

# POO no Delphi

## Conceitos e Implementação - Parte 2



**ALESSANDREIA DE OLIVEIRA**  
(amojulio@granbery.edu.br)  
é Professora do curso de Bacharelado em Sistemas de Informação da Faculdade Metodista Granbery, Mestre em Engenharia de Sistemas e Computação pela COPPE/UFRJ.

**MARCO ANTÔNIO P. ARAÚJO**  
(maraujo@granbery.edu.br)  
é Professor do Curso de Bacharelado em Sistemas de Informação da Faculdade Metodista Granbery, Doutorando e Mestre em Engenharia de Sistemas e Computação pela COPPE/UFRJ e Analista de Sistemas da Prefeitura de Juiz de Fora.

**JOÃO CARLOS DA SILVA**  
(jcsilva@pos.granbery.edu.br)  
é graduado em Sistemas de Informação pela Faculdade Metodista Granbery, desenvolvedor de sistemas há 8 anos.

**N**o artigo da edição anterior iniciamos a apresentação de alguns dos principais conceitos da programação orientada a objetos (POO) no Delphi, abordando classes, atributos, métodos, encapsulamento e herança. Estes conceitos foram apresentados através de exemplos a partir de um modelo de classes representando departamentos e seus funcionários, de diferentes tipos. Este artigo complementa o anterior abordando outros importantes conceitos de POO como polimorfismo, associação e interfaces, além de apresentar um estudo de caso que se utiliza dos conceitos apresentados em ambos os artigos, mostrando como podem ser utilizados na prática.

### Polimorfismo

O polimorfismo é uma característica da Orientação a Objetos que permite aos objetos das classes terem comportamentos diferentes, sejam de seus ancestrais, da própria classe ou mesmo de outras classes. Assim, os métodos definidos e implementados em classes ancestrais, podem ser redefinidos e reimplementados em classes descendentes.

Para que isso funcione adequadamente, é necessário garantir qual implementação de um método será executada, visto que pode existir mais de uma com o mesmo nome, seja na hierarquia ou dentro da própria classe. Dessa forma o Delphi oferece um conjunto de palavras reservadas que permitem definir o comportamento das chamadas de métodos, ou seja, a ligação da chamada com a implementação desejada.

De acordo com sua forma de ligação, os métodos podem ser classificados em três tipos: *static*, *virtual* e *dynamic*. Os métodos estáticos (*static*) são o padrão do Delphi, ou seja, caso não sejam utilizadas explicitamente as diretivas *virtual* ou *dynamic*, os mesmos são considerados estáticos. A chamada desses tipos de métodos ativa a implementação

feita na classe que instanciou o objeto.

Na hierarquia de funcionários, onde estão definidos os métodos polimórficos *CalcularSalario* e *CalcularPremio*, ainda não foram aplicadas diretivas nos métodos modificando o seu tipo e, por isso, os mesmos são estáticos.

Durante a execução do sistema, variáveis podem receber objetos criados a partir de classes diferentes do tipo declarado, uma vez que estejam numa mesma hierarquia. Dessa forma a ativação dos métodos ocorre de maneira diferente entre os objetos. Isso ocorre porque para métodos estáticos, a implementação ativada é a correspondente da classe que define a variável e não da classe que define sua instância.

Tal comportamento pode ser modificado para métodos virtuais e dinâmicos, através do uso das diretivas *virtual* e *dynamic*, em conjunto com a diretiva *override*. Assim, a chamada dos métodos passa a ativar a implementação do tipo em tempo de execução, ou seja, de acordo com o tipo da instância e não o tipo definido para a variável.

O uso de *override*, garante que apenas uma implementação do método exista em tempo de execução nas classes descendentes, ocultando as implementações dos ancestrais. Porém, esses métodos devem ter exatamente a mesma assinatura, ou seja, mesmo nome do método e mesmo número, ordem e tipos dos parâmetros.

As diretivas *virtual* e *dynamic* são equivalentes, sendo diferentes no sentido de que a *virtual* otimiza a velocidade de acesso e *dynamic* otimiza o tamanho do código, sendo a *virtual* a forma mais comum e eficiente de implementar o polimorfismo. Na **Listagem 1** demonstra-se o efeito do uso dessas diretivas, através de algumas modificações na hierarquia de *Funcionario*.

#### Listagem 1. Definição da hierarquia de Funcionario com método virtual

```
type
  Funcionario = class
    function CalcularSalario: real; virtual;
  end;

  FuncionarioMensalista = class (Funcionario)
    function CalcularSalario: real; override;
  end;

  FuncionarioDiarista = class (Funcionario)
    function CalcularSalario: real; override;
  end;
```

Na **Listagem 1**, utiliza-se na assinatura do método *CalcularSalario* na classe *Funcionario*, a diretiva *virtual*, enquanto que nas classes descendentes utiliza-se a diretiva *override*. As implementações dos métodos continuam as mesmas.

A **Listagem 2** apresenta a utilização dos métodos com a diretiva *override*. Pressupõe-se que os valores dos campos dos objetos tenham sido modificados entre a criação dos objetos e a execução dos serviços *CalcularSalario*. A **Listagem 2** deve ser implementada em outra unit que não a *untClasses*, devendo referenciá-la.

#### Listagem 2. Chamada de métodos virtuais com override

```
01: var
02:   Func1: Funcionario;
03:   Func2: FuncionarioDiarista;
04: begin
05:   {Funcionario 1}
06:   Func1 := FuncionarioMensalista.Create;
07:   Func1.CalcularSalario;
08:   {Funcionario 2}
09:   Func2 := FuncionarioDiarista.Create;
10:   Func2.CalcularSalario;
11: end;
```

Durante a execução do trecho de código apresentado na **Listagem 2**, as chamadas de métodos nas linhas 07 e 10 ativam a implementação correspondente das classes *FuncionarioMensalista* e *FuncionarioDiarista* respectivamente. Embora o tipo de dados da variável *Func1* seja *Funcionario* (linha 02), o objeto armazenado nela é do tipo *FuncionarioMensalista* (linha 06). Sendo assim, em função da diretiva *override*, são executadas as implementações relativas às instâncias, e não relativas ao tipo da variável, como no caso dos métodos estáticos.

Vale a pena ressaltar que, para uso da diretiva *override*, a assinatura dos métodos deve ser exatamente igual, pois de outra forma é apresentado um erro durante a compilação. Além disso, *override* somente pode ser utilizado em conjunto com *virtual* ou *dynamic*, não podendo ser utilizado em métodos estáticos.

