Programação Orientada a Objetos em .NET

por Claudio Lassala
matérias publicada na MSDN Magazine

Qual é a importância de objetos no mundo do .NET? O que devo saber sobre a Programação Orientada a Objetos (POO) para poder criar programas em .NET? Estas são perguntas comuns entre muitos desenvolvedores que dão seus primeiros passos no mundo do .NET. Entretanto, muitos autores tentam explicar a POO utilizando exemplos tão complicados ou superficiais que o leitor acaba por não entender nem o exemplo e nem o conceito explicado (que por sinal é o mais importante!). Neste artigo dividido em duas partes procurarei trazer ao leitor exemplos simples, visando o entendimento claro dos principais conceitos da POO. Apesar de manter o foco em linguagens .NET, grande parte dos conceitos pode ser aplicada para qualquer linguagem orientada a objetos.

A importância dos Objetos

No mundo do .NET, objetos são fundamentais e o desenvolvedor deve entendê-los, por um simples motivo: tudo, isto mesmo, tudo, é objeto em .NET. Mesmo strings ou inteiros (integer) são objetos. Tente por exemplo, compilar o código a seguir. Apesar de não obter auxílio do IntelliSense ao digitar "." (ponto) após o inteiro 3, e portanto não ser possível ver uma lista de métodos disponíveis, o código é perfeitamente aceitável em .NET, e irá compilar e executar sem problemas.

```
// Em C#:

string teste = 3.ToString();



‘ Em VB.NET:

Dim teste as String = 3.ToString()
```

Mesmo funções disponíveis em VB.NET que aparentemente são funções procedurais estão lá apenas para compatibilidade com versões anteriores da linguagem. Isto significa que ao usar a função MsgBox, por exemplo, o código compilado irá chamar

um método em um objeto, ao invés de usar uma função procedural. O mesmo ocorre com os “módulos” existentes em VB.NET: os mesmos serão compilados como classes.
Baseando-se nestes fatos, pode-se então dizer que objetos são realmente importantes, e o desenvolvedor precisa realmente entendê-los para ter fluência na leitura e escrita de código em .NET.

Qual é a vantagem?

Outra pergunta freqüente é sobre a vantagem em se programar utilizando objetos. Qualquer pessoa ou empresa que vive da criação de software sem dúvidas não fica contente ao ver-se escrevendo código similar infinitamente, gastando tempo e recursos na programação de rotinas que já foram criadas anteriormente, mas que pela falta de uma metodologia apropriada, não podem ser reutilizadas ou customizadas para suprirem necessidades específicas de cada cenário. Resumindo, as principais vantagens da POO são o reuso de código e a capacidade de expansão do mesmo.
Neste momento ainda não começamos a dissecar os conceitos da POO, e por isso é natural que as vantagens da POO não sejam tão aparentes.

Os alicerces da POO

Qualquer linguagem orientada a objetos deve oferecer suporte aos seguintes conceitos da POO:

- Abstração
- Encapsulamento
- Herança
- Polimorfismo

A incapacidade de uma linguagem em suportar qualquer um destes conceitos a desqualifica como uma linguagem orientada a objetos. Sendo estes os alicerces da POO, vamos dissecar cada um deles a seguir.

Abstração

Abstração pode ser definida como a capacidade de representar cenários complexos usando termos simples.
Pode-se tomar um exemplo da vida real para ilustrar o conceito: um carro é uma abstração de um veículo que um indivíduo pode utilizar com o objetivo de mover-se de um ponto a outro. No dia-a-dia, ninguém dirá: “Vou abrir a porta daquele veículo

movido a combustível, entrarei, me sentarei, darei a partida no motor, pisarei na embreagem, engatarei a primeira marcha, acelerarei, controlarei a direção em que o carro irá se mover utilizando o volante". Tamanha explicação não se faz necessária pois todo o conceito daquilo foi abstraído para algo que conhecemos como "carro". Apesar de um carro ser algo bastante complexo, basta dizer "vou usar o meu carro para ir ao trabalho amanhã", e qualquer pessoa entenderá o recado.

Da mesma forma, imagine a confusão para qualquer pessoa entender a seguinte frase: "Quando abro aquele programa em meu computador, surge uma tela que tem várias caixas retangulares, normalmente brancas, nas quais eu posso clicar dentro e começar a digitar. Algumas dessas caixas já trazem algo escrito, e outras aparecem completamente vazias". Seria muito mais fácil substituir toda esta explicação apenas dizendo "O programa tem uma tela com diversos TextBox". Mais uma vez, o complexo objeto que se parece com uma caixa retangular, e permite ao usuário digitar dentro dela (além de possuir outros atributos e ações), foi sabidamente abstraído para a palavra TextBox. O mesmo ocorre com botões de comando, caixas de seleção, grupos de opção, etc. Todos eles são objetos complexos abstraídos para simples termos que todo desenvolvedor utiliza quando precisa referenciar a tais objetos.

Além de objetos como aqueles que possuem representação visual (TextBox, Button, Form, etc.), existem também objetos que são criados em muitas aplicações com o intuito de abstraírem objetos complexos da vida real, como por exemplo Pedido, Cliente, Produto, e muitos outros. Mas veremos exemplos destes objetos mais adiante.

