VB. NET - Programação Orientada a Objetos (POO)

SISTEMA DE AGENDA ELETRÔNICA PESSOAL

Pato Branco, 30 de maio de 2014.

Para exemplificar os conceitos da POO será criado um sistema de Agenda Pessoal, onde será possível cadastrar Contatos, e também Estabelecimentos, com informações como Nome, Telefone, E-mail, Site Pessoal, Endereço, etc. O sistema possibilitará visualizar a lista cadastrada, assim como imprimir a mesma.

Esta definição nos indica o objetivo do sistema: Cadastrar contatos e estabelecimentos na Agenda, mas podemos também tirar outras informações. Uma técnica muito conhecida para diagramação de classes é analisar a frase de definição de um sistema, identificar os substantivos como classes ou atributos e os verbos como métodos.

Neste exercício serão apresentados e usados alguns conceitos e diagramas da UML (*Unified Modeling Language*) para esclarecer a arquitetura e o escopo do sistema.

A *Unified Modeling Language* é linguagem de modelagem não proprietária de terceira geração. É um método aberto usado parta especificar, visualizar, construir e documentar os artefatos de um sistema de software orientado a objetos.

A UML tem origem na compilação das “melhores práticas de engenharia” que provaram ter sucesso na modelação de sistemas grandes e complexos. Sucedeu os conceitos de Booch, OMT (Rumbaugh) e OOSE (Jacobson), fundindo-se em uma única linguagem de modelagem comum e largamente utilizada. A UML pretende ser a linguagem de modelagem padrão para modelar sistemas concorrentes e distribuídos. Para mais informações, visite o site http://www.uml.org.

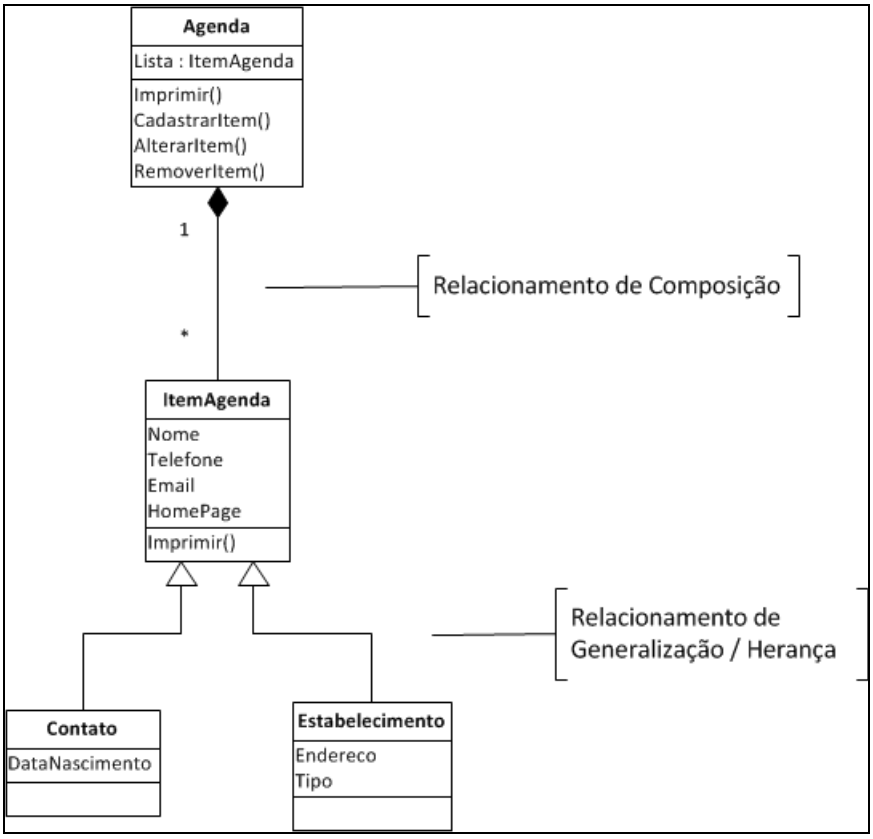
Após a análise do texto do primeiro parágrafo, obtemos a seguinte lista:

Classes/Atributos (substantivos)	Métodos (verbos)
Agenda	Criaremos
Contato	Cadastrar
Estabelecimento	Possibilitará
Nome	Visualizar
E-mail	Imprimir
Telefone	
Site Pessoal	
Endereço	
Lista	

Com a lista, podemos então continuar com uma análise mais detalhada, identificando se há necessidade de criarmos todas as classes encontradas, quais substantivos são classes, quais são atributos, quais verbos são os métodos, etc. Vale a

pena lembrar que esta técnica de análise não é “à prova de falhas”. Quanto maior e mais complexo o sistema, maior a prioridade do levantamento dos requisitos, entrevistas com usuários e assim por diante.

No final, a estrutura do sistema fica como a seguinte:



Esse é um diagrama de classes UML e nas próximas linhas será explicado de forma sucinta como é a construção deste diagrama que é um dos principais elementos da programação orientada a objetos.