Para casos de polimorfismo onde a assinatura do método é diferente nos seus descendentes, podem ser utilizadas duas outras diretivas: *overload* e *reintroduce*.

A diretiva *overload* pode ser utilizada para qualquer tipo de método, seja estático, virtual ou dinâmico. Para métodos virtuais pode-se utilizar a diretiva *reintroduce* em conjunto para os descendentes. Além de permitir assinaturas diferentes de um mesmo método, a diretiva *overload* não oculta as implementações do método nos ancestrais, estando assim disponíveis mais de uma implementação simultaneamente.

Para ativar a implementação correta, são analisados os parâmetros utilizados na chamada do método, executando assim a implementação que corresponder a esses parâmetros. Porém, para os métodos virtuais, caso seja necessário, os métodos dos ancestrais podem ser ocultados, utilizando a diretiva *reintroduce*.

A **Listagem 3** apresenta um exemplo do uso das diretivas *overload* e *reintroduce*.

Nota-se a presença de uma hierarquia de classes, onde a classe *Funcionario* é ancestral direta de *FuncionarioMensalista* (linha 08). Na subclasse são alterados os métodos *CalcularSalario* e *CalcularPremio* anteriormente existentes, de forma que tenham parâmetros diferentes da classe ancestral (linhas 11 e 12). Percebe-se também, que no nível mais alto dessa hierarquia, ou seja, na classe *Funcionario*, o método *CalcularSalario* é estático (linha 05) e o método *CalcularPremio* é virtual (linha 06).

Na classe descendente são utilizadas as diretivas *overload* e *reintroduce*, onde o método *CalcularPremio* possui a diretiva *reintroduce* (linha 12), visto que é um método virtual

### Listagem 3. Hierarquia com uso de overload e reintroduce

```
01: type
02:   Funcionario = class
03:     {Definições existentes}
04:   public
05:     function CalcularSalario: real;
06:     function CalcularPremio: real; virtual;
07:   end;
08:   FuncionarioMensalista = class (Funcionario)
09:     {Definições existentes}
10:   public
11:     function CalcularSalario(pHoraExtra: real): real; overload;
12:     function CalcularPremio(pTaxa: real): real; reintroduce;
13:   end;
14: implementation
15: {Implementações existentes}
16: function FuncionarioMensalista.CalcularSalario(pHoraExtra: real): real;
17: begin
18:   result := fValorMes + pHoraExtra;
19: end;
20: function FuncionarioMensalista.
  CalcularPremio(pTaxa: real): real;
21: begin
22:   result := fValorMes * pTaxa;
23: end;
```

e deseja-se que a implementação desse método no seu ancestral seja ocultada, ou seja, não disponível nessa classe nem em seus descendentes.

É necessário ressaltar que essas modificações feitas para a classe *FuncionarioMensalista* devem ser repetidas para a classe *FuncionarioDiarista*. A **Figura 1** demonstra os métodos disponíveis para a classe *FuncionarioMensalista* após a modificação das diretivas.

Na Figura anterior, observa-se que está disponível uma única implementação do método *CalcularPremio* e duas do método *CalcularSalario*. O método *CalcularSalario* tem a implementação da classe ancestral, ou seja, a classe *Funcionario*, graças à diretiva *overload*. Já o método *CalcularPremio* possui disponível apenas a implementação da própria classe *FuncionarioMensalista*, pois nesse método foi utilizada a diretiva *reintroduce*, ocultando a implementação do seu ancestral.

A diretiva *overload* pode ser utilizada também para a definição de métodos sobrecarregados dentro de uma mesma classe. Para isso, é necessário usar a diretiva na definição de todos os métodos desejados.

A diretiva *abstract* permite definir onde realmente devem ser implementados os métodos em uma hierarquia.

```
var
  func: FuncionarioMensalista;
begin
  func := FuncionarioMensalista.Create;
  func.
end;
end;
procedure AfterConstruction;
procedure BeforeDestruction;
function CalcularPremio(pTaxa: Real): Real;
function CalcularSalario(pHoraExtra: Real): Real;
function CalcularSalario: Real;
function ClassInfo: Pointer;
function ClassName: ShortString;
function ClassNames(const Name: String): Boolean;
function ClassParent: TClass;
function ClassType: TClass;
```

Figura 1. Métodos disponíveis utilizando reintroduce e overload

O seu uso implica que determinado método não tenha implementação na classe onde foi declarado. A implementação de fato deve ocorrer nos seus descendentes.

Ao definir pelo menos um método como abstrato, determina que a classe normalmente não vai ser instanciada e também será abstrata. Somente métodos virtuais ou dinâmicos podem ser abstratos. A **Listagem 4** demonstra o uso da diretiva *abstract*.

### Listagem 4. Uso da diretiva abstract

```
01: Funcionario = class
02:   {Definições existentes}
03: public
04:   destructor Destroy; virtual;
05:   function CalcularSalario: real; virtual; abstract;
06:   function CalcularPremio: real; virtual; abstract;
07: end;
08: FuncionarioMensalista = class (Funcionario)
09:   {Definições existentes}
10: public
11:   function CalcularSalario: real; override;
12:   function CalcularPremio: real; override;
13: end;
14: FuncionarioDiarista = class (Funcionario)
15:   {Definições existentes}
16: public
17:   destructor Destroy; override;
18:   function CalcularSalario: real; override;
19:   function CalcularPremio: real; override;
20: end;
21: implementation
22: {Implementações existentes}
23: // function Funcionario.CalcularSalario: real;
24: // begin
25: // {Código comum aos descendentes da classe}
26: // end;
27: // function Funcionario.CalcularPremio: real;
28: // begin
29: // {Código comum aos descendentes da classe}
30: // end;
31: function FuncionarioMensalista.CalcularSalario: real;
32: begin
33:   result := fValorMes;
34: end;
35: function FuncionarioMensalista.
  CalcularPremio: real;
36: begin
37:   result := fValorMes * 0.05;
38: end;
39: function FuncionarioDiarista.
  CalcularSalario: real;
40: begin
41:   result := fValorDia * fNumDias;
42: end;
43: function FuncionarioDiarista.
  CalcularPremio: real;
44: begin
45:   result := fValorDia * fNumDias * 0.05;
46: end;
```

Na **Listagem 4**, tem-se uma hierarquia de classes, sendo as classes *FuncionarioMensalista* e *FuncionarioDiarista*, descendentes da classe *Funcionario* (linhas 08 e 14). Nesse caso, a classe *Funcionario* possui os seus métodos *CalcularSalario* e *CalcularPremio* definidos como abstratos (linhas 05 e 06). Assim, a classe *Funcionario* torna-se uma classe abstrata, não permitindo que objetos sejam instanciados diretamente dela, obrigando aos seus descendentes a possuir implementação para tais métodos.