Neste ponto é importante compreender a diferença entre classe e objeto. Quando mencionado que algo complexo foi abstraído para algo conhecido como TextBox, este algo é uma classe. Uma classe é um modelo para algo que deve ser criado. Por exemplo, quando alguém vai fazer um bolo de chocolate, pega-se uma "Receita para Bolo de Chocolate", que será usada como um modelo para o bolo que será criado. De forma análoga, a receita é uma classe, e o bolo é um objeto.

Uma classe não é utilizada diretamente. Ninguém come a receita de bolo. A classe é utilizada somente para criar objetos baseados nela, e são os objetos que serão realmente utilizados. Deste modo, não utiliza-se a classe TextBox diretamente, e sim os objetos criados a partir daquele classe (caso membros estáticos ou compartilhados venham à mente do leitor, no último parágrafo desta seção procuro esclarecer uma confusão que pode surgir com o conceito). A Figura 1 ilustra este conceito. No lado esquerdo da figura vê-se um modelo da classe TextBox, listando alguns de seus membros, como as propriedades Text, Location, Size, e alguns métodos como Focus e Copy. Ao lado direito, vê-se um Form que faz uso de objetos baseados na classe TextBox.

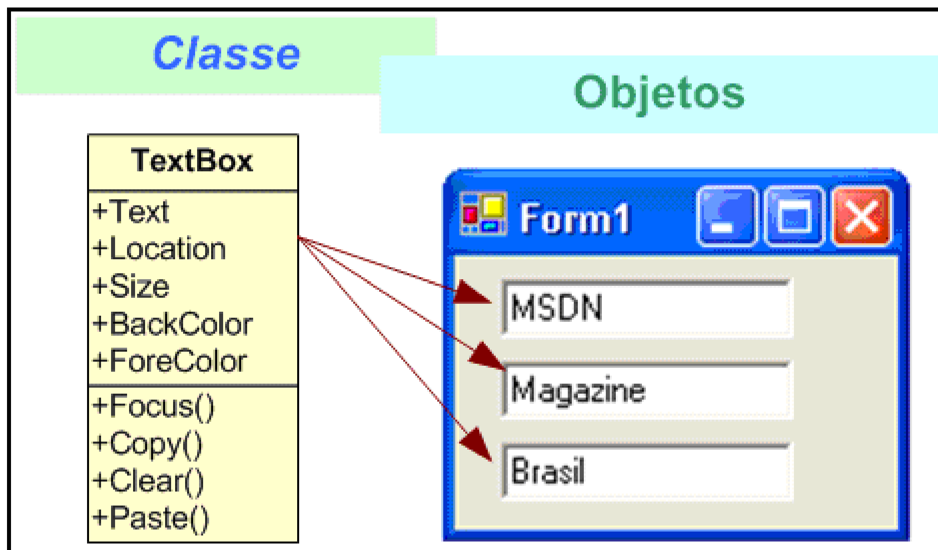

Figura 1. Classes e Objetos

Por definição, afirma-se que um objeto é a instância de uma classe. Deste modo, na Figura 1 pode-se ver que o Form possui três instâncias da classe TextBox. TextBox é um exemplo de classe disponível nativamente no .NET Framework. Em muitos casos é necessário criar classes que não existem no framework, como por exemplo uma classe Produto. Criar classes é algo muito simples tanto em VB.NET como em C#. O código a seguir mostra a sintaxe necessária em ambas as linguagens.

```
// Em C#:

public class Produto

{

}



‘ Em VB.NET:

Public Class Produto

End Class
```

Neste ponto estou omitindo propositalmente a criação de membros para as classes, como propriedades ou métodos, apenas para manter a clareza do exemplo. Neste artigo existem seções dedicadas a membros de classes. Instanciar classes (ou criar objetos) é também bastante simples, como pode-se ver no seguinte código:

```
// Em C#:

Produto meuProduto = new Produto();



' Em VB.NET:

Dim meuProduto as Produto = New Produto()
```

O leitor familiarizado com membros compartilhados ou estáticos (shared em VB.NET, static em C#, respectivamente) pode questionar minha afirmação de que classes não são utilizadas diretamente, visto que tais membros (compartilhados ou estáticos) são utilizados sem a necessidade de instanciar a classe. Isto é parte verdade, uma vez que o programador não precisa instanciar a classe manualmente. Entretanto, o runtime do .NET irá instanciar a classe e controlar aquela instância nos bastidores, então vale a tese de que uma classe é utilizada através de algum objeto instanciado a partir da mesma.

Encapsulamento

Encapsulamento pode ser definido como a tarefa de tornar um objeto o mais auto-suficiente possível.

Na seção anterior, quando o exemplo do carro foi mencionado, de maneira intuitiva deixou-se de abordar os detalhes técnicos de como o motor de um carro funciona. Existem tantos detalhes a serem explicados sobre o funcionamento do motor de um carro que a maioria das pessoas provavelmente desistiriam de utilizar um carro caso precisassem entender como funciona cada parte daquilo. Basicamente, uma pessoa não precisa daquele entendimento. Apenas saber que o motor é o dispositivo que propulsiona o carro já é o suficiente. A maior parte dos detalhes do motor está "encapsulada" dentro do mesmo, e pessoas utilizando o carro não precisam lidar diretamente com o motor.