As classes em um diagrama UML são representadas pelo retângulo dividido em três partes:

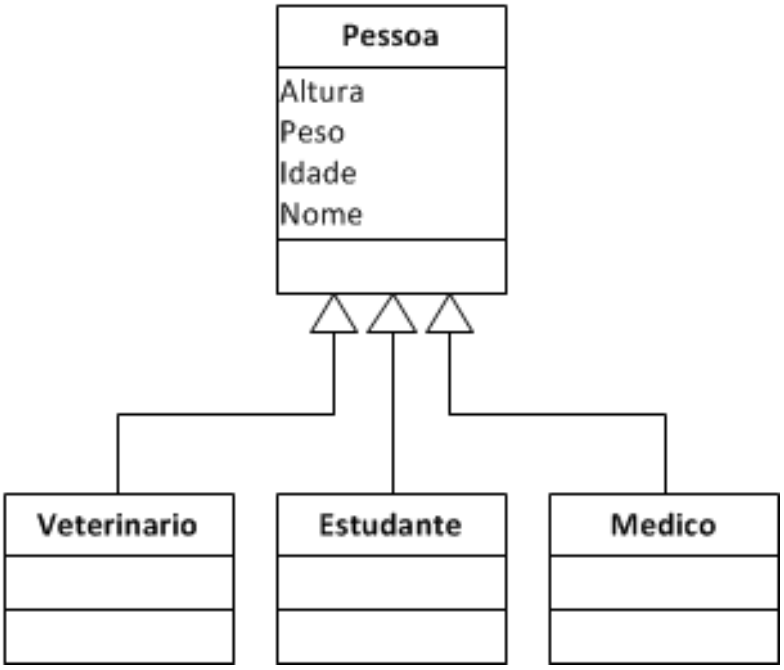
- **Topo:** Nome da Classe.
- **Centro:** Lista de Atributos da Classe.
- **Rodapé:** Lista de Métodos da Classe.

Note também os relacionamentos a seguir.

Generalização

Representado por uma linha sólida conectando as classes, onde um triângulo indica a **Super Classe** ou **Classe Base** e a outra indica a **Classe Derivada** ou **SubClasse**. A **Herança** é o conceito chave da OO, permitindo a generalização de características comuns a outras classes.

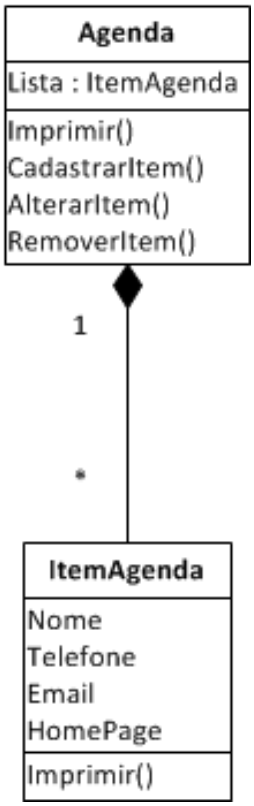
Exemplo:



Composição

Representado por uma linha sólida conectando as classes, onde um losango preenchido indica a classe composta por uma ou várias instâncias da casse da outra ponta. É o relacionamento de “Todo-parte Total”.

Exemplo:



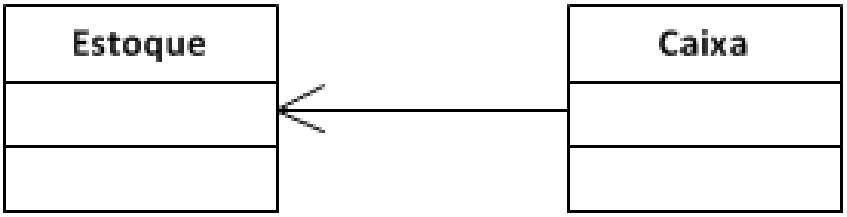
A classe **Agenda** é composta por várias instâncias da classe **ItemAgenda**.

Associação

Relacionamento entre classes. Indica uma dependência estrutural, sendo representado por uma linha sólida entre elas. O fluxo de dados pode ser unidirecional ou bidirecional (indicando uma comunicação em um sentido, onde somente uma das classes tem “conhecimento” da existência da outra ou se as duas possuem esse “conhecimento”, respectivamente). Entre duas classes pode existir mais de uma associação.

- *Associação Unidirecional*

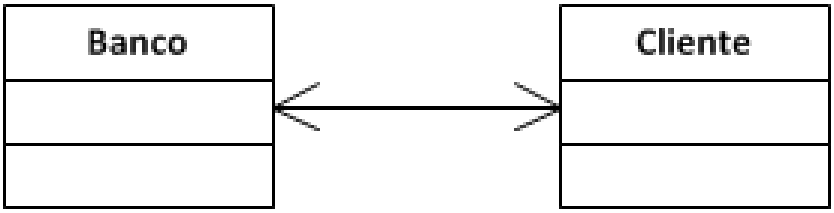
Exemplo:



Neste caso, somente a classe **Caixa** sabe da existência da classe **Estoque**.

- *Associação Bidirecional*

Exemplo:



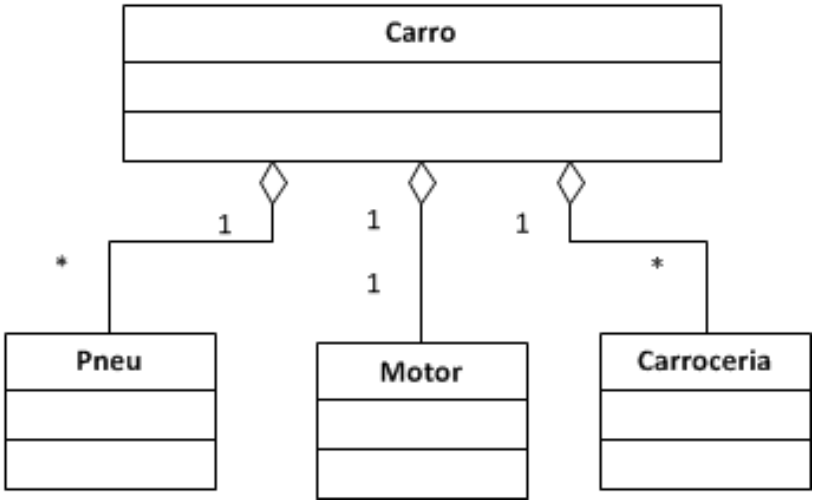
Em contrapartida, nesse caso as duas classes possuem “conhecimento” da existência da outra.