As implementações desses métodos nas classes *FuncionarioMensalista* e *FuncionarioDiarista* devem ser alteradas de acordo com as linhas 31 a 46 da **Listagem 4** devendo, ainda, remover tais implementações na classe *Funcionario*, pois caso contrário ocorrerá um erro de compilação, visto que métodos abstratos só possuem implementação nos

descendentes da classe (linhas 23 a 30).

O uso de *abstract* torna-se interessante, por exemplo, em casos onde se deseja realizar a chamada de um método comum a várias classes a partir do seu ancestral. Dessa forma, o ancestral define um método abstrato e os seus descendentes implementam os códigos específicos que serão executados.

A **Listagem 5** demonstra este exemplo. Para o cálculo do salário novamente se pressupõe que os valores dos campos de cada objeto tenham sido modificados logo após sua criação.

#### Listagem 5. Fazendo um typecasting com uma classe abstrata

```
01: var
02:   funcionarios: TList;
03:   objFunc1: FuncionarioMensalista;
04:   objFunc2: FuncionarioDiarista;
05:   i: integer;
06: begin
07:   funcionarios := TList.Create;
08:   objFunc1 := FuncionarioMensalista.Create;
09:   funcionarios.Add(objFunc1);
10:   objFunc2 := FuncionarioDiarista.Create;
11:   funcionarios.Add(objFunc2);
12:   for i := 0 to funcionarios.Count - 1 do
13:     begin
14:       Funcionario(funcionarios[i]).CalcularSalario;
15:     end;
16: end;
```

Na **Listagem 5**, nas linha 03 e 04, são declaradas variáveis que armazenam objetos de classes diferentes, porém descendentes da classe *Funcionario*. Além disso, declara-se a variável *funcionarios*, que é uma lista de objetos (*TList*), que guardará as referências dos objetos instanciados (linha 02). Na linha 07 a lista de funcionários é criada e, entre as linhas 08 e 11, são criadas instâncias de cada uma das classes e armazenadas na lista.

Após isso, a lista é percorrida calculando o salário dos funcionários nela armazenados (linhas 12 a 15). A execução do cálculo do salário ocorre na linha 14. Percebe-se que, independente de qual tipo de funcionário, ou seja, mensalista ou diarista, os mesmos podem se comportar de maneira geral como um funcionário (*typecast*), visto que herdam da classe *Funcionario*.

Dessa forma, a chamada do método abstrato *CalcularSalario*, feita à classe *Funcionario*, é redirecionada à implementação específica em cada uma das classes descendentes de acordo com o seu tipo, executando o algoritmo correto relativo à classe que instanciou o objeto. A partir do Delphi 8 existe um outro tipo de diretiva chamado *final*. Utilizar essa diretiva nos métodos de uma classe impede que os mesmos sejam redefinidos nas classes descendentes.

## Associação

As associações criam conexões entre as classes, através das quais essas podem se comunicar pelo envio de mensagens. As mensagens são chamadas dos membros públicos das classes. Associações entre classes podem ser do tipo 1-para-1, 1-para-N, N-para-N.

Na POO pode-se representar os dois lados de uma associação, ou apenas um deles, de acordo com o sentido do fluxo de mensagens desejado, ou seja, a navegabilidade. No contexto deste exemplo, será utilizada navegabilidade bidirecional, uma vez que não está definida a navegabilidade no modelo anteriormente apresentado.

Nesse tipo de navegabilidade qualquer dos objetos da associação possui referência ao outro objeto, ou seja, de qualquer um dos lados da associação é possível enviar mensagens ao outro.

## LIVROS DE DELPHI



**Estudo Dirigido de Delphi 2005**  
Editora Érica  
R\$ 76,00

**Delphi 2005: Aplicações com Banco de Dados com Interbase 7.5 e MySQL 4.0.23**

Editora Érica  
R\$ 145,00



**Guia do Desenvolvedor de Delphi for .NET**  
Makron Books  
R\$ 165,00

**Rave Report com Delphi**

Ciência Moderna  
R\$ 25,00



**Universidade Delphi**  
Digerati Books  
R\$ 49,90

Delphi: Progra. Banco de Dados e Web - R\$69,00  
Delphi 7 & E-commerce: Trein. Inter. CD - R\$381,00  
Delphi - R\$39,00  
Aplic. das Estruturas de Dados em Delphi - R\$59,00  
Impl. .NET no Delphi 8: Uma Visão Geral - R\$31,00  
CD com 50 Program./Ex. Fon. p/ Delphi - R\$29,90  
Delphi 8 Para Plat. .NET: Curso Completo- R\$249,00  
Program. em Delphi 6: Orient. por Projeto - R\$56,00  
Int. ao Desen. de Aplicações em Delphi - R\$50,00  
Conhecendo e Trabalhando com Delphi 8 - R\$80,00  
Estudo Dirigido de Delphi 8 - R\$58,00  
Pr Pascal: Ling. do Turbo Pascal do Delphi -R\$52,00  
Kylx: Delphi Linux com Interbase/Firebird - R\$52,00  
Delphi 6: Conceitos Básicos (E-Book) - R\$27,90  
Acessando MySQL com Delphi 7 (em CD) - R\$20,00  
Dominando o Delphi 7: A Bíblia - R\$189,00  
Redes Neurais em Delphi - R\$24,00  
Estudo Dirigido de Delphi 7: Avançado - R\$65,00  
CLX Portabilidade com Delphi 7 e Kylx 3 - R\$39,00  
Delphi 7: Apl. Avan. de Banco de Dados - R\$87,00  
Delphi 7: Conceitos Básicos - R\$42,00  
Estudo Dirigido de Delphi 7 - R\$74,00  
Programação Gráfica em Delphi 6 - R\$50,00  
Delphi/Kylx: Desenv. de Banco de Dados - R\$72,00  
Boleto Bancário em Delphi - R\$35,00  
Conectividade Utilizando Delphi 6 - R\$40,00



**FRETE GRÁTIS**  
Para todo o Brasil!

**LivrosdeProgramacao.com.br**

Para que as associações sejam estabelecidas de fato, normalmente é necessário que as classes recebam novos membros, mais especificamente campos, que correspondam ao tipo da outra classe com a qual está se associando. Tais campos podem representar uma única instância de outra classe, como também uma lista dessas, estabelecendo assim a cardinalidade desejada.

A **Listagem 6** apresenta a definição e implementação dos métodos `getDescricaoDepartamento` e `getNomeFuncionarios`, os quais permitem a troca de mensagens entre essas classes.