Além disso, é importante frisar que o motor de um carro tem um funcionamento independente das outras partes (tais como direções, pneus, bancos, etc.). Ou seja, mesmo que um carro não tenha as quatro rodas, isto não impede de o motor funcionar, pois ele funciona como uma unidade independente, uma "caixa-preta".

Tomando agora o exemplo da caixa de texto em uma tela de um programa, pode-se refletir sobre os diversos detalhes que estão encapsulados dentro daquele objeto. Por exemplo, o desenvolvedor não sabe exatamente (ou não precisa saber) como é que o

sistema operacional irá desenhar a representação visual daquele objeto na tela, mandando sinais da CPU para a placa de vídeo, e então para o monitor, e então criando o objeto na posição que foi especificada previamente. Felizmente o desenvolvedor não precisa se preocupar com este tipo de detalhe. Tudo o que é preciso é colocar a caixa de texto na tela e configurar as propriedades Top e Left, para determinar onde o controle deverá aparecer. A mágica do como o controle irá aparecer na tela é algo que o objeto é auto-suficiente o bastante para fazer aquilo sozinho.
Os conceitos de Abstração e Encapsulamento andam de mãos dadas, visto que com a abstração definimos a entidade que representa um objeto complexo, e o encapsulamento esconde detalhes daquele objeto, e com isto esconde detalhes de seu funcionamento que poderiam assustar a qualquer pessoa tentando utilizá-lo.
Imagine se o motorista tivesse que saber a quantidade exata de combustível a ser inserida no carburador quando o acelerador do carro é acionado? Acho que eu só andaria de ônibus, ou de carona.

Herança

Herança pode ser definida como a capacidade de uma classe herdar atributos e comportamento de uma outra classe.
Basta um passeio ao redor da cidade para descobrir-se que existem vários tipos e modelos de carros lá fora. Carros de passeio, carros de corrida, carros conversíveis, carros com volante do lado esquerdo e outros do lado direito, carros de diferentes cores, fabricantes, etc. Cada vez que um novo carro é criado, não é necessário "reinventar a roda", e começar todo um projeto novamente. Diversos conceitos abstraídos e encapsulados no objeto carro já foram planejados, testados e aprovados, e por isso não faz sentido gastar-se tempo e recursos para fazer tudo aquilo de novo.
Todo carro terá um meio de propulsão, direção, acelerador e freio, rodas e pneus, e deste modo, todo novo carro pode herdar estas características e funcionamento de um projeto já existente. Caso seja necessário criar um novo carro que seja quase igual a todos os outros, mas que possua algumas coisas diferentes (por exemplo, utilizando um câmbio automático ao invés de manual), o projetista pode herdar o novo carro a partir do projeto básico, e então adicionar as características específicas do novo carro ao novo projeto.
Voltando ao exemplo da caixa de texto: a classe-base da caixa de texto é retangular e geralmente tem um fundo branco e os caracteres mostrados em preto. Isto é suficiente para diversos cenários, mas imaginando-se o cenário onde faz-se necessária a inclusão de uma caixa de texto que deverá receber somente a entrada de números

(proibindo a digitação de letras), e deverá mostrar o número em azul caso seja positivo e vermelho caso seja negativo, logo percebe-se a necessidade em se criar uma nova classe. Esta nova classe será bastante semelhante à caixa de texto padrão. De fato, semelhante o suficiente para que a mesma seja derivada da caixa de texto padrão, e então as necessidades específicas serão definidas nesta subclasse. A Listagem 1 mostra uma possível implementação em C# para tal classe. A principal diferença para o código em VB.NET seria apenas sintaxe, mas os conceitos são os mesmos.

Listagem 1. Exemplo de herança

```
public class ValorTextBox : System.Windows.Forms.TextBox

{



 protected override void OnTextChanged(EventArgs e)

 {

   base.OnTextChanged (e);



   string valor = this.Text.Trim();



   if (valor.Length == 1 && valor == "-")

   {

     // apenas o sinal de negativo foi digitado

     this.ForeColor = Color.Red;

   }

   else

   {

     try

     {
```

```
      // Tenta-se converter a string em um valor decimal
      decimal val = Convert.ToDecimal(valor);


      // Se tiver sucesso, formatar o TextBox para números
      // positivos e negativos.
      if (val >= 0)
        this.ForeColor = Color.Blue;
      else
        this.ForeColor = Color.Red;
    }
  catch
    {
      // Se não foi possível converter a string em decimal,
      // cancela o último caracter digitado pelo usuário
      if (valor.Length > 0)
        this.Text = valor.Substring(0, valor.Length-1);
      this.SelectionStart = this.Text.Trim().Length;
    }
  }
 }
}
```

A primeira linha de código determina que a nova classe chama-se ValorTextBox, e deriva da classe System.Windows.Forms.TextBox. Em VB.NET, troca-se os dois-

pontos (“:”) pela palavra-chave Inherits. Todas as características da classe-base são herdadas ao definir-se que a nova classe herda da mesma. O restante do código apenas implementa o comportamento de validar a entrada de dados do usuário e a formatação diferenciada para valores positivos e negativos.