Agregação

Muito similar do relacionamento de composição visto anteriormente, a agregação simboliza o relacionamento “Todo-Parte Parcial”, mas não totalmente, ou seja, as duas classes envolvidas existem independente da outra. É representado por uma linha sólida ligando as classes, com um losango indicando a classe composta.

- *Agregação da classe Carro*

Exemplo:



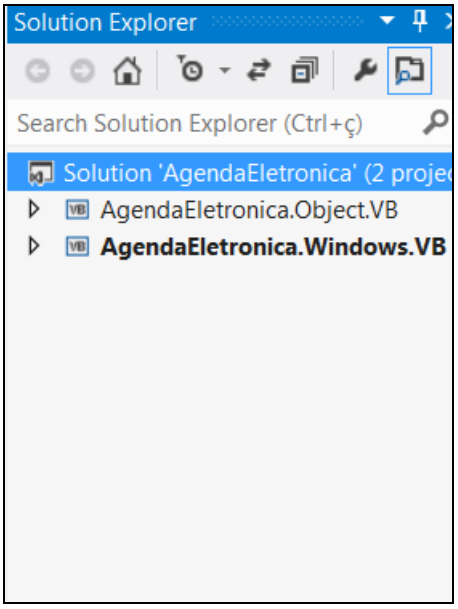
Sistema Agenda - O Início

Para criarmos nossa Agenda Eletrônica usaremos a ferramenta Visual Studio .NET com um projeto do tipo **Visual Studio Solutions** :: **Blank Solution**. Coloque o nome da solução como **AgendaEletronica** e salve a mesma em um diretório do seu computador. Uma solução é um conjunto de projetos e um projeto pode constar em diferentes soluções. No Visual Studio é permitido visualizar apenas uma solução por vez. Em nosso projeto iremos criar dois projetos: um do tipo *Class Library* e outro do tipo *Windows Forms Application*. A seguir são esclarecidos os conceitos referentes aos dois tipos de projetos.

- **Windows Forms Application:** É utilizado para se criar a IU (Interface do Usuário) em um ambiente Windows, utilizando o conceito de Forms, muito usado em outras linguagens como o Delphi. Ele conterá a "cara" de nosso sistema e irá acessar as classes do projeto *Class Library*.
- **Class Library:** É utilizado para a criação de bibliotecas de classes, componentes, etc. Ele pode ser usado para encapsular a lógica de controles, mapeamento de entidades e assim por diante. Uma *Class Library* pode ser referenciada por outros projetos e sua compilação pode gerar tanto uma DLL (*Dynamic Link Library*) ou um arquivo EXE (arquivos executáveis).

Realize as seguintes etapas:

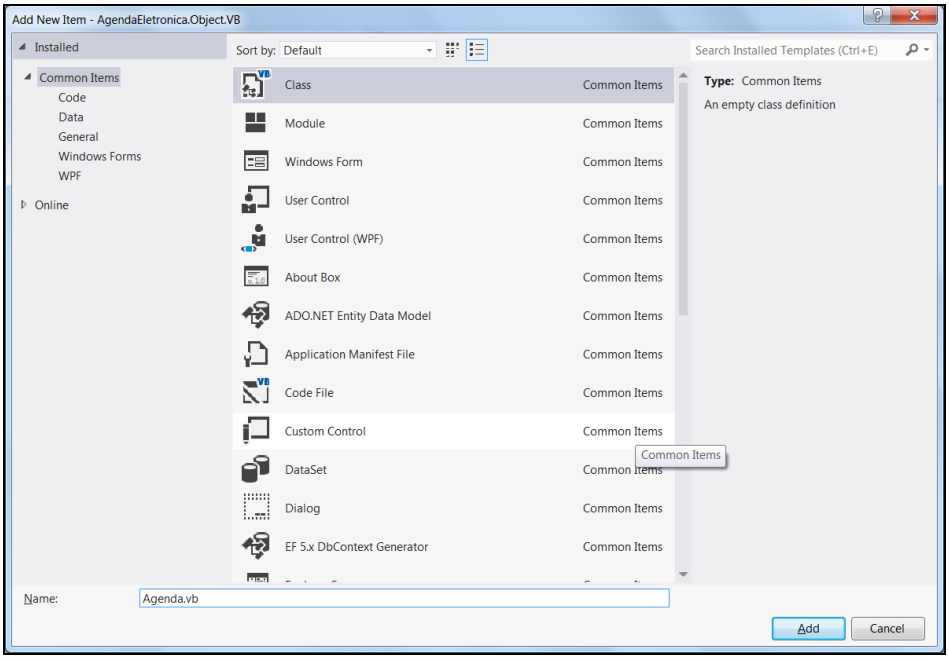
- ✓ Adicione um novo projeto do tipo **Windows Forms Application** na solução criada anteriormente com o nome **AgendaEletronica.Windows.VB**.
- ✓ Adicione um novo projeto do tipo **Class Library** na solução criada anteriormente com o nome **AgendaEletronica.Object.VB**.
- ✓ Exclua os arquivos que a ferramenta criou porque não iremos utilizá-los. Eles se chamam **Class1.vb** e **Form1.vb**.