**Listagem 6. Estabelecendo uma associação de classes 1-para-N**

```

01: type
02: Funcionario = class
03: private
04:   fNome: string;
05:   fDepartamento: Departamento;
06: public
07:   function getDescricaoDepartamento: string;
08:   {Demais definições existentes}
09: published
10:   property Nome: string read getNome
      write setNome;
11:   property Departamento: Departamento
      read getDepartamento write setDepartamento;
12: end;
13: Departamento = class
14: private
15:   fDescricao: string;
16:   fFuncionarios: TList;
17: public
18:   function getNomeFuncionarios: TStringList;
19:   {Demais definições existentes}
20: published
21:   property Descricao: string
      read getDescricao write setDescricao;
22: end;
23: implementation
24:   function Funcionario.
      getDescricaoDepartamento: string;
25: begin
26:   result := fDepartamento.Descricao;
27: end;
28:   function Departamento.
      getNomeFuncionarios: TStringList;
29: var
30:   i: integer;
31: begin
32:   result := TStringList.Create;
33:   for i:= 0 to fFuncionarios.Count -1 do
34:     begin
35:       result.add(Funcionario(fFuncionarios[i]).Nome);
36:     end;
37: end;
38: end.

```

Na **Listagem 6** pode-se perceber a presença dos métodos `getDescricaoDepartamento` e `getNomeFuncionarios`, nas linhas 07 e 18, respectivamente. Esses métodos são responsáveis pela troca de mensagens entre as classes envolvidas, solicitando informações uma da outra, visto que o acesso direto à informação não é possível, pois os campos das classes são privados.

Com a execução do método `getDescricaoDepartamento` (linhas 24 a 27) a partir de uma instância da hierarquia da classe `Funcionario`, pode-se obter a descrição do `Departamento` daquele funcionário. Já, nas linhas 28 a 37, encontra-se a implementação do método `getNomeFuncionarios` que tem como resultado uma lista com os nomes dos funcionários instanciados e armazenados por um objeto da classe `Departamento`.

Para associar um funcionário a um departamento utiliza-se o serviço `addFuncionario` da classe `Departamento`. Por outro

lado, para associar um departamento a um funcionário, utiliza-se o serviço `setDepartamento` da classe `Funcionario` ou a propriedade `Departamento` da mesma classe.

As associações de classes são indispensáveis na construção de sistemas Orientados a Objetos. Normalmente essas classes são dependentes e necessitam comunicar-se através da troca de mensagens.

## Interfaces

Através de *interfaces* pode-se obter uma outra forma de definir e implementar métodos comuns a várias classes, pertencentes a uma hierarquia ou não. *Interfaces* são conjuntos de definições de métodos que podem ser implementados por diferentes classes. Dessa forma, as interfaces são apenas definições de métodos, isso quer dizer que não possuem implementações, sendo as classes as responsáveis por implementar os métodos definidos na interface.

Por exemplo, a hierarquia de classes de funcionários, pode implementar uma interface para calcular seus encargos trabalhistas. Assim, essas classes passam a ser obrigadas a implementar os métodos definidos pela interface.

Na **Listagem 7**, apresenta-se a definição da interface chamada `IEncargos`, a qual pode definir diferentes métodos que tratam do cálculo de encargos. Neste exemplo apresenta-se apenas um único método chamado `ContribuicaoSindical`.

Ainda nessa listagem, apresentam-se as modificações necessárias nas classes da hierarquia de funcionários (a partir da linha 6), especificamente nas classes `Funcionario` e `FuncionarioMensalista`, para que essas passem a implementar tal interface (não se deve esquecer de declarar no *uses* da seção *interface* da *untClasses* a unit da interface criada).

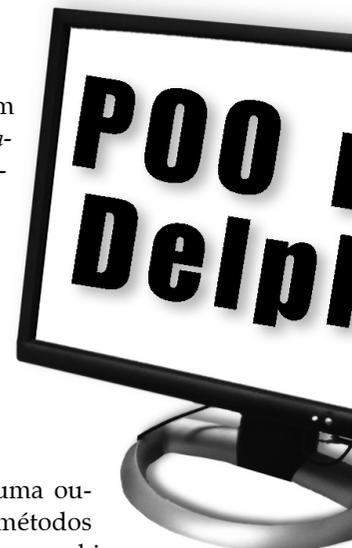
**Listagem 7. Usando Interfaces**

```

01: type
02:   IEncargos = interface
03:   [ '{41E3330E-8707-4544-87EE-B70EE7911C7A}' ]
04:   function ContribuicaoSindical: real;
05: end;
06: Funcionario = class(TInterfacedObject, IEncargos)
07:   {Definições existentes}
08: public
09:   function ContribuicaoSindical: real; virtual; abstract;
10: end;
11: FuncionarioMensalista = class (Funcionario)
12:   {Definições existentes}
13: public
14:   function ContribuicaoSindical: real; override;
15: end;
16: FuncionarioDiarista = class (Funcionario)
17:   {Definições existentes}
18: public
19:   function ContribuicaoSindical: real; override;
20: end;

```

Na **Listagem 7**, da linha 02 a 05, encontra-se a declaração da interface `IEncargos`. Por padrão do Delphi, as interfaces começam com a letra "I". Essa interface define o método



chamado *ContribuicaoSindical*, podendo ter outros métodos conforme a necessidade.

Assim como objetos descendem direta ou indiretamente da classe *TObject* do Delphi, as interfaces descendem de *IInterface*. Toda interface possui os métodos *QueryInterface*, *\_AddRef* e *\_Release*, que também devem ser implementados. Esses métodos tratam do gerenciamento e consulta dinâmica à interface. Para evitar essa codificação adicional pode-se determinar que classes que implementam interfaces, herdem direta ou indiretamente da classe *TInterfaceObject* do Delphi. Assim, os métodos *QueryInterface*, *\_AddRef* e *\_Release* já estarão implementados.

Além disso, as interfaces podem opcionalmente possuir um identificador do tipo GUID (*Globally Unique Identifier*) como mostrado na linha 03. O GUID é uma expressão em hexadecimal e pode ser gerada em tempo de design através do pressionamento das teclas Ctrl+Shift+G. Com o uso desse identificador pode-se obter referência à implementação de uma interface através de *QueryInterface*, usando por exemplo o operador *As*.

Finalmente, para determinar que uma classe implementa uma interface é necessário modificar a sua definição. Dessa forma, à frente da palavra *class* adiciona-se o nome da interface desejada, e como recomendado, estabelecendo a herança da classe *TInterfacedObject*. Tal modificação pode ser percebida na linha 06.