Com esta nova classe criada, o próximo passo é apenas utilizá-la. O código a seguir mostra trechos de código em um Form que utiliza tanto esta nova classe como a classe básica disponível no Framework. Os principais pontos a enfatizar no código a seguir são a declaração dos campos que mantém as instâncias dos TextBox e a criação dos objetos baseados nas classes específicas. Pode-se ver que é criado um TextBox baseado na classe padrão, e dois outros TextBox baseados na classe especializada ValorTextBox criada neste exemplo.

```
public class Heranca : System.Windows.Forms.Form

{

  private System.Windows.Forms.TextBox textBox1;

  private MSDNBrasil.ValorTextBox txtValor1;

  private MSDNBrasil.ValorTextBox txtValor2;



  // código omitido para clareza do exemplo

}

private void InitializeComponent()

{

  this.textBox1 = new System.Windows.Forms.TextBox();

  this.txtValor1 = new MSDNBrasil.ValorTextBox();

  this.txtValor2 = new MSDNBrasil.ValorTextBox();



  // código omitido para clareza do exemplo

}
```

O Form em execução pode ser visto na Figura 2. Os dois primeiros TextBox mostram nossa classe especializada em ação, enquanto que o terceiro é um TextBox básico. O importante ponto a frisar na utilização da herança é que a classe é criada uma vez, e então utilizada em quantos Forms forem necessários. Supondo que em algum momento seja necessário alterar a aparência e/ou comportamento de todos os TextBox que utilizam esta classe, basta alterar diretamente na classe e todos os objetos refletirão a alteração. Por exemplo, ao invés de mostrar os números negativos em cor vermelha, agora o usuário deseja ver os números em letras amarelas, e a cor-de-fundo do TextBox deve passar a ser vermelha.

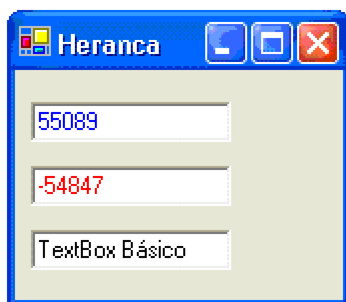

Figura 2. Herança em ação

Existem alguns termos utilizados freqüentemente, e o leitor precisa estar familiarizado com eles. Chama-se subclasse a classe criada derivada de uma outra classe, a qual por sua vez é chamada superclasse, ou classe-base. Diz-se também que a subclasse é uma especialização de sua superclasse, ou então que a superclasse é uma generalização de suas subclasses.

Além disso, a herança é definida com um relacionamento "é um" (ou é uma). Por exemplo, pode-se dizer que o homem é um ser humano, ou que um calhambeque é um carro, assim como nossa nova caixa de texto especialmente criada para números é uma caixa de texto.

A Figura 3 mostra uma representação gráfica destes termos. A seta preta que conecta a classe Homem (SubClasse) à classe Humano (SuperClasse) indica que Homem é um Humano. A verde indica que a classe Homem é uma especialização da classe Humano, e a seta azul indica que a classe Humano é uma generalização da classe Homem.

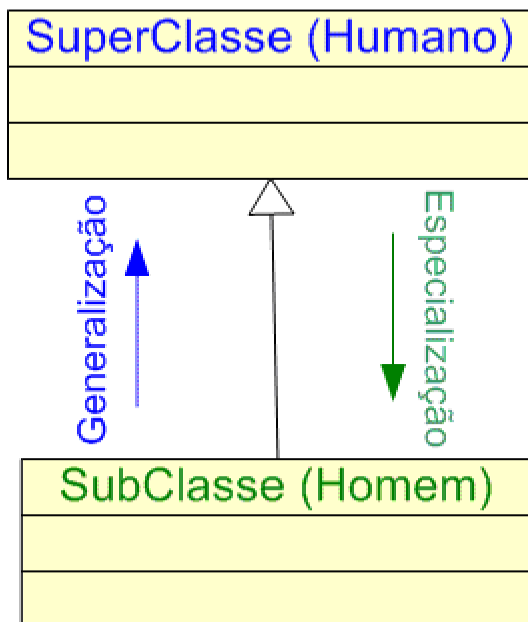

Figura 3. Ilustrando alguns termos importantes

Polimorfismo

Literalmente, polimorfismo significa “muitas formas” ou “o estado de existir em muitas formas”. No mundo da programação, polimorfismo é muitas vezes definido como a “capacidade de objetos diferentes possuírem métodos de mesmo nome e mesma lista de parâmetros que quando chamados executam tarefas de maneiras diferentes”.
Estas definições em nada ajudam ao desenvolvedor que está tentando desmistificar o paradigma da POO. E sinceramente não conheço nenhuma forma para se explicar polimorfismo usando poucas palavras, por isso sempre procuro usar bastante analogias e exemplos diversos para ilustrar a explanação.
Seguindo com o exemplo do carro, pode-se dizer que carro “é-um” veículo. Pode-se também dizer que moto “é-um” veículo, e o mesmo vale para bicicleta. Todo veículo possui algum meio para acelerar, não importa qual mecanismo é usado para isso, e o mecanismo geralmente é diferente, principalmente se comparado um carro (que usa um motor) à uma bicicleta (que usa pedais, corrente e coroa). Colocado em outras palavras, temos objetos diferentes (carro, moto, bicicleta), que derivam de uma mesma classe (veículo). Esta classe possui um método Acelerar, e deste modo, podemos utilizar os diferentes objetos de modo polimórfico, invocando um método de mesmo nome, mas que possui implementação e comportamento diferente em cada um dos objetos.
Um típico exemplo utilizado na explicação de polimorfismo é a criação de formas geométricas, ou qualquer outro objeto que crie algum tipo de desenho na tela.