Criando as classes do sistema

A partir do diagrama de classes do nosso sistema iremos criar as classes no projeto que foi adicionado na solução. Para isto basta clicar com o botão direito do mouse no projeto **AgendaEletronica.Object.VB** e adicionar as seguintes classes:

- ItemAgenda
- Agenda
- Contato
- Estabelecimento



Depois de todas as classes criadas, iremos adicionar suas funcionalidades. Começaremos pela **ItemAgenda**, pois é a principal classe do nosso sistema.

Clique duas vezes sobre a mesma e adicione o seguinte código:

```
1  Imports System
2  Imports System.Text
3
4  Public MustInherit Class ItemAgenda
5
6  End Class
```

A declaração **Imports** é utilizada para importar um **namespace**. Vamos ver na seção a seguir os conceitos relacionados a um *namespace*.

Namespaces

É um meio de organizar logicamente as classes. Todas as classes do .Net framework estão organizadas em *namespaces*, sendo separadas por categoria, tecnologia a que se referem, etc. Os *namespaces* não estão ligados à estrutura física das pastas e, ao importa-los, você estará criando um "atalho". Isso quer dizer que não é preciso especificar o nome completo de uma classe (Ex: *System.Text.StringBuilder*) quando quiser declará-la ou acessá-la. Poderá simplesmente usar *StringBuilder* (desde que o namespace *System.Text* esteja declarado).

Os elementos que compõem a classe **ItemAgenda** são:

- **Public:** Modificador de acesso da classe (ver lista a seguir).
- **MustInherit:** Modificador de herança da classe (ver lista a seguir).
- **Class:** Define que o elemento declarado é uma classe.
- **ItemAgenda:** Identificador / Nome da Classe.

Modificadores de Acesso

Os modificadores de acesso são utilizados para especificar qual o nível de acesso do elemento declarado, ou seja, no exemplo anterior, colocando a palavra-chave **Public** antes da declaração da classe, ela torna-se pública, em outros termos, ela pode ser acessada por qualquer classe, componente, módulo, etc.

Existem outros modificadores de acesso que o .NET dispõe. Segue agora uma lista deles:

Modificador de acesso	Descrição
Public	Como dito anteriormente, com esse modificador o elemento declarado será acessado de qualquer nível.
Protected	O elemento declarado somente é acessado pela classe que o contém, ou por classes que derivam da classe que o contém.
Friend	O elemento declarado somente é acessado por outros elementos que estão no mesmo *Assembly*.
Private	O elemento declarado somente é acessado pelo elemento que o contém.

Outros modificadores:

Modificador de acesso	Descrição
MustInherit	Especifica que o elemento declarado não possui implementação e que as classes derivadas devem fornecer a implementação daquele elemento. Se for utilizado na declaração de uma classe, aquela classe não pode ser instanciada diretamente, somente pode ser utilizada como Classe Base.
NotInheritable	Indica que a classe não pode ser usada como Classe Base.
Shared	Todas as instâncias que possuam o elemento usarão a mesma cópia da memória.

Construtores e Destrutores

Construtores e destrutores também são métodos, só que com uma particularidade e são executados no momento da inicialização e destruição da classe.

- **Construtores:** São os encarregados de instanciar uma classe. Ao fazer isso, é criado um objeto e alocado na memória. Também podem garantir que quando um cliente for instanciar sua classe, ela será inicializada corretamente. Eles aceitam parâmetros e retornam um ponteiro para a instância que criaram.

  Objeto é a instância de uma classe.

  - Inclua o seguinte código entre o bloco da classe que criamos anteriormente (**ItemAgenda**):

```
1   Imports System
2   Imports System.Text
3
4   Public MustInherit Class ItemAgenda
5       Public Sub New()
6
7       End Sub
8   End Class
```

- **Destrutores:** São encarregados da destruição da instância de uma classe. Eles podem ser utilizados para fechar recursos externos ou *unmanaged* (não gerenciáveis).

  Pontos importantes sobre Destrutores:

  - ✓ Não podem ser definidos em *structs*. Somente são usados em classes.
  - ✓ Uma classe pode ter somente um Destrutor.
  - ✓ Não podem ser herdados ou sobrecarregados.
  - ✓ Não podem ser chamados. São executados automaticamente.
  - ✓ Não possuem modificadores e não recebem parâmetros.

- ✓ Não use Destrutores vazios. Ao criar um destrutor, sua instância é movida para uma fila de finalização. Ao mover para essa fila sem executar nenhum código, é gerado um processamento desnecessário, debilitando o desempenho.

Exemplo:

```
1   Imports System
2   Imports System.Text
3
4   Public MustInherit Class ItemAgenda
5       Public Sub New()
6
7       End Sub
8
9       Protected Overrides Sub Finalize()
10          MyBase.Finalize()
11      End Sub
12  End Class
```

Propriedades

Propriedades são estruturas que dão acesso aos dados da classe. Elas são muito parecidas com os campos, mas com várias vantagens. Propriedades **devem** ser utilizadas, pois é uma boa prática de Encapsulamento. Propriedades são, na verdade, métodos de acesso a variáveis.

Vamos declarar as propriedades da classe **ItemAgenda** pata depois entrarmos em detalhes:

```
4   Public MustInherit Class ItemAgenda
5       Private _email As String
6       Private _homePage As String
7       Private _nome As String
8       Private _telefone As String
9
10      Public Property Email() As String
11          Get
12              Return Me._email
13          End Get
14          Set(ByVal value As String)
15              Me._email = value
16          End Set
17      End Property
18
19      Public Property HomePage() As String
20          Get
21              Return Me._homePage
22          End Get
23          Set(ByVal value As String)
24              Me._homePage = value
25          End Set
26      End Property
```

Analisando a declaração, vemos que foram mostradas quatro variáveis do tipo **String**, que serão utilizadas para armazenarem as informações do objeto **ItemAgenda**. Mas se repararmos bem, todos os campos estão com o Modificador de Acesso privado, então como eles serão acessados de fora da classe? E aí que

entram as **propriedades**.