Vale a pena ressaltar que, essa forma de implementar interfaces no Delphi não se trata de herança múltipla. A definição da classe *Funcionario* apenas determina que essa passa a herdar da classe *TInterfacedObject* e implementar obrigatoriamente a interface *IEncargos*. Herança múltipla aconteceria se uma classe herdasse simultaneamente de duas ou mais classes, situação não permitida no Delphi. Sendo assim, uma classe pode herdar de apenas uma outra classe, mas pode implementar diversas interfaces ao mesmo tempo.

Após essas modificações, é necessário ainda adicionar à definição das classes *Funcionario*, *FuncionarioMensalista* e *FuncionarioDiarista* os métodos definidos pela interface, nesse caso *ContribuicaoSindical*. A assinatura dos métodos deve corresponder exatamente como definido na interface, como pode ser visto nas linhas 09, 14 e 19.

Deve-se lembrar que, devido à classe *Funcionario* implementar a interface *IEncargos*, ou ela ou todos os seus descendentes devem obrigatoriamente implementar os métodos definidos pela interface. O código da **Listagem 8** apresenta a implementação do método *ContribuicaoSindical* nas classes *FuncionarioMensalista* e *FuncionarioDiarista*.

#### Listagem 8. Implementação do método *ContribuicaoSindical*

```
01: function FuncionarioMensalista.ContribuicaoSindical: real;
02: begin
03:   result := fValorMes/30;
04: end;
05: function FuncionarioDiarista.ContribuicaoSindical: real;
06: begin
07:   result := (fValorDia * NumDias)/30;
08: end;
```

O código da **Listagem 8** retorna o valor da contribuição sindical que é correspondente, nesse caso, ao valor de um dia de trabalho em um mês de 30 dias. A chamada do método pode ser feita normalmente a partir do objeto, como qualquer outro tipo de método, visto que é implementado pela própria classe.

Porém, outras formas podem ser usadas para executar o método. A **Listagem 9** demonstra um trecho de código onde se apresentam outras duas formas de chamada do método *ContribuicaoSindical*, utilizando a interface criada. Essa listagem deve ser implementada fora da *untClasses*, em outra unit que a referencie.

#### Listagem 9. Chamadas do método *ContribuicaoSindical*

```
01: var
02:   Encargos: IEncargos;
03:   Func: FuncionarioMensalista;

04: begin
05:   {Primeira forma de chamada do método}
06:   Encargos := FuncionarioMensalista.Create;
07:   Encargos.ContribuicaoSindical;
08:   {Segunda forma de chamada do método}
09:   Func := FuncionarioMensalista.Create;
10:   (Func as IEncargos).ContribuicaoSindical;

11: end;
```

A **Listagem 9** apresenta um código correspondente para realização de chamadas de métodos implementados através de uma interface. Na primeira forma utiliza-se uma variável do tipo da interface (linha 02). Como a classe *FuncionarioMensalista* implementa a interface, tal variável pode receber a referência de um objeto dessa classe. Pode-se assim chamar o método através da interface (linhas 05 a 07).

A segunda forma pode ser realizada por meio de uma variável do tipo da classe *FuncionarioMensalista* (linha 03). Dessa forma, a variável recebe uma instância dessa classe. Posteriormente, por meio do operador *As*, a chamada é feita utilizando a interface (linhas 08 a 10).

Essa forma só é possível se a interface possuir o identificador do tipo GUID, como visto anteriormente. Caso contrário ocorrerá um erro, pois a interface utiliza o identificador para obter referência às suas implementações. Essa forma é conhecida como *interface querying*.

## Estudo de Caso

A seguir, será apresentada uma aplicação simples que utiliza a maioria dos conceitos abordados, bem como o modelo anteriormente apresentado. Para isso, será adicionada uma outra classe chamada *Aplicacao* à unit *untClasses*, que manterá uma lista de funcionários e uma lista de departamentos, com o objetivo de simular a persistência (armazenamento) dos objetos.

Essa classe ainda apresenta alguns métodos que atuam nos conjuntos dos objetos. A **Listagem 10** apresenta o código da classe, com os métodos necessários para manipulação das listas.

#### Listagem 10. Classe Aplicacao

```
01: type
02:   Aplicacao = class
03:   private
04:     fDepartamentos: TList;
05:     fFuncionarios: TList;
06:   public
07:     constructor Create;
08:     destructor Destroy;
09:     procedure addDepartamento(pDepartamento: Departamento);
10:     procedure addFuncionario(pFuncionario: Funcionario);
11:     procedure ExcluirFuncionario(pIndice: integer);
12:     function getDescricaoDepartamentos: TStringList;
13:     function getDepartamento(pIndice: integer): Departamento;
14:     function getFuncionarios: TList;
15:     function getNomeFuncionarios: TStringList;
16:   end;
17: var
18:   aplic: Aplicacao;
19: implementation
20: constructor Aplicacao.Create;
21: begin
22:   fDepartamentos := TList.Create;
23:   fFuncionarios := TList.Create;
24: end;
25: destructor Aplicacao.Destroy;
26: begin
27:   fDepartamentos.Free;
28:   fFuncionarios.Free;
29: end;
30: procedure Aplicacao.addDepartamento(pDepartamento: Departamento);
31: begin
32:   fDepartamentos.Add(pDepartamento);
33: end;
34: procedure Aplicacao.addFuncionario(pFuncionario: Funcionario);
35: begin
36:   fFuncionarios.Add(pFuncionario);
37: end;
38: procedure Aplicacao.ExcluirFuncionario(pIndice: integer);
39: begin
40:   Funcionario(fFuncionarios[pIndice]).Destroy;
41:   fFuncionarios.Delete(pIndice);
42: end;
43: function Aplicacao.getDepartamento(pIndice: integer): Departamento;
44: begin
45:   Result := Departamento(fDepartamentos[pIndice]);
46: end;
47: function Aplicacao.getDescricaoDepartamentos: TStringList;
48: var
49:   i: integer;
50: begin
51:   result := TStringList.Create;
52:   for i := 0 to fDepartamentos.Count - 1 do
53:     begin
54:       result.Add(Departamento(fDepartamentos[i]).Descricao);
55:     end;
56: end;
57: function Aplicacao.getFuncionarios: TList;
58: begin
59:   result := fFuncionarios;
60: end;
61: function Aplicacao.getNomeFuncionarios: TStringList;
62: var
63:   i: integer;
64: begin
65:   result := TStringList.Create;
66:   for i := 0 to fFuncionarios.Count - 1 do
67:     begin
68:       result.Add(Funcionario(fFuncionarios[i]).Nome);
69:     end;
70: end;
```

Na **Listagem 10** temos a definição e implementação da classe *Aplicacao*. Pode-se destacar nas linhas 04 e 05 a definição das listas de funcionários e departamentos que serão mantidos por uma única instância da classe *Aplicacao*.