Considere uma classe base chamada Forma, a qual representa qualquer forma que deverá ser desenhada na tela. Esta classe é mostrado no seguinte código:

```
public abstract class Forma

{

  public Forma()

  {

  }



  public virtual void Desenhar(System.Drawing.Graphics g)

  {

  }

}
```

A classe Forma é marcada com a palavra-chave abstract (indicando tratar-se de uma classe abstrata). Classes abstratas são classes que jamais serão instanciadas diretamente pois não possuem implementação sufience para oferecer funcionalidades concretas a serem utilizadas. Neste exemplo, a classe Forma tem o método Desenhar. Este método recebe um objeto do tipo Graphics, o qual será a superfície onde a forma será desenhada. Não há muito que colocar dentro deste método pois cada forma exige código diferente para ser desenhada. É nas classes derivadas da classe Forma que este método será implementado, e por isso o método é marcado como virtual (ou overridable em VB.NET), como pode ser visto no seguinte código, o qual mostra a implementação da classe Circulo.

```
using System.Drawing;

public class Circulo : Forma

{

  public override void Desenhar(System.Drawing.Graphics g)

  {
```

```
    base.Desenhar (g);

    g.DrawEllipse(Pens.Red, 5, 5, 100, 100);

  }

}
```

A classe Circulo herda da classe Forma. Esta é uma classe concreta, o que significa que pode ser instanciada diretamente, visto que esta classe oferece algum tipo de funcionalidade concreta. O método Desenhar é sobrescrito, tendo o código necessário para desenhar um círculo perfeito na superfície passada por parâmetro ao método. Note que o método na superclasse é chamado explicitamente (em C# usa-se base.NomeDoMétodo, enquanto que em VB usa-se MyBase.NomeDoMétodo) . De modo similar, o seguinte código mostra a classe Quadrado.

```
public class Quadrado : Forma

{

  public override void Desenhar(System.Drawing.Graphics g)

  {

          base.Desenhar (g);

          g.DrawRectangle(Pens.Green, 5, 5, 100, 100);

  }

}
```

O código para o método Desenhar em ambas as classes Circulo e Quadrado são bastante parecidos pelo fato de que ambos fazem uso de métodos capazes de desenharem elipses e retângulos. Para tornar o exemplo um pouco mais interessante, o seguinte código mostra a classe Pessoa, a qual terá a funcionalidade de desenhar uma pessoa.

```
public class Pessoa : Forma

{

  public string Falar(string frase)
```

```
  {
              return "Pessoa falando " + frase;
  }

  public override void Desenhar(System.Drawing.Graphics g)
  {
              base.Desenhar (g);
              GraphicsPath pessoa = new GraphicsPath();
              pessoa.AddEllipse(23, 1, 14, 14);
              pessoa.AddLine(18, 16, 42, 16);
              pessoa.AddLine(50, 40, 44, 42);
              pessoa.AddLine(38, 25, 37, 42);
              pessoa.AddLine(45, 75, 37, 75);
              pessoa.AddLine(30, 50, 23, 75);
              pessoa.AddLine(16, 75, 23, 42);
              pessoa.AddLine(22, 25, 16, 42);
              pessoa.AddLine(10, 40, 18, 16);
              g.DrawPath(Pens.Blue, pessoa);
  }
}
```

O método Desenhar da classe Pessoa é um pouco mais complexo pois necessita de mais código para desenhar a representação de uma pessoa na superfície passada ao método. Caso o leitor não esteja familiarizado com GDI+ (que vem sendo utilizado para criar os desenhos neste exemplo), não se preocupe: neste momento, o ponto

importante é salientar que o método Desenhar de cada classe concreta pode ser bastante diferente entre as classes, ou seja, a implementação pode ser completamente diferente entre as diversas classes.

Na classe Pessoa também está definido um método Falar. Isto foi feito apenas para enfatizar o fato de que as subclasses da classe Forma podem ter diferentes membros (métodos, propriedades, etc.), além daqueles herdados da superclasse. Para a visualização deste exemplo, foi criado um Form bastante simples, definido com o código mostrado na Listagem 2 (partes irrelevantes do código foram omitidas para manter a clareza do exemplo).

