em uma linguagem conhecida como MSIL (Microsoft Intermediate Language), e em tempo de execução, interpreta este código.

15.2 Problema

Se pararmos para pensar, um grande problema de desenvolvedores de software é a segurança de seu código, pois muitas pessoas mau intencionadas podem plagiá-lo, ter brechas de segurança, entre muitos outros malefícios de se possuir um código aberta.

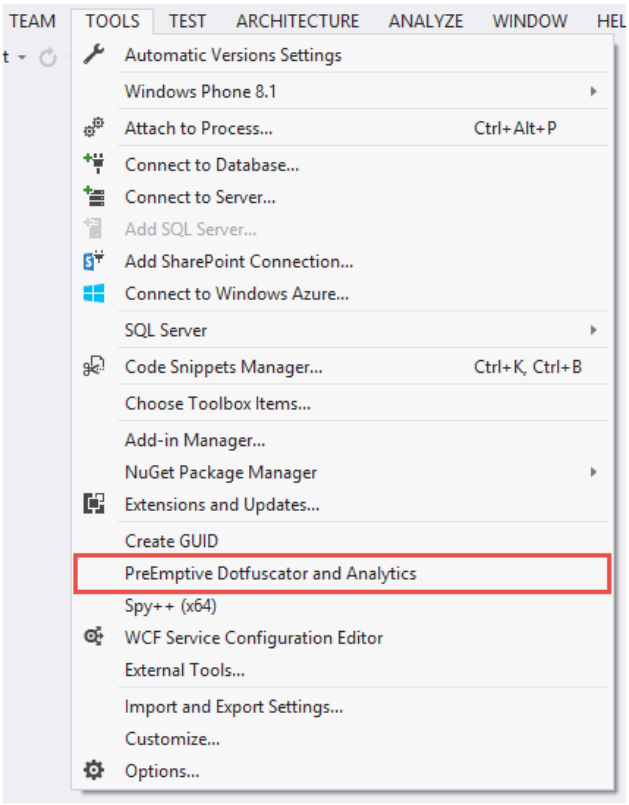
Existem muitos recursos hoje em dia que conseguem “Descompilar”, ou seja, fazer uma engenharia reversa com os códigos intermediários, com o intuito de chegar no Código-Fonte original.

Existem softwares gratuitos que conseguem fazer essa engenharia reversa de uma forma fácil, caso não haja um ofuscamento em seu código.

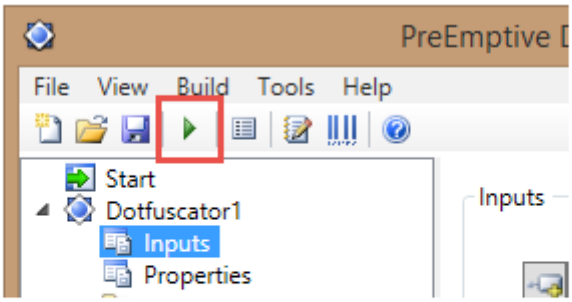
Para tentar desviar destas ferramentas, foram e ainda são criados artifícios que ofuscam, ou seja, “criptografam” seu código intermediário para ficar mais difícil de se fazer esta engenharia reversa. Porém, inevitavelmente, não é possível deixar um código inviolável.

15.3 Exemplo de Ofuscamento no C#

Para começar, basta, no projeto que está desenvolvendo entrar em Tools →



Ao abrir o DotOfuscator, você verá uma interface. Ao clicar para adicionar um novo arquivo a ela, você deverá colocar arquivos de extensão .exe e .dll para serem ofuscados. Estes são os códigos intermediários que serão transformados, e quando desofuscados, seu código fonte estará quase que ilegível. Para ofuscá-lo, basta dar um build nestes arquivos.

