

Desenvolvimento de aplicações OO no Delphi

Uma abordagem prática - Parte 1



JOÃO CARLOS DA SILVA

(jcsilva@pos.granbery.edu.br)

é Graduado em Sistemas de Informação pela Faculdade Metodista Granbery e Pós-Graduando em Ciência da Computação pela Universidade Federal de Viçosa.



MARCELO SANTOS DAIBERT

(mdaibert@npq.granbery.edu.br)

é Graduado em Sistemas de Informação pela Faculdade Metodista Granbery e Pós-Graduando em Ciência da Computação pela Universidade Federal de Viçosa.



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, Especialista em Métodos Estatísticos Computacionais e Bacharel em Informática pela UFJF, Analista de Sistemas da Prefeitura de Juiz de Fora.

Com o passar dos anos, o paradigma Orientado a Objetos tem sido extensivamente utilizado na construção de sistemas de software e tornou-se um dos padrões no desenvolvimento de aplicações. São vários os benefícios que esse paradigma traz à equipe de desenvolvimento e talvez a isso se deva o seu sucesso.

No entanto, para realmente fazer uso dos benefícios da Orientação a Objetos, como reutilização de código, manutenibilidade do sistema, entre outros, é necessário que a equipe tenha domínio de assuntos co-relacionados como padrões de projeto, da linguagem de programação utilizada, persistência e principalmente entender os conceitos da Orientação a Objetos.

Atualmente, um dos obstáculos no desenvolvimento de aplicações OO é a persistência de objetos. Os bancos de dados puramente Orientados a Objetos são de fato os mais adequados para a persistência, mas a indisponibilidade atual desses, seja devido ao custo, diversidade e principalmente amadurecimento, faz com que seja necessária uma busca por alternativas para a realização dessa tarefa.

Uma das soluções encontradas para o problema é a utilização de camadas de persistência para manipulação dos objetos utilizando bancos de dados relacionais. Essa abordagem vem sendo amplamente utilizada no mercado, principalmente para o desenvolvimento de sistemas de médio a grande porte.

Basicamente une a rapidez e o amadurecimento das bases de dados relacionais com os benefícios da Programação Orientada a Objetos (POO). A função principal de uma camada de persistência é portar transparentemente os objetos de uma aplicação para uma base de dados relacional de forma genérica.

Para isso, são utilizadas principalmente técnicas de mapeamento objeto-relacional com o intuito de mapear as classes e associações para tabelas e relacionamentos. A utilização de uma camada de persistência, embora pareça trivial inicialmente, se mostra complexa na prática, uma vez que são várias as configurações que

devem ser feitas para que a camada funcione corretamente, além de outros fatores que dificultam sua utilização.

Utilizando esse mesmo raciocínio, uma outra solução a abordada neste artigo, é a própria aplicação acessar o banco de dados relacional e realizar o mapeamento objeto-relacional, persistindo seus objetos. Com isso, a aplicação é capaz de persistir e manipular os objetos em bancos de dados relacionais de forma rápida e transparente, não necessitando de uma camada de persistência desenvolvida por terceiros.

Buscando entender essa abordagem, o objetivo deste artigo é exemplificar o uso prático do paradigma Orientado a Objetos no desenvolvimento de uma aplicação no ambiente Delphi 7, utilizando persistência em um banco de dados relacional.

Para isso, são apresentados alguns padrões de desenvolvimento que buscam maximizar os benefícios da utilização do paradigma e facilitam a tarefa de persistência de objetos.

Este artigo está dividido em duas partes. Na primeira é apresentado um estudo de caso utilizado para exemplificar o desenvolvimento de software orientado a objetos utilizando os padrões de desenvolvimento, além da definição das classes, métodos e atributos do modelo. É apresentado ainda o conceito de mapeamento objeto-relacional e como utilizar esta abordagem no escopo do estudo de caso apresentado. Por fim, ainda na primeira parte é definida a estratégia de conexão com o banco de dados e das classes responsáveis pela persistência a manipulação dos objetos.

Na segunda parte será implementada a estratégia de persistência de objetos utilizada, além de exemplos e codificação de persistência e manipulação dos objetos.