Listagem 2. Form Visualizador

```
public class Visualizador : System.Windows.Forms.Form

{

  // Mantém instância genérica de uma Forma.

  private Forma forma = null;



  /// Construtor usado para instanciar o Form,

  /// já preparando-o para visualizar a Forma correta.

  /// </summary>

  /// <param name="forma">Forma que será desenhada e visualizado no
Form.</param>

  public Visualizador(Forma forma)

  {

              InitializeComponent();

              // Persiste o objeto passado durante a existência do Form

              this.forma = forma;

  }
```

```
 protected override void OnPaint(PaintEventArgs e)

 {

        base.OnPaint (e);

        // Mostra o nome da classe sendo visualizada.

        this.Text = ((Type)this.forma.GetType()).Name;

        // Invoca o método Desenhar, passando a superfície

        // onde a forma será desenhada.

        this.forma.Desenhar(e.Graphics);

 }

}
```

Neste momento, vamos nos concentrar no segundo Form que foi criado (chamado ExemploPolimorfismo), o qual apenas contém um botão chamado btnExecutar, o qual irá instanciar as diversas Formas e utilizar o Visualizador para mostrar o resultado do método Desenhar de cada uma delas. O código executado quando o botão Executar deste Form é clicado, bem como o método-auxiliar Mostrar são mostrados no seguinte código.

```
private void btnExecutar_Click(object sender, System.EventArgs e)

{

 Circulo circ = new Circulo();

 this.Mostrar(circ);



 Quadrado quad = new Quadrado();

 this.Mostrar(quad);



 Pessoa pes = new Pessoa();
```

```
  this.Mostrar(pes);
}

private void Mostrar(Forma forma)
{
  Visualizador v = new Visualizador(forma);
  v.Show();
}
```

Não há muito segredo no método btnExecutar_Click. As diversas formas são instanciadas e passadas para o método Mostrar, que toma conta de instanciar o Visualizador e passar a forma como parâmetro. O resultado disto é mostrado na Figura 4.

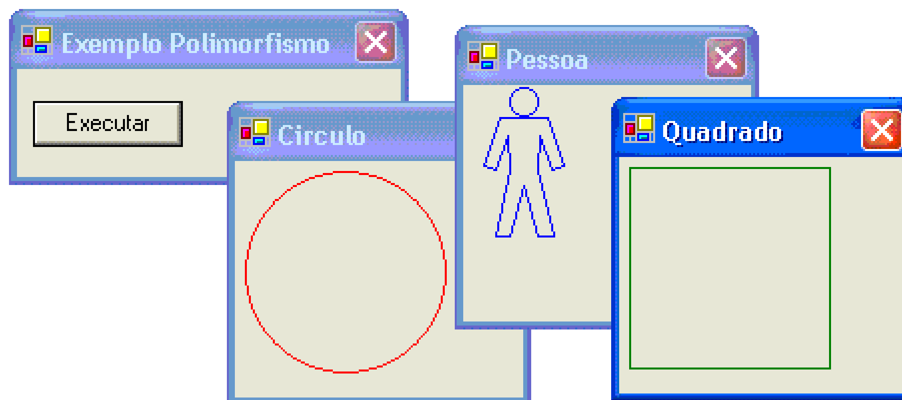

Figura 4. Exemplo de polimorfismo

Vamos voltar um pouco e dissecar o código da classe abstrata Forma. O Form Visualizador é bastante simples. Um campo privado do tipo Forma é declarado na classe com o intuito de manter uma referência genérica para qualquer tipo de forma passada para ser visualizada. Um construtor especializado é definido para receber referência para a forma que se deseja visualizar, e persistir esta referência no campo privado do Form. Note que qualquer tipo de forma pode ser passada (um círculo, um quadrado, uma pessoa, ou qualquer outra forma que venha a ser criada), e por isso é necessário declarar o tipo através do mínimo denominador comum entre estas classes, que neste caso, é a superclasse Forma.

O método OnPaint (que ocorre a toda vez que o Form é desenhado na tela) recebe um parâmetro do tipo PaintEventArgs, e este parâmetro possui um membro chamado Graphics, e é este membro que é passado para o método Desenhar das classes derivadas da superclasse Forma.
A linha de comando ((Type)this.forma.GetType()).Name (em VB.NET seria CType(this.forma.GetType()).Name) é usada apenas para mostrar o nome da classe na barra de título do Form.
A linha de comando this.forma.Desenhar(e.Graphics) mostra o uso do polimorfismo: diversos tipos diferentes de objetos (círculos, quadrados, pessoas, etc.) podem ser passados como parâmetro para este método pois todos estes objetos derivam da classe Forma. Então, é válido chamar este método passando um círculo, pois círculo "é-uma" forma, e Forma é o tipo que o método está esperando receber. Apesar de chamar o mesmo método (Desenhar) e passar o mesmo tipo de parâmetro (Graphics), o resultado de tal operação pode ser bastante diferente, como é mostrado na Figura 4. Neste ponto, o uso de polimorfismo é possível porque as classes Circulo, Quadrado e Pessoa compartilham algo em comum: todas elas derivam da superclasse Forma, a qual define o método Desenhar, que neste exemplo, vem sendo usado de maneira polimórfica. Entretanto, existem cenários onde diferentes objetos precisam ser usados de maneira polimórfica, mas infelizmente eles não são derivados da mesma superclasse.
Suponha que uma nova classe Funcionário seja adicionada a este exemplo. Esta é uma classe de negócios, e portanto, herda de uma superclasse Negócios (em um cenário real, uma classe deste tipo abstrai operações normais envolvidas com negócios, como lidar com a validação de dados do objeto, por exemplo). Esta classe possui um campo do tipo Bitmap que armazena uma foto do funcionário. Até então, as classe Funcionário e Negócios foram definidas como o código a seguir.

```
// Classe base para objetos de negócios.

public abstract class Negócios

{

  // Método para validação dos dados mantidos pela classe.

  public abstract void Validar();

}
```

```
public class Funcionário : Negócios

{

  // Campo que contém foto do funcionário. Em cenário real,

  // possivelmente traríamos a foto a partir de um banco de dados ou algo assim.

  public Bitmap Foto = new Bitmap(@"e:\ClaudioLassala.jpg");



  public override void Validar()

  {

             // Código para validação dos dados.

  }

}
```