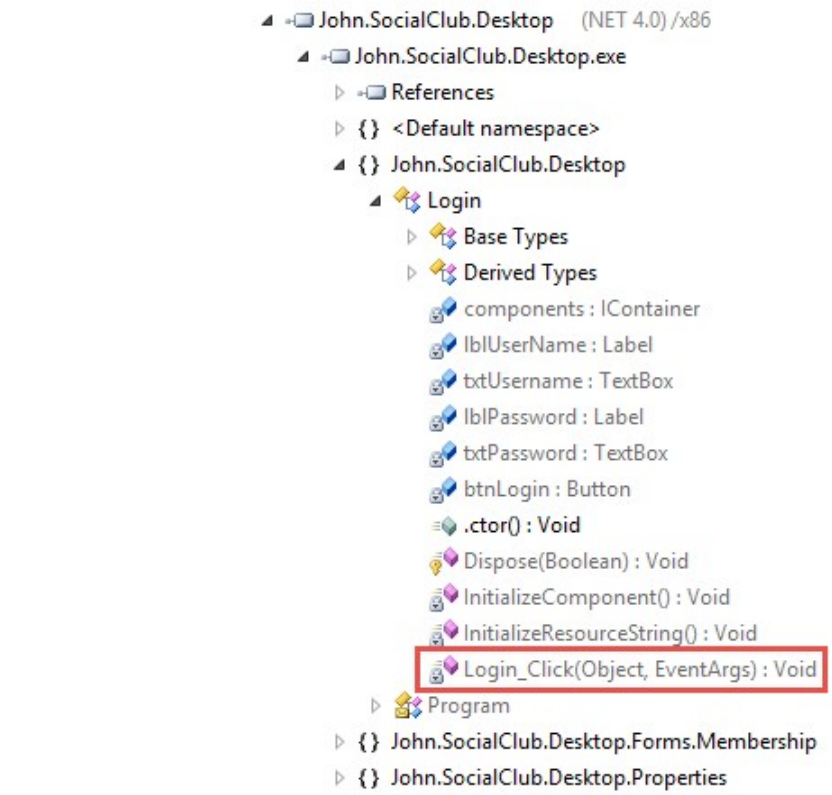



```
Build Finished.
Build Statistics     Total  Renamed  Percent Renamed
Types:                  16        1           6,25 %
Methods:               134       79          58,96 %
Fields:                103       85          82,52 %
```

O assemblie descompilado será salvo na pasta DotOfuscated.

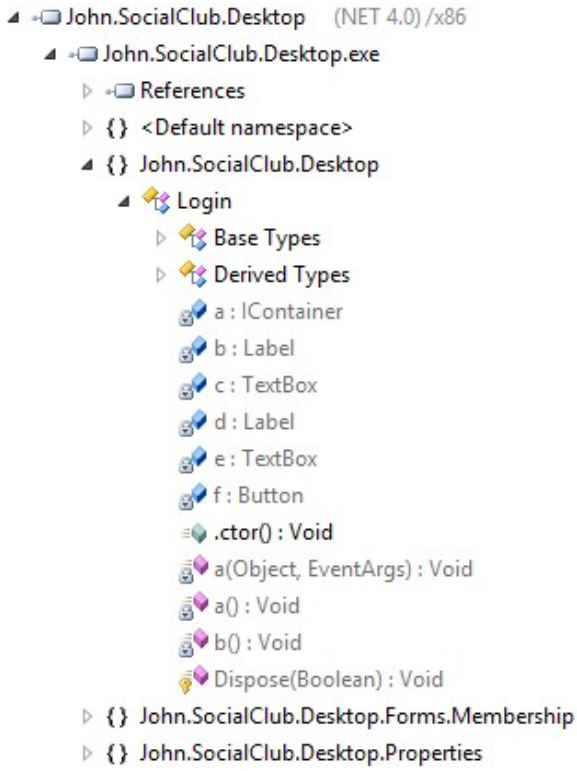
Podemos ver um exemplo de código ofuscado e desofuscado aqui:

Código original:




```
private void Login_Click(object sender, EventArgs e)
{
    if ((this.txtUsername.Text.Trim() != Settings.Default.Username ? true : !(this.t
    {
        MessageBox.Show(Resources.Login_Validation_Message, Resources.Login_Validati
    }
    else
    {
        (new Manage()).Show();
        base.Hide();
    }
}
```

Código ofuscado:

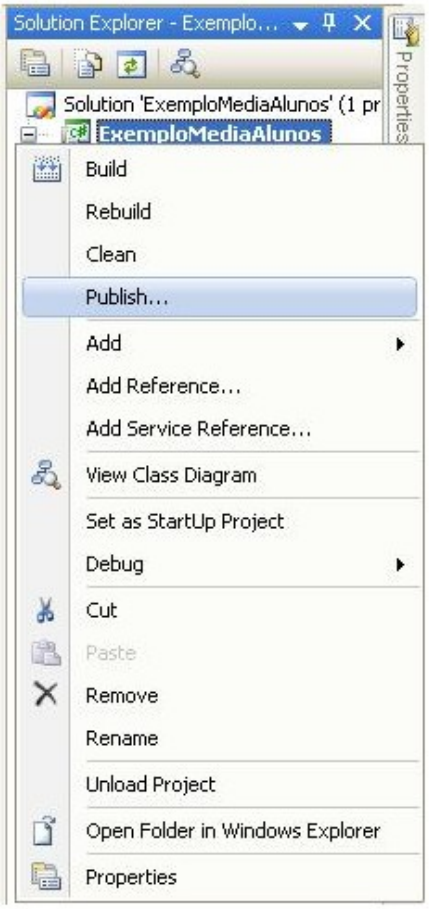



```
private void a(object A_0, EventArgs A_1)
{
    if ((this.c.Text.Trim() != Settings.Default.Username ? true : !(this.e.Text.Trim()
    {
        MessageBox.Show(Resources.Login_Validation_Message, Resources.Login_Validation_
    }
    else
    {
        (new Manage()).Show();
        base.Hide();
    }
}
```

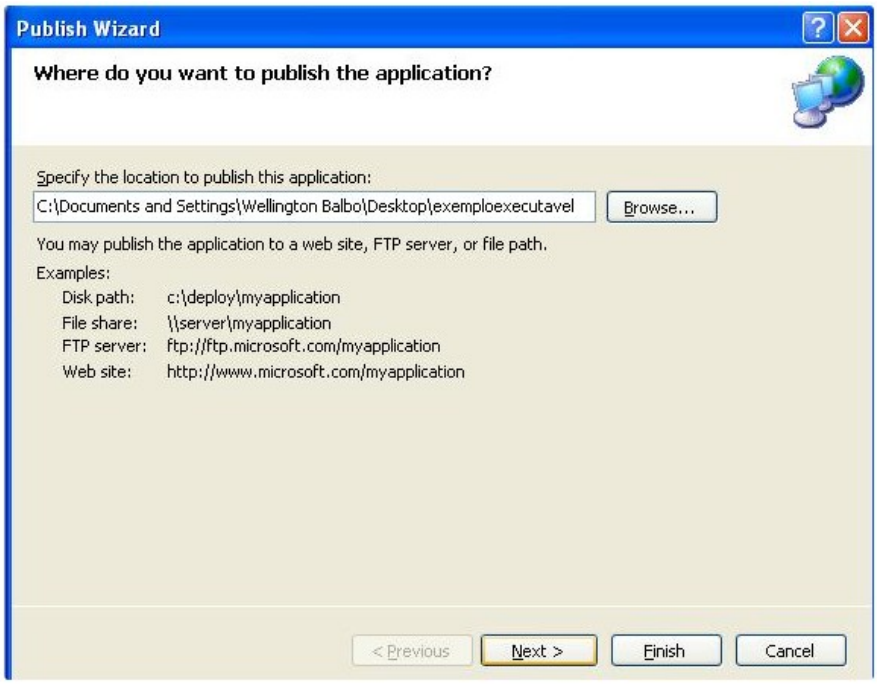
Exemplo de código retirado de: http://www.andrealveslima.com.br/blog/index.php/2016/05/18/como-podemos-obfuscar-aplicacoes-net/