As propriedades podem ser de leitura e escrita, somente leitura ou somente escrita. Elas usam os *getters* (leitura) e os *setters* (escrita) para disponibilizarem o acesso ao campo.

Com o encapsulamento dos campos em propriedades, você aumenta ainda mais a probabilidade de reutilização de sua classe e também diminui ou até previne maiores impactos em clientes que a estão consumindo, caso alguma alteração precise ser feita. Imagine que agora precisaremos adicionar validação ao campo de e-mail, prevendo que somente endereços válidos possam ser preenchidos. Lembre-se que as informações das classes cabem a elas, portanto nada mais justo que essa validação fique na própria classe. Mas, se meus clientes estivessem acessando diretamente o campo de e-mail, provavelmente eles teriam que alterar seu código para se adequarem à validação, o que com a utilização de uma propriedade isso não acontece. Eu poderia facilmente adicioná-la no método *Setter* da minha propriedade e qualquer erro poderia disparar uma exceção. Simples, eficiente e sem alteração no código dos clientes.

Com a criação das propriedades, vamos agora criar um novo construtor, prevendo a instanciação da classe, já passando o valor dos campos. Iremos utilizar a sobrecarga de métodos.

Sobrecarga de Métodos (Overload)

A sobrecarga permite que você crie métodos com o mesmo nome, mas com assinaturas diferentes. Assinatura é a lista de parâmetros que o método recebe, que é um recurso muito útil, pois proporciona a possibilidade de conservar a semântica dos nomes dos membros públicos em casos que se necessita de um suporte diferenciado para diversas origens de dados. A palavra “sobrecarga” pode trazer uma ideia de estresse ou excesso de trabalho, mas neste caso (OO) é um elemento que nos ajuda (e muito!). Um bom exemplo disso que é o método **Show** da classe **MessageBox**. Vemos que o editor mostra que existem 21 alternativas diferentes para o método. O que diferencia um do outro é a combinação do número de parâmetros, seus nomes e tipos.

Vamos então alterar o **construtor** da classe **ItemAgenda**, para aplicar a sobrecarga de métodos:

```
Public Sub New()
    Me.New(String.Empty, String.Empty, String.Empty, String.Empty)
End Sub

Public Sub New(ByVal strNome As String, ByVal strEmail As String, ByVal strTelefone As String, ByVal strHomePage As String)
    Me.Nome = strNome
    Me.Email = strEmail
    Me.Telefone = strTelefone
    Me.HomePage = strHomePage
End Sub
```

Agora temos dois construtores, um que recebe parâmetros para todos os campos da classe, inicializando, assim, todas as variáveis corretamente e um, que não recebe nenhum, mas utilizada do outro para inicializar as variáveis como *String.Empty* (representa uma strings vazia). Essa é uma técnica comum que visa à correta inicialização da classe, reutilizando o código. Podemos reparar também a *keyword* **Me**. Ela indica que o campo a ser utilizado é o da própria instância.

Veremos sobre isso mais tarde.

A classe **ItemAgenda** somente possui um método chamado **Imprimir**, que retorna uma variável strings com as informações da instância.

```
Public Overridable Function Imprimir() As String
    Dim impressao As StringBuilder = Nothing
    impressao = New StringBuilder
    impressao.AppendFormat("Nome: {0}", Me.Nome)
    impressao.AppendLine()
    impressao.AppendFormat("E-mail: {0}", Me.Email)
    impressao.AppendLine()
    impressao.AppendFormat("Telefone: {0}", Me.Telefone)
    impressao.AppendLine()
    impressao.AppendFormat("Home Page: {0}", Me.HomePage)
    impressao.AppendLine()
    Return impressao.ToString
End Function
```

É um método simples que instancia uma classe *StringBuilder* (utilizada para melhor performance em concatenações de strings) e recupera as informações dos campos da instância, retornando a *string* completa.

Vamos agora analisar o código da classe **Contato** (ver o código na página seguinte).

A declaração **#Region** serve estritamente para organização do código e com ela podemos quebrar as partes que compõem a classe. Estão de exemplo as **Regions** *Campos Privados*, *Propriedades*, *Construtores* e *Métodos*, mas você pode criar qualquer uma e até utilizar uma estrutura hierárquica.

Exemplo:

```
#Region “Pai”
    #Region “Filho”
        #Region “Neto”
        #End Region
    #End Region
    #Region “Filho 2”
    #End Region
#End Region
```

Observação: As declarações #Region só tem efeito de organização do código, não irão alterar ou afetar de modo algum o resultado da compilação.

```vb
Imports System
Imports System.Text

Public Class Contato
    Inherits ItemAgenda
#Region "Campos Privados"
    Private _dataNascimento As DateTime
#End Region

#Region "Propriedades"
    Public Property DataNascimento() As DateTime
        Get
            Return Me._dataNascimento
        End Get
        Set(ByVal value As DateTime)
            Me._dataNascimento = value
        End Set
    End Property
#End Region

#Region "Construtores"
    Public Sub New()
        Me.New(String.Empty, String.Empty, String.Empty, String.Empty, New DateTime)
    End Sub

    Public Sub New(ByVal strNome As String, ByVal strEmail As String, ByVal strTelefone As String, _
                   ByVal strHomePage As String, ByVal dtDataNascimento As DateTime)
        MyBase.New(strNome, strEmail, strTelefone, strHomePage)
        Me.DataNascimento = dtDataNascimento
    End Sub
#End Region

#Region "Métodos"
    Public Overrides Function Imprimir() As String
        Dim impressao As StringBuilder = Nothing
        impressao = New StringBuilder
        impressao.Append(MyBase.Imprimir)
        impressao.AppendFormat("Data de Nascimento: {0}", Me.DataNascimento)
        impressao.AppendLine()
        impressao.AppendLine()
        Return impressao.ToString
    End Function
#End Region
End Class
```