Surge a necessidade de desenhar a foto do funcionário em uma superfície, como foi feito com as formas circulo, quadrado e pessoa anteriormente. Por ser uma classe de negócios, a classe Funcionário não pode herdar de qualquer outra classe senão a classe Negócios. Isto significa que a classe Funcionário não possui um método Desenhar, e portanto não compartilha nada em comum com as classes que sabem como "desenhar a si mesmas".

Como este caso pode ser resolvido? Antes de se precipitar e responder "basta adicionar um método Desenhar à classe Funcionário", devo mencionar que isto não fará com que a classe passe a compartilhar algo em comum com as classes circulo e quadrado, pois ela continua não herdando da mesma superclasse, e adicionar um método que porventura tenha o mesmo nome e assinatura (no caso do método Desenhar) não irá resolver o problema. A classe pode até ser instanciada e ter seu próprio método Desenhar sendo chamado diretamente, contudo não será possível utilizar à classe genérica Visualizador, pois lembre-se, aquela classe espera por um objeto que "seja-uma" Forma, o que não é o caso do Funcionário.

Aqui entra o conceito da Interface. Além de classes que "são" subclasses de uma mesma superclasse, pode-se usar polimorfismo com classes que são capazes de fazer algo em comum. Neste cenário, é necessário utilizar objetos que sejam capazes de desenhar algo que os represente.

Interface é a forma pela qual um objeto permite que outros objetos interajam com ele. Isto é muito diferente da “interface visual”, que é a forma como um ser humano pode interagir com objetos visuais, tais quais um Textbox, Button, CheckBox, etc.
Como já foi mencionado, a herança é um relacionamento do tipo “é-um”, onde dizemos que uma classe “é-um” tipo de outra classe. Interfaces, por outro lado, permitem criar um relacionamento entre objetos que não seria possível caso os objetos não herdassem da mesma superclasse. Ao utilizar uma interface, define-se “o que uma classe é capaz de fazer”, ao invés de definir “o que uma classe é” (como no caso da herança).
Para resolver o cenário proposto, foi criada uma interface chamada . É convencionado nomear classes começando com um “I” maiúsculo (indicando “interface”), e seguindo com uma ou mais palavras que definam qual é a capacidade garantida por aquela interface. Neste caso, objetos que implementam esta interface podem ser interpretados como “classes desenháveis” (este padrão é usado no .NET Framework, como nas interfaces IDisposable, IComparable, IEnumerable, etc.). Veja no código a seguir a interface .

```
public interface

{

  void Desenhar(Graphics g);

}
```

A interface possui apenas um membro, que é o método Desenhar. Todos os membros de uma interface serão sempre públicos pois uma interface é como um contrato, e toda classe implementando esta interface deve saber exatamente quais são as “cláusulas” (membros) deste contrato para que possa satisfazer a todos os detalhes.
O próximo passo é alterar as classes Funcionário e Forma, de modo que estas classes implementem a interface . O código seguinte mostra em negrito as principais alterações (partes irrelevantes com relação à interface foram omitidos pra manter a clareza do exemplo).

```
public class Funcionário : Negócios,

{

  #region Membros da Interface

  public void Desenhar(Graphics g)

```

```
  {
              // Carrega a foto em um objeto Image.
              Image im = Image.FromHbitmap(this.Foto.GetHbitmap());


              // Desenha a foto na superfície do gráfico.
              g.DrawImage(im, 1, 1, im.Width, im.Height);
              }
  #endregion
}


public abstract class Forma :
{
  public virtual void Desenhar(System.Drawing.Graphics g)
  {
  }
}
```

Note que a classe Funcionário, além de herdar da classe Negócios, agora também implementa a interface. Em .NET, herança-múltipla não é suportada, portanto cada classe pode herdar de apenas uma superclasse, mas no entanto pode implementar qualquer número de interfaces. Visto que a interface possui o método Desenhar, a classe Funcionário passa automaticamente a ser obrigada a implementar tal método (naquele método está o código necessário para utilizar o bitmap que possui a foto do funcionário e desenhar aquela imagem na superfície). Caso deixe de implementar o método, um erro será acusado pelo compilador, o que proibirá a compilação do código.
No caso da classe Forma, foi adicionada a informação de que aquela classe implementa a interface, e como já existia um método Desenhar em conformidade com

aquele definido na interface, nada mais precisou ser alterado. Lembre-se que as classes Circulo, Quadrado e Pessoa herdam da classe Forma, e portanto, não é necessário alterar àquelas classes, pois a informação de quais interfaces são implementadas também é herdada para as subclasses.