16. Criando Executável

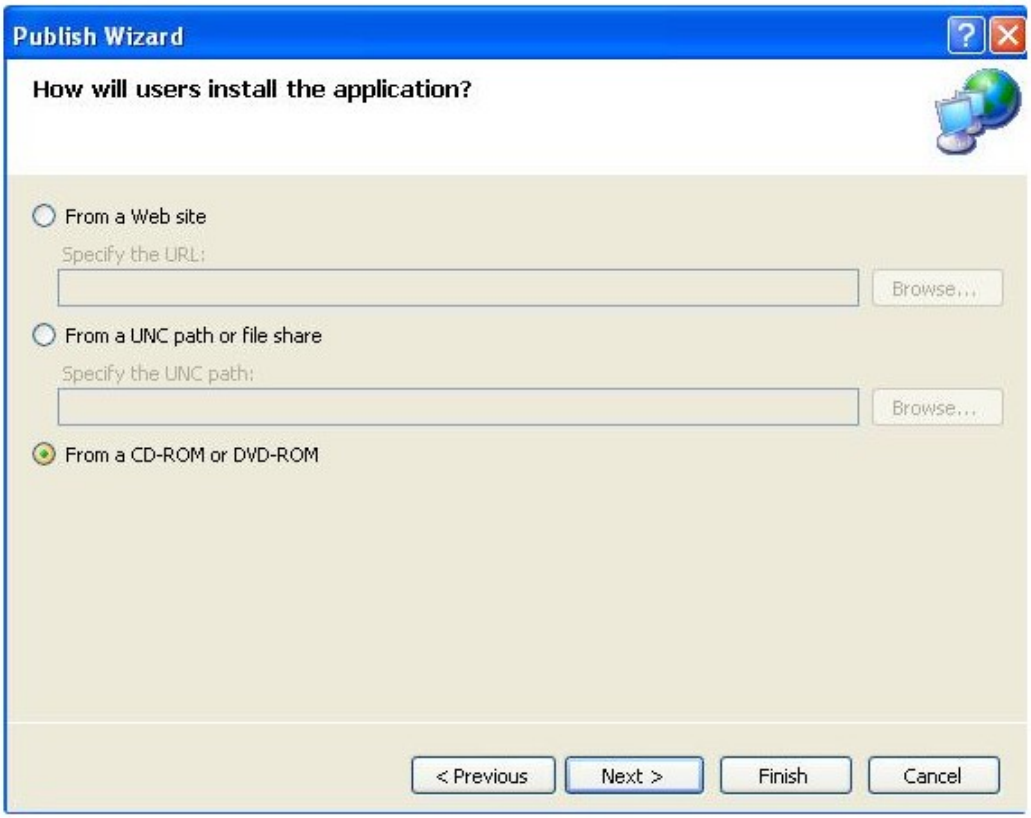
Para criar um executável do projeto, abra o Solution Explorer, clique com o botão direito no projeto e vá na opção Publish para que seja aberto o Wizard.



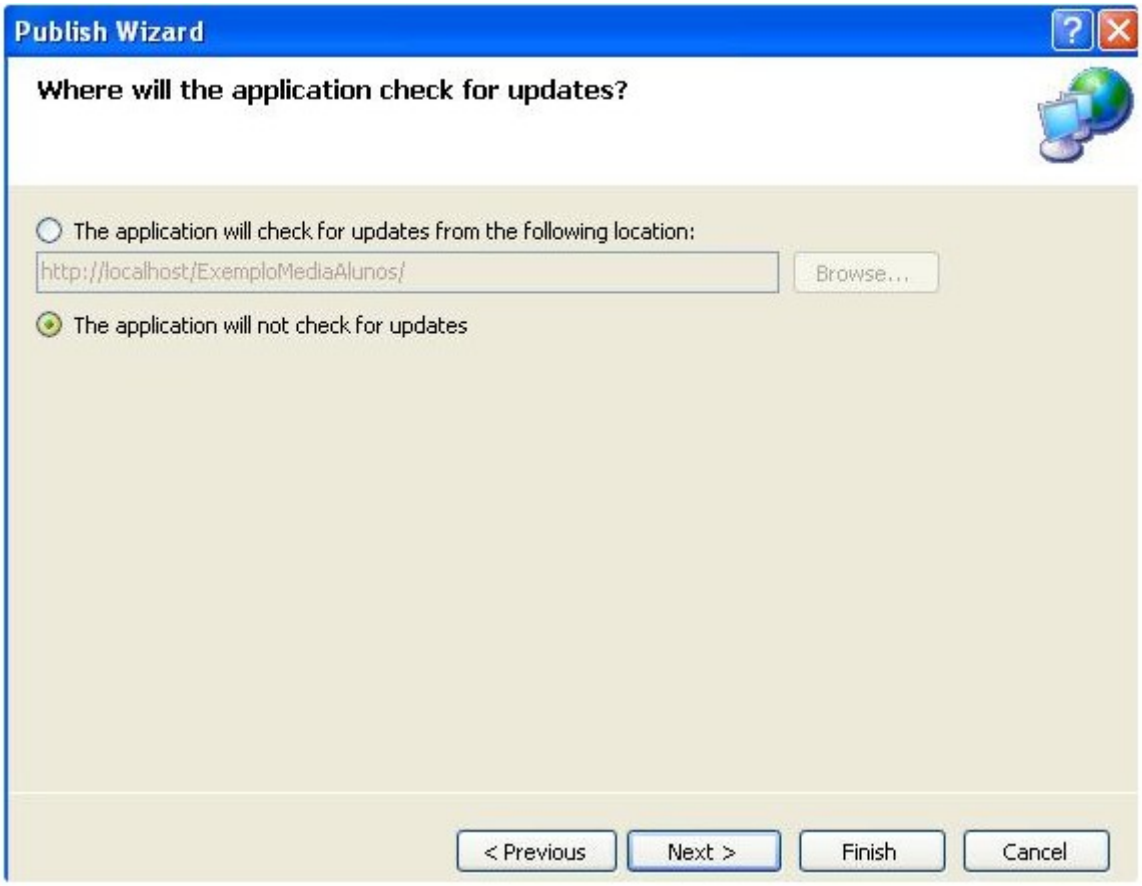
Na tela que abre, clique em Browse e selecione uma pasta em que será gerado o instalador ou crie uma se for necessário. Após isso, clique em Next.