Outra *keyword* nova é **MyBase**. Ao contrário do **Me**, ela indica que em tempo de execução deve ser acessado o elemento da **Classe Base** e não o da instância. Em nosso caso, especificamos que a **SubClasse** utilize os construtores da **SuperClasse**, para depois então adicionar a sua lógica.

Também não podemos esquecer que essa é a nossa primeira **Classe Derivada**. Notaram o trecho de código **Inherits ItemAgenda** após a declaração da classe **Contato**? É a magia da OO entrando em ação.

Agora todas as propriedades e métodos da **Classe Base** estarão disponíveis para as **SubClasses**, além das particularidades adicionais (data de nascimento, por exemplo).

Mas esperem aí, como existe um método **Imprimir()** na classe **ItemAgenda** e outro agora na Contato? Não é uma sobrecarga de métodos, pois os dois possuem a mesma assinatura (nenhum deles recebe parâmetro). Então o que aconteceu? Nós acabamos de criar uma **Sobreposição**.

Sobreposição (Override)

Sobreposição é o mecanismo que uma **SubClasse** possui de literalmente “passar por cima” de métodos ou propriedades definidos em sua **SuperClasse**. Para que a **SubClasse** consiga sobrepor o elemento, a **SuperClasse** precisa declará-lo com a *keyword* **Overridable**, para possibilitar a sobreposição.

Mantendo a mesma identificação e assinatura do método **Imprimir**, por exemplo, a classe **Contato** adicionando a *keyword* **Overrides**, descarta a definição do método **Imprimir** da classe **ItemAgenda** e cria a sua própria.

Vamos criar agora a classe **Estabelecimento**:

```
Imports System
Imports System.Text

Public Class Estabelecimento
    Inherits ItemAgenda
#Region "Campos Privados"
    Private _endereco As String
#End Region

#Region "Propriedades"
    Public Property Endereco() As String
        Get
            Return Me._endereco
        End Get
        Set(ByVal value As String)
            Me._endereco = value
        End Set
    End Property
#End Region

#Region "Construtores"
    Public Sub New()
        Me.New(String.Empty, String.Empty, String.Empty, String.Empty, String.Empty)
    End Sub

    Public Sub New(ByVal strNome As String, ByVal strEmail As String, ByVal strTelefone As String, _
                   ByVal strHomePage As String, ByVal strEndereco As String)
        MyBase.New(strNome, strEmail, strTelefone, strHomePage)
        Me.Endereco = strEndereco
    End Sub
#End Region

#Region "Métodos"
    Public Overrides Function Imprimir() As String
        Dim impressao As StringBuilder = Nothing
        impressao = New StringBuilder
        impressao.Append(MyBase.Imprimir)
        impressao.AppendFormat("Endereço: {0}", Me.Endereco)
        impressao.AppendLine()
        impressao.AppendLine()
        Return impressao.ToString
    End Function
#End Region
End Class
```

Assim como a classe **Contato**, a classe **Estabelecimento** também é derivada da classe **ItemAgenda**, adicionando uma propriedade e uma variável para armazenar o Endereço, utilizando os construtores da **Classe Base** e também fazendo a sobreposição do método **Imprimir**. Vejamos agora a classe **Agenda**:

```vb
Imports System
Imports System.Text
Imports System.Collections

Public Class Agenda
#Region "Campos Privados"
    Private _itens As ArrayList
#End Region

#Region "Propriedades"
    Public ReadOnly Property Itens() As ArrayList
        Get
            Return Me._itens
        End Get
    End Property
#End Region

#Region "Construtores"
    Public Sub New()
        Me.SetLista(New ArrayList)
    End Sub
#End Region

#Region "Métodos"
    Public Sub CadastrarItem(ByVal objItemAgenda As ItemAgenda)
        Me.Itens.Add(objItemAgenda)
    End Sub

    Public Function Imprimir() As String
        Dim impressao As StringBuilder = Nothing
        impressao = New StringBuilder

        For Each item As ItemAgenda In Me.Itens
            impressao.AppendLine(item.Imprimir)
        Next
        Return impressao.ToString
    End Function

    Public Sub RemoverItem(ByVal objItemAgenda As ItemAgenda)
        Me.Itens.Remove(objItemAgenda)
    End Sub

    Public Sub RemoverItem(ByVal intIndice As Integer)
        Me.Itens.RemoveAt(intIndice)
    End Sub

    Private Sub SetLista(ByVal itens As ArrayList)
        Me._itens = itens
    End Sub
#End Region
End Class
```

A classe **Agenda** possui uma propriedade Itens, que nada mais é que uma lista de objetos da classe **ItemAgenda**. Ela também dispõe de métodos que manipulam a lista, como **CadastrarItem** e **RemoverItem**.

Podemos notar outro caso de sobrecarga de métodos, onde o método **RemoverItem** possui duas assinaturas diferentes. Uma em que ele recebe como parâmetro o próprio objeto a ser removido e no outro ele recebe o índice no array do objeto a ser removido.