Outra importante alteração é a mudança do tipo utilizado no Visualizador e no Form ExemploPolimorfismo. Basicamente, ao invés de fazer referências a objetos do tipo Forma, passa-se a referenciar a objetos que implementam. A Listagem 3 mostra em negrito as alterações em ambos os Forms (mais uma vez, código irrelevante neste contexto foi omitido).

Listagem 3. Programando para a interface

```
// Form Visualizador.
public class Visualizador : System.Windows.Forms.Form
{
  // Mantém instância genérica de objeto que implementa .
  private  forma = null;


  ///
  /// Construtor usado para instanciar o Form,
  /// já preparando-o para visualizar a Forma correta
  ///
  /// Objeto que será desenhado
  ///             e visualizado no Form.
  ///
  public Visualizador( objeto)
  {
              InitializeComponent();
              // Persiste o objeto passado durante a existência do Form
```

```
        this.forma = objeto;
  }
}


// Form ExemploPolimorfismo.
public class ExemploPolimorfismo : System.Windows.Forms.Form
{
  private void btnExecutar_Click(object sender, System.EventArgs e)
  {
    Circulo circ = new Circulo();
        this.Mostrar(circ);
        Quadrado quad = new Quadrado();
        this.Mostrar(quad);
        Pessoa pes = new Pessoa();
        this.Mostrar(pes);
        Funcionário fun = new Funcionário();
        this.Mostrar(fun);
  }


  private void Mostrar( objeto)
  {
        Visualizador v = new Visualizador(objeto);
        v.Show();
```

```
  }

}
```

Além das alterações onde trocou-se o tipo Forma para o tipo , foi adicionado código para instanciar a classe Funcionário e passar o objeto para o método Mostrar. A Figura 5 mostra o resultado de todas estas alterações.

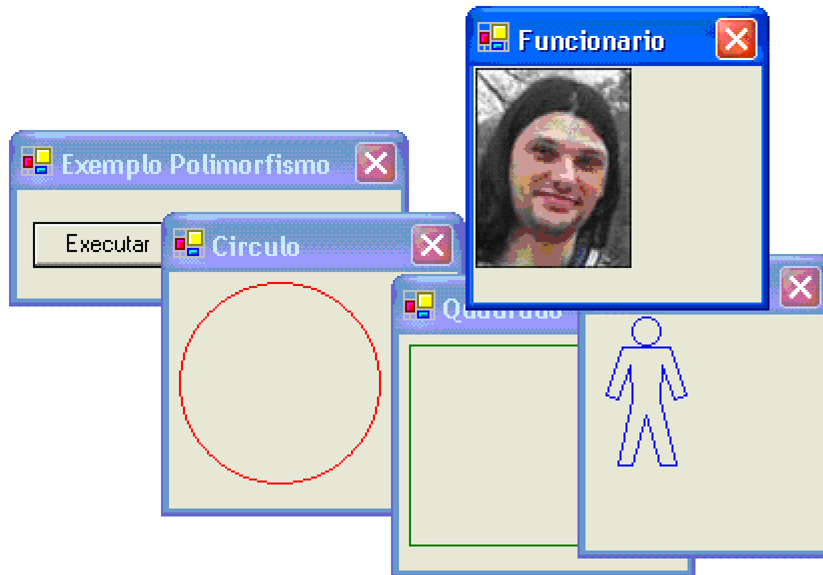

Figura 5. Polimorfismo em ação

Refrescando a mente: usando o conceito de polimorfismo podemos ter objetos diferentes que possuam algum método de mesmo nome e assinatura, e chamar estes métodos de modo genérico: Objeto.Método(parâmetros).

E o resultado poderá ser bastante diferente, dependendo da forma como cada objeto implementa a operação. Isso é possível para objetos que tenham um relacionamento estabelecido pelo fato de herdarem de uma mesma subclasse, ou implementarem uma interface em comum.

Conclusão

Nesta parte deste artigo, vimos que objetos são importantes em .NET pelo simples fato de que tudo em .NET é objeto. Por isto, uma clara compreensão da Programação

Orientada a Objetos (POO) é necessária para que o desenvolvedor escreva e leia código fluentemente. Vimos que os quatro principais alicerces da POO são: Abstração, Encapsulamento, Herança e Polimorfismo, sobre os quais vimos os conceitos e simples exemplos para ilustrar aos conceitos.
Na próxima parte deste artigo continuaremos investigando mais sobre POO. Até o próximo mês!

Claudio Lassala (classala@eps-software.com) é Desenvolvedor Senior da EPS Software Corp., em Houston, TX. É palestrante freqüente em grupos de usuários, seminários e conferências na América do Norte e Brasil. É MVP C#.NET, MCSD for .NET, e colunista na MSDN Brasil. Criou e apresentou vários treinamentos em vídeo para o Universal Thread, e tem artigos publicados em diversas revistas internacionais. Bio completa: www.lassala.net/bio