Os métodos *addDepartamento* (linhas 30 a 33) e *addFuncionario* (linhas 34 a 37) permitem adicionar objetos às respectivas listas, bem como os métodos *getDescricaoDepartamentos* (linhas 47 a 56) e *getNomeFuncionarios* (linhas 61 a 70) retornam listas de *strings* com as descrições de departamentos e nomes dos funcionários armazenados nas respectivas listas de objetos.

O método *ExcluirFuncionario* (linhas 38 a 42) permite a exclusão de um funcionário, além de removê-lo da lista de

funcionários mantidos pela aplicação, tendo ainda o cuidado de desfazer a associação com o objeto associado da classe *Departamento*.

O método *getFuncionarios* (linhas 57 a 60) retorna a lista de objetos de funcionários que será útil para a realização dos cálculos dos funcionários armazenados pela aplicação. Já o método *getDepartamento* (linhas 43 a 46) retorna apenas um departamento de acordo com o índice do objeto armazenado na lista, sendo útil para determinar qual objeto foi selecionado a partir de um *ComboBox*, por exemplo. Por fim, os métodos *constructor* (linhas 20 a 24) e *destructor* (linhas 25 a 29) instanciam e liberam as listas da memória, respectivamente.

Um detalhe importante a respeito deste exemplo, trata-se da variável *aplic* (linha 18), que disponibiliza uma instância da classe *Aplicacao*, permitindo compartilhar as mesmas listas de funcionários e departamentos em toda a aplicação.

A **Figura 2** demonstra algumas opções do menu do formulário principal da aplicação que permitem o acesso aos demais formulários do sistema, devendo ser criado a partir de uma nova aplicação no Delphi.

As funcionalidades oferecidas pelo exemplo são cadastros dos departamentos e funcionários (no item *Cadastros* adicione “Departamentos”, “Funcionários” e “Excluir Funcionários”), consultas aos funcionários de determinado departamento e calcular salários de cada funcionário. Todos esses formulários devem referenciar na sua cláusula *uses* a unit *untClasses*.

Esse formulário principal, por ser iniciado juntamente com a aplicação, tem a responsabilidade de instanciar o objeto da classe *Aplicacao*, referenciado pela variável *aplic*, definida na unit *untClasses*. A **Listagem 11** demonstra os códigos dos eventos *OnClose* e *OnCreate* desse formulário.

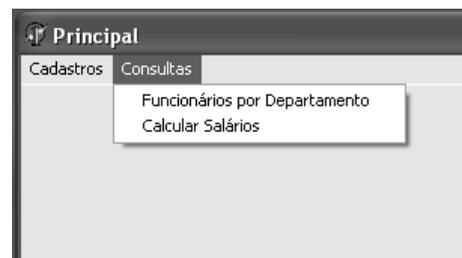


Figura 2. Formulário Principal da aplicação

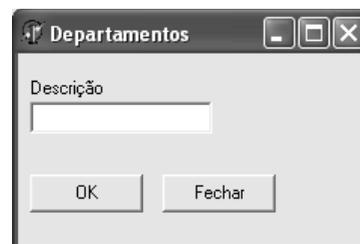


Figura 3. Formulário de cadastro de Departamentos

**Listagem 11. Eventos OnCreate e OnClose do formulário principal da aplicação**

```

procedure TfrmPrincipal.FormCreate(Sender: TObject);
begin
    aplic := Aplicacao.Create;
end;
procedure TfrmPrincipal.FormClose(Sender: TObject;
    var Action: TCloseAction);
begin
    aplic.Free;
end;

```

Na **Listagem 11** pode-se perceber a criação da instância da classe *Aplicacao* e atribuição à variável *aplic*. Ainda, nessa listagem, apresenta-se a liberação da variável da memória ao fechar o formulário, ou seja, ao finalizar a aplicação. A **Figura 3** apresenta o formulário de cadastro de departamentos, que permite a inclusão de novos departamentos na aplicação.

O cadastro de departamentos é um formulário bastante simples, com apenas um campo (“edtDescricao”), conforme definido na classe *Departamento*. A **Listagem 12** apresenta o código responsável pela inclusão de um novo departamento, implementado no botão OK.

**Listagem 12. Código do botão OK do cadastro de departamentos**

```

01: var
02:   depto: Departamento;
03: begin
04:   depto := Departamento.Create;
05:   depto.Descricao := edtDescricao.Text;
06:   aplic.addDepartamento(depto);
07:   edtDescricao.Clear;
08:   edtDescricao.SetFocus;
09: end;

```

O código apresentado na **Listagem 12** demonstra a criação de uma nova instância da classe *Departamento* e sua atribuição à variável local *depto* (linha 04). Após a propriedade *Descricao* do objeto, receber o texto digitado pelo usuário (linha 05), a nova instância de *Departamento* é adicionada à lista de departamentos mantida pela aplicação (linha 06).

As linhas seguintes limpam o campo em tela e posicionam o cursor para uma nova inclusão. O usuário pode incluir quantos departamentos forem necessários, sabendo-se que os mesmos estão armazenados somente em memória para efeito de exemplo.

A **Figura 4** apresenta o formulário de cadastro de funcionários, permitindo adicionar novos funcionários do tipo mensalista ou diarista, além de associá-los a um determinado departamento. Deve-se atentar para os nomes dos componentes, de forma a evitar erros de compilação (**Listagem 13**).

**Figura 4.** Formulário de cadastro de Funcionários

Nesse formulário são incluídas ainda as demais informações dos funcionários, como os valores que serão utilizados nos cálculos de salário. Além disso, associa-se um departamento ao funcionário, bem como o adiciona à lista de funcionários daquele departamento.

Embora pareça uma redundância, isso é necessário para representar a navegabilidade bi-direcional da associação. Pois, a partir de um funcionário pode-se obter seu departamento, e da mesma forma, a partir de um departamento pode-se obter a lista de seus funcionários.

A **Listagem 13** demonstra o código do formulário de cadastro de funcionários, responsável por realizar a tarefa de inclusão.