Estudo de Caso

No escopo deste artigo é apresentado um estudo de caso para exemplificar o desenvolvimento de um software Orientado a Objetos utilizando padrões de desenvolvimento no ambiente Delphi 7. Com ele é possível verificar e exemplificar a utilização de padrões de projeto que auxiliam o desenvolvimento da aplicação, de técnicas de Programação Orientadas a Objetos e de persistência de objetos.

O estudo de caso, apresenta um fragmento de um sistema maior, representado aqui pela relação entre funcionários e departamentos de uma empresa, através das classes *Departamento*, *Funcionario*, *FuncionarioMensalista* e *FuncionarioDiarista*. A classe *Funcionario* é abstrata, servindo para a definição dos elementos comuns às suas subclasses através do mecanismo de herança.

Todos os funcionários devem estar lotados em um *Departamento*, que pode ter vários funcionários. A **Figura 1** representa o diagrama de classes do estudo de caso.

Através desse estudo de caso, são abordados temas como classes, objetos, métodos, atributos, persistência de objetos, herança, polimorfismo, associações e padrões de projeto, além de temas pertinentes ao ambiente de desenvolvimento, como utilização de componentes, classes e controles disponibilizados pelo Delphi.

Para a implementação do estudo de caso será utilizado o Delphi 7 como ferramenta de desenvolvimento e, para a persistência de objetos será utilizado o banco de dados relacional

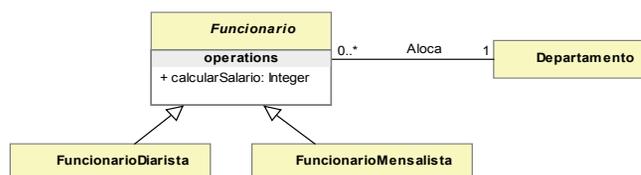


Figura 1. Diagrama de Classes do estudo de caso

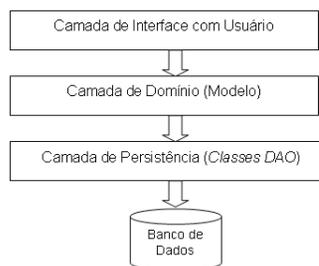


Figura 2. Disposição das camadas do estudo de caso

Firebird. É utilizada ainda a abordagem de programação em camadas, que busca, entre outras coisas, o desacoplamento de funções da aplicação, facilitando assim o desenvolvimento e garantindo maior manutenibilidade.

No escopo deste artigo é utilizada uma camada para apresentação, representada pela interface do sistema, uma camada de domínio, representada pelas classes de negócio da aplicação, uma camada de persistência, representada pelas classes DAO (*Data Access Object*) que são responsáveis pela persistência e manipulação dos objetos no banco de dados.

E, por fim, uma camada de banco de dados, representado pelo SGBD (Sistema Gerenciador de Banco de Dados) Firebird. A **Figura 2** exibe a disposição das camadas utilizadas no estudo de caso.

Classes, Atributos e Métodos

Com um projeto criado no Delphi 7, o primeiro passo é implementar a definição das classes do modelo apresentado. Para isso, é criada uma unit chamada *untDominio.pas*. Neste artigo, todas as classes do modelo são definidas e implementadas nessa unit.

A **Listagem 1** apresenta a definição da classe *Departamento* e sua implementação. Já na **Listagem 2** é apresentada a classe abstrata *Funcionario*. Por fim, a **Listagem 3**, exibe a implementação da classe *FuncionarioDiarista*. A definição e implementação da classe *FuncionarioMensalista* não é apresentada pois é muito similar à *FuncionarioDiarista*.

Na **Listagem 1**, nas linhas 4 e 5 são definidos os atributos privados da classe *Departamento*. O primeiro é o atributo *OID* (*Object Identifier*), que representa o identificador do objeto no modelo. No modelo relacional, esse atributo é considerado o campo chave-primária. O segundo atributo é a descrição do departamento.

Entre as linhas 7 e 14 são apresentados os métodos da classe. Os primeiros são os métodos *get* e *set*, responsáveis por manipular os valores dos atributos, mantendo assim o encapsulamento da classe.

Nota: Encapsulamento é a prática de limitar o acesso às informações de um objeto, fazendo com que so-

mente os seus métodos tenham acesso aos dados e à manipulação dos atributos. O encapsulamento disponibiliza o objeto com toda sua funcionalidade sem a necessidade de saber internamente o seu funcionamento, aumentando assim a manutenibilidade e reutilização do sistema.