Agora o Wizard irá perguntar como os usuários instalarão sua aplicação. Selecionaremos a opção From a CD-ROM ou DVD-ROM para a demostração.

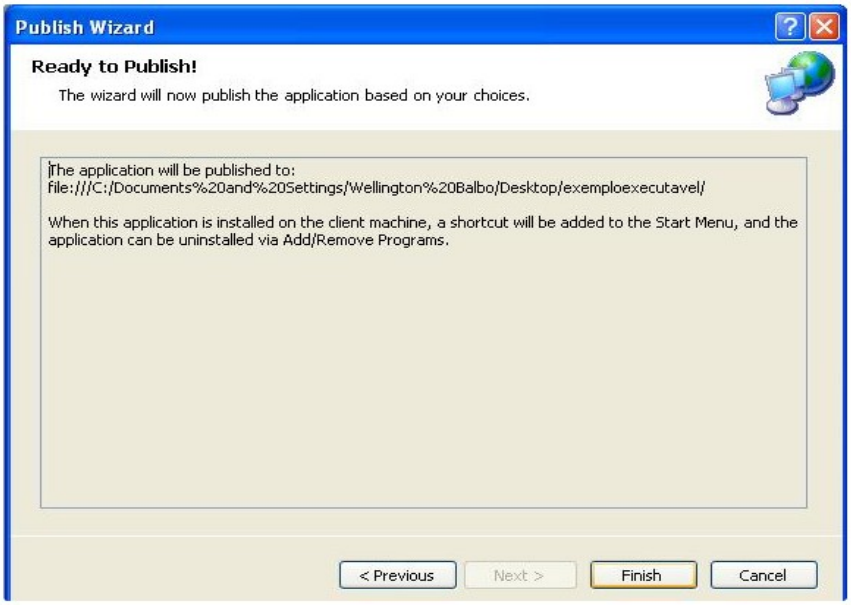


Na próxima tela o Wizard irá perguntar se a aplicação irá procurar por atualizações. Selecionaremos a opção The application will not check for updates depois clique em Next.



Será exibida uma tela avisando que quando o executável for instalado na máquina do cliente, será gerado um atalho no menu **Iniciar** e a opção de Desinstalar o Aplicativo no **Adicionar/ Remover Programas**, do **Painel de Controle**.

Clique em **Finish** e aguarde o **Wizard** gerar seu instalador.



Quando terminar, será mostrada abaixo no canto esquerdo a mensagem **Publish Suceeded**.

Se seu browser abrir tentando carregar o executável e der erro, ignore.

Agora vá na pasta em que foi gerado o executável e dê dois cliques no arquivo Setup.exe. Na tela que aparece, clique em **Install**:

Pronto, o programa foi instalado e pode ser visto no menu Inciar e em Todos os programas.