**Listagem 13. Código do formulário de cadastro de funcionários**

```

01: procedure TfrmFuncionario.FormActivate(Sender: TObject);
02: begin
03:   cmbxDepartamento.Items := aplic.getDescricaoDepartamentos;
04: end;
05: procedure TfrmFuncionario.btnOKClick(Sender: TObject);
06: var
07:   func: Funcionario;
08:   depto: Departamento;
09: begin
10:   case rdgrpTipoFuncionario.ItemIndex of
11:     0: begin
12:       func := FuncionarioMensalista.Create;
13:       FuncionarioMensalista(func).ValorMes :=
14:         StrToFloat(edtMensalistaValorMes.Text);
15:     end;
16:     1: begin
17:       func := FuncionarioDiarista.Create;
18:       FuncionarioDiarista(func).ValorDia :=
19:         StrToFloat(edtDiaristaValorDia.Text);
20:       FuncionarioDiarista(func).NumDias :=
21:         StrToInt(edtDiaristaNumDias.Text);
22:     end;
23:   else
24:     begin
25:       ShowMessage('Tipo de funcionário não selecionado');
26:     end;
27:   if Assigned(depto) then
28:     begin
29:       func.Departamento := depto;
30:       depto.addFuncionario(func);
31:     end;
32:   aplic.addFuncionario(func);
33:   edtNome.Clear;
34:   cmbxDepartamento.ItemIndex := -1;
35:   edtMensalistaValorMes.Clear;
36:   edtDiaristaValorDia.Clear;
37:   edtDiaristaNumDias.Clear;
38:   edtNome.SetFocus;
39: end;

```

Na **Listagem 13** apresentam-se dois procedimentos, um para o evento *OnActivate* do formulário (linhas 01 a 04) e outro para o *click* do botão OK (linhas 5 a 39). No momento em que o formulário é criado, é necessário o preenchimento do *ComboBox* (“cmbxDepartamento”) com as descrições dos departamentos anteriormente cadastrados. Para isso, é acionado o método *getDescricaoDepartamentos*, do objeto *aplic*, que retorna uma lista de *strings* contendo os nomes dos departamentos (linha 03), atribuído ao *ComboBox*.

Para o código do botão OK, são necessárias duas variáveis para trabalhar com instâncias de *Funcionario* e *Departamento*, declaradas nas linhas 07 e 08, respectivamente. Como o funcionário pode ser mensalista ou diarista, a classe que define a variável *func* é a classe ancestral *Funcionario*

(linha 07), tornando o procedimento genérico, independente do tipo do funcionário.

Após a verificação da escolha feita pelo usuário de um dos tipos de funcionário, é instanciado um objeto da classe descendente apropriada, bem como são atribuídos os valores às propriedades por meio de um *typecasting* (linhas 10 a 24).

Na linha 26, a variável *depto* recebe a referência do objeto de *Departamento*, armazenado na lista da aplicação, de acordo com o índice do item selecionado no *ComboBox*, através da execução de *getDepartamento* da classe *Aplicacao*. Na linha 27 é testada a referência do objeto retornado utilizando a função *Assigned*, que testa se a referência aponta para um objeto ou para *nil* (sem referência). De posse de uma referência válida do objeto da classe *Departamento*, é feita a associação 1-para-N com *Funcionario* de forma bidirecional. Ou seja, na linha 29, o funcionário associa-se com o departamento e, na linha 30, o departamento recebe mais um funcionário em sua lista.

A linha 32 adiciona o funcionário recém criado na lista de funcionários da aplicação e, em seguida, os campos do formulário são preparados para uma nova inclusão. Completando a manutenção dos cadastros, a **Figura 5** apresenta o formulário de exclusão de funcionários.

No formulário apresentado na **Figura 5** é possível realizar a exclusão de funcionários cadastrados na aplicação. A exclusão implica não apenas em eliminar o objeto funcionário, mas também retirar sua referência da lista mantida pelo objeto *Departamento* associado.

Além disso, deve-se retirar também sua referência da lista de objetos de funcionários mantida pela aplicação. A **Listagem 14** demonstra o código responsável pela exclusão de funcionários.

#### Listagem 14. Código do formulário de exclusão de funcionários

```
01: procedure TfrmExcluirFuncionario.  
    FormActivate(Sender: TObject);  
02: begin  
03:   lstbxFuncionarios.Items := aplic.getNomeFuncionarios;  
04: end;  
05: procedure TfrmExcluirFuncionario.btnExcluirClick(Sender: TObject);  
06: begin  
07:   if lstbxFuncionarios.ItemIndex <> -1 then  
08:     begin  
09:       aplic.ExcluirFuncionario(lstbxFuncionarios.ItemIndex);  
10:       lstbxFuncionarios.Items := aplic.getNomeFuncionarios;  
11:     end;  
12: end;
```

Na **Listagem 14** estão dois eventos programados, *OnActivate* do formulário e *OnClick* do botão. No evento *OnActivate* (linhas 01 a 04) é feita a leitura dos nomes dos funcionários mantidos pela aplicação, listando-os na janela.

No evento *OnClick* (linhas 05 a 12) está o código responsável pela exclusão de um determinado objeto a partir do índice do elemento selecionado no *lstbxFuncionarios*. O método *ExcluirFuncionario* (**Listagem 10**) do objeto *aplic* é chamado para realizar a exclusão e, em seguida, é atualizada novamente a lista com os nomes dos funcionários restantes.

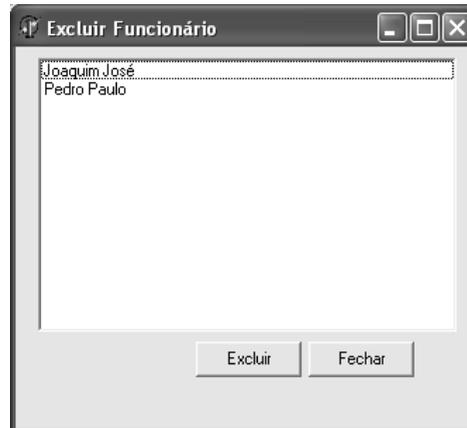


Figura 5. Formulário de exclusão de funcionários



Figura 6. Formulário de consulta de funcionários por departamento

A seguir, a **Figura 6** apresenta o formulário de consulta de funcionários por departamento.

No formulário de consulta de funcionários por departamento, como o próprio nome indica, são listados os nomes dos funcionários que fazem parte da lista de funcionários mantida por um objeto da classe *Departamento*. A **Listagem 15** apresenta o código para o funcionamento desse formulário. Deve-se atentar para os nomes dos componentes, de forma a evitar erros de compilação.