Nessa classe, o método que mais nos interessa é o **Imprimir**, pois é um exemplo da utilização de **Polimorfismo**. Notem que o método Imprimir apenas sabe que a lista contém instâncias da classe **ItemAgenda**, mas não é definido em lugar algum que ela é um **Contato** ou um **Estabelecimento**. Em tempo de execução, O CLR verifica o tipo do objeto, para, então, chamar o método apropriado.

Sistema Agenda - Criando a Interface do Usuário

Com as classes criadas, precisamos agora criar a nossa Interface do Usuário. Para isso, caso não tenha feito, apague o arquivo **Form1.vb**, pois não iremos precisar do mesmo.

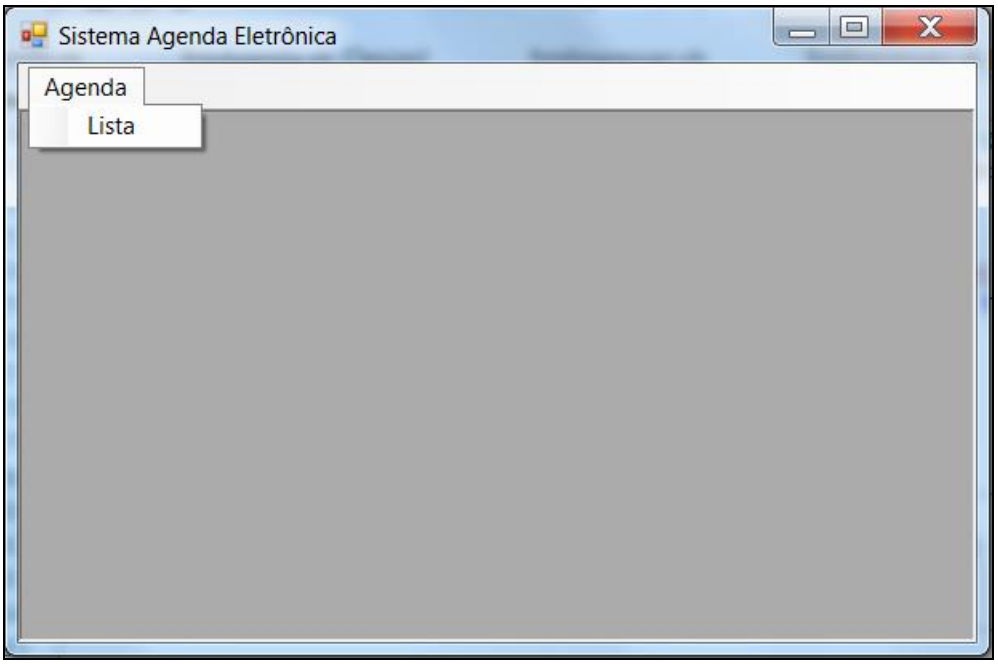
Adicione os seguintes forms ao projeto **AgendaEletronica.Windows.VB**:

- ✓ **frmAgenda**
- ✓ **frmLista**
- ✓ **frmDetalheContato**
- ✓ **frmDetalheEstabelecimento**
- ✓ **frmImpressao**

Descrição dos Forms:

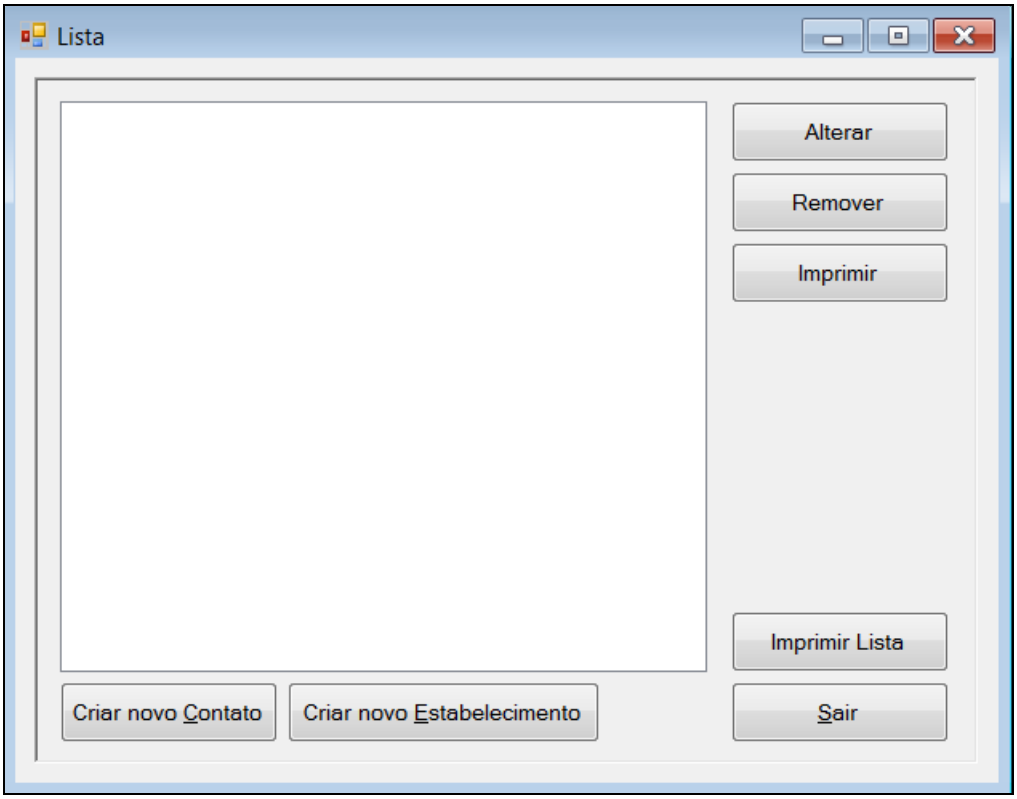
- **frmAgenda**

  É um simples form com um Menu, usado como form pai de todos os outros forms.



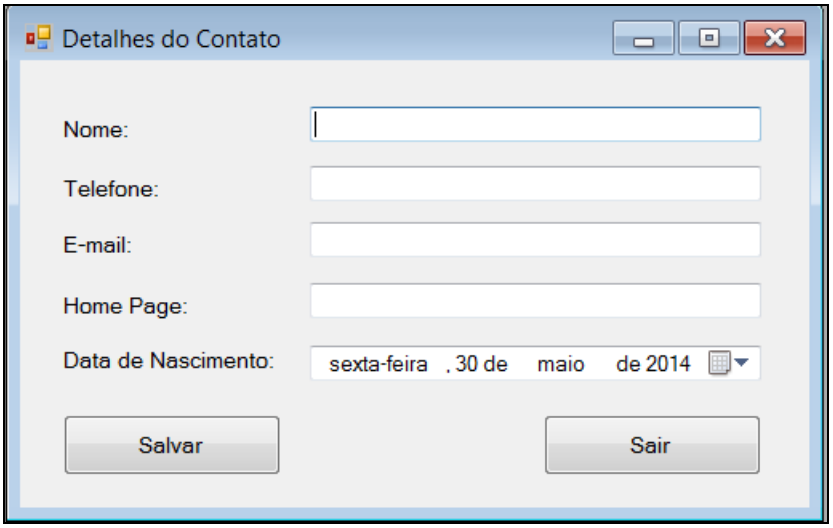
- **frmLista**

  Contém a lista da Agenda, assim como botões para as operações relacionadas.



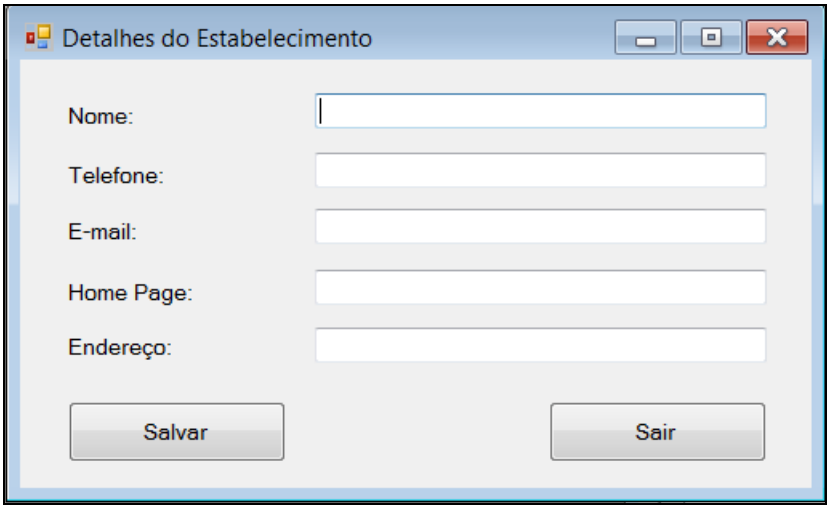
- **frmDetalheContato**

  Insere/Atualiza/Visualiza os detalhes de um Contato.



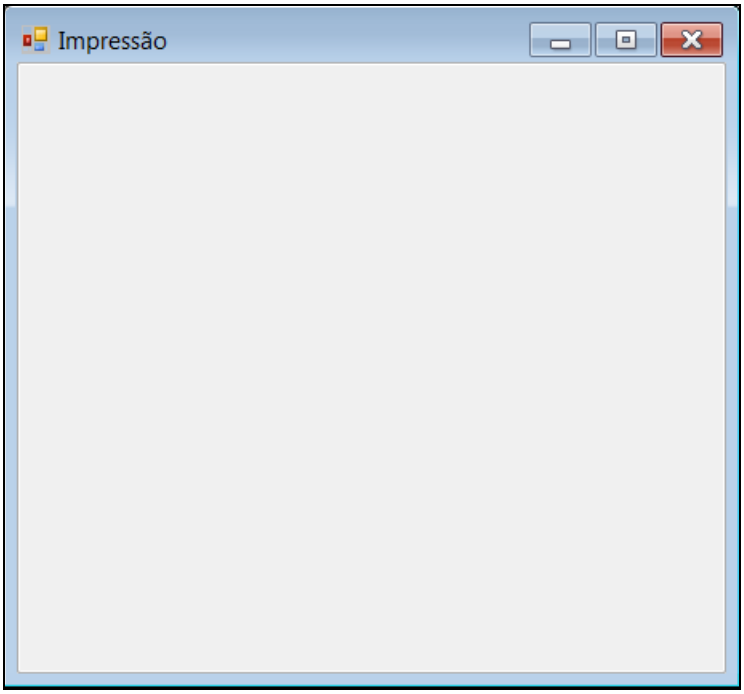
- **frmDetalheEstabelecimento**

  Insere/Atualiza/Visualiza os detalhes de um Estabelecimento.



- **frmImpressao**

  Form usado para impressão.



Importante!

Como o código dos forms contém várias linhas, os mesmos serão disponibilizados na ferramenta Moodle para download.

Dicas:

Você pode melhorar o Sistema da Agenda incluindo novas funcionalidades, salvando os dados em um banco de dados e modificando a aparência da mesma ☺.