#### Listagem 15. Código do formulário de consulta de funcionários por departamento

```
01: procedure TfrmFuncionariosPorDepartamento.  
    FormActivate(Sender: TObject);  
02: begin  
03:   lstbxFuncionarios.Clear;  
04:   cmbxDepartamento.Items := aplic.getDescricaoDepartamentos;  
05: end;  
06: procedure TfrmFuncionariosPorDepartamento.  
    cmbxDepartamentoChange(Sender: TObject);  
07: var  
08:   deptoConsulta: Departamento;  
09: begin  
10:   lstbxFuncionarios.Clear;  
11:   deptoConsulta := aplic.getDepartamento(  
    cmbxDepartamento.ItemIndex);  
12:   lstbxFuncionarios.Items := deptoConsulta.getNomeFuncionarios;  
13: end;
```

Primeiramente, de maneira idêntica ao cadastro de funcionários, o *ComboBox* de departamentos (“cmbxDepartamento”) é preenchido com as descrições dos departamentos mantidos na lista da aplicação (linhas 01 a 05). Posteriormente, ao selecionar um departamento, o evento *OnChange* do *ComboBox* é acionado (linhas 06 a 13).

Com isso, a variável *deptoConsulta*, do tipo *Departamento*, recebe a referência ao departamento selecionado através do índice do *ComboBox* (linha 11). Por último, o método *getNomeFuncionarios* do objeto *Departamento* é acionado, retornando uma lista com nomes dos funcionários que estão associados àquele departamento. Essa lista é atribuída ao *ListBox* (“lstbxFuncionarios”) para apresentação (linha 12).

Por fim, a **Figura 7** apresenta o formulário de cálculos de salários através da lista de todos os funcionários mantidos pela aplicação com seus referidos cálculos, independente do departamento.

Através da **Figura 7** pode-se ver os nomes dos funcionários, seu tipo (nome da classe a partir da qual foi instanciado) e descrição do departamento em que está lotado, seguido dos resultados dos cálculos de salários, prêmios e contribuições sindicais, respectivamente, calculados em função da classe que instanciou o objeto.

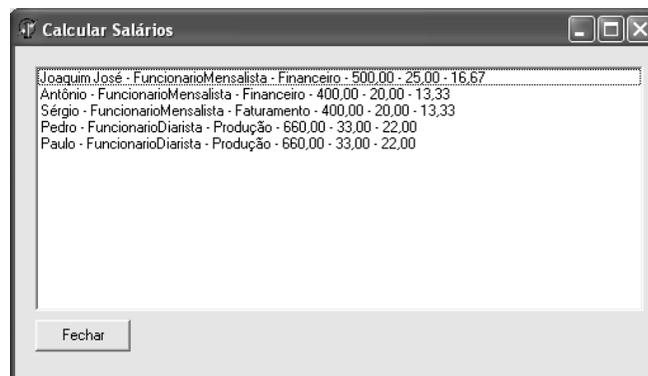
A **Listagem 16** apresenta o código para geração desses resultados. Deve-se novamente atentar para os nomes dos componentes, de forma a evitar erros de compilação.

#### Listagem 16. Códigos da tela de cálculo de salários

```
01: procedure TfrmCalcularSalarios.FormActivate(Sender: TObject);
02: var
03:   funcionarios: TList;
04:   i: integer;
05: begin
06:   lstbxSalarios.Clear;
07:   funcionarios := aplic.getFuncionarios;
08:   for i := 0 to funcionarios.Count - 1 do
09:     begin
10:       lstbxSalarios.Items.add(Funcionario(
11:         funcionarios[i].Nome + ' - ' +
12:         Funcionario(funcionarios[i]).ClassName +
13:         ' - ' + Funcionario(funcionarios[i]).
14:         getDescricaoDepartamento + ' - ' +
15:         FormatFloat('##0.00', Funcionario(
16:           funcionarios[i]).CalcularSalario) + ' - ' +
17:         FormatFloat('##0.00', Funcionario(
18:           funcionarios[i]).CalcularPremio) + ' - ' +
19:         FormatFloat('##0.00', Funcionario(
20:           funcionarios[i]).ContribuicaoSindical));
21:     end;
22: end;
```

Na **Listagem 16** apresenta-se o código correspondente ao evento *OnActivate* do formulário de cálculo de salários. Esse código é responsável por acionar os cálculos e apresentar as informações de todos os funcionários mantidos pela aplicação. Para isso utiliza-se de uma variável do tipo *TList* para referenciar a lista dos funcionários (linha 07), obtida a partir do objeto *aplic* que representa a aplicação.

Essa lista é percorrida para que cada funcionário seja processado e seus dados sejam adicionados ao *ListBox* (“lstbxSalarios”). Na linha 10 é recuperada a informação do nome do funcionário através da propriedade *Nome* da classe *Funcionario*.



**Figura 7.** Formulário de consulta aos cálculos de salários de funcionários

Utiliza-se o método *ClassName* para obter o nome da classe que define o objeto (linha 11). Recupera-se a descrição do departamento associado ao funcionário, por meio do método *getDescricaoDepartamento* da classe *Departamento* (linha 12). São calculados e recuperados os valores de salário, prêmio e contribuição sindical de cada funcionário (linhas 13 a 15). Percebe-se nessa listagem que é utilizada apenas a classe ancestral *Funcionario* para fazer referências aos objetos, não sendo necessário referenciar cada tipo de funcionário específico.

Essa é uma característica marcante de programação Orientada a Objetos. Exceto pelo momento da instanciação, não é preciso verificar o tipo de um objeto numa hierarquia, pois esse é o papel das subclasses e o polimorfismo possibilita que os métodos sejam chamados corretamente em função do tipo do objeto. Com isso, economizam-se linhas de código e verifica-se um aumento significativo da legibilidade e reuso do mesmo.

Pode-se perceber ainda que o código de cada formulário é bem sucinto. Isso é uma outra característica da programação Orientada a Objetos, pois, como grande parte do código é implementado nas classes, a programação dos formulários tende a tornar-se bastante simples.

Para visualizar o funcionamento da aplicação, basta realizar a chamada dos respectivos formulários no formulário principal do projeto.

## Conclusão

A programação orientada a objetos apresenta-se atualmente como uma forma eficaz para o desenvolvimento de software, e a ampla compreensão deste paradigma torna-se fundamental para que seja explorado eficientemente.

Apresentamos nestes dois artigos os conceitos mais importantes de POO, aplicando-os em um estudo de caso simples em Delphi, mas que explora a maioria destes conceitos.

Como dito anteriormente, não se teve o objetivo de ser completo em relação a esse assunto. Conceitos importantes, como persistência de objetos, não foram explorados e merecem um estudo aprofundado. ■