Os demais métodos definidos são os responsáveis pela manipulação dos objetos da classe no banco de dados. Eles basicamente fazem acesso às classes de persistência, que são definidas posteriormente neste artigo, e manipulam os objetos no banco de dados.

Para isso são definidos os seguintes métodos: *Persistir*, salva e atualiza os objetos, *Obter*, recupera um objeto específico na base de dados, *ObterTodos*, recupera todos os objetos da classe

Listagem 1. Definição da classe Departamento

```

1.type
2. Departamento = class
3.private
4. OID: string;
5. fDescricao: string;
6.public
7. function getOID(): string;
8. procedure setOID(pOID: string);
9. function getDescricao: string;
10. procedure setDescricao(pDescricao: string);
11. function Persistir: Boolean;
12. class function Obter(pOID: string): Departamento;
13. class function ObterTodos: TList;
14. function Destruir: Boolean;
15.end;
16.
17. implementation
18.
19.function Departamento.getOID: string;
20.begin
21. result := OID;
22.end;
23.procedure Departamento.setOID(pOID: string);
24.begin
25. OID := pOID;
26.end;
27.function Departamento.getDescricao: string;
28.begin
29. result := fDescricao;
30.end;
31.procedure Departamento.setDescricao(pDescricao: string);
32.begin
33. fDescricao := pDescricao;
34.end;
35.function Departamento.Persistir: Boolean;
36.var
37. objPersistente: DepartamentoDAO;
38.begin
39. objPersistente:= DepartamentoDAO.Create;
40. result:=objPersistente.Persistir(self);
41. objPersistente.Free;
42.end;
43.class function Departamento.Obter(pOID: string): Departamento;
44.var
45. objPersistente: DepartamentoDAO;
46.begin
47. objPersistente := DepartamentoDAO.Create;
48. result := objPersistente.Obter(pOID);
49. objPersistente.Free;
50.end;
51.class function Departamento.ObterTodos: TList;
52.var
53. objPersistente: DepartamentoDAO;
54.begin
55. objPersistente := DepartamentoDAO.Create;
56. result := objPersistente.ObterTodos;
57. objPersistente.Free;
58.end;
59.function Departamento.Destruir: Boolean;
60.var
61. objPersistente: DepartamentoDAO;
62.begin
63. objPersistente := DepartamentoDAO.Create;
64. result:= objPersistente.Destruir(self.getOID);
65. objPersistente.Free;
66.end;

```

e, por fim, o *Destruir*, responsável pela exclusão de um determinado objeto.

Entre as linhas 35 e 42, é apresentado o *Persistir*. Ele inicialmente define o uso de uma variável do tipo *DepartamentoDAO*, que é a implementação responsável pela persistência e comunicação da aplicação com a base de dados, para a classe *Departamento*.

Após instanciado, na linha 39, a variável *objPersistente* invoca o seu *Persistir*, passando como parâmetro o *Departamento* em memória (*self*). Caso seja obtido sucesso na tarefa, a função então retorna *true*, caso contrário, *false*. Ao final, na linha 41, o objeto é destruído.

A linha 43 apresenta a assinatura do método de classe *Obter*, que recebe como parâmetro o OID do objeto a ser recuperado. Esse método retorna o objeto recuperado da base de dados. Nas linhas 51 a 58, é apresentado o método de classe *ObterTodos*.

Esse retorna um *TList* dos objetos *Departamento* da base de dados. Já entre as linhas 59 a 66, é definido o *Excluir*, que é responsável pela exclusão do objeto da base de dados, retornando *true* caso obtenha êxito, ou *false* em caso de falha.

Nota: Método de Classe é um método que não há a necessidade de se instanciar um objeto da classe para poder acessá-lo. Diferentemente do método de instância, que como o próprio nome diz, é necessário a instanciação do objeto para sua invocação.

Na classe *Departamento*, o *Obter* e *ObterTodos* são métodos de classe, já que para realizar essas tarefas não é necessário o objeto instanciado na memória. Para definir um método de classe no Delphi, basta utilizar o modificador *class* na definição da assinatura do método. Caso não seja utilizado esse modificador, o compilador trata-o como um método de instância.

Já na **Listagem 2**, entre as linhas 2 e 21 é definida a classe *Funcionario* com seus atributos e métodos. Entre as linhas 24 e 82 são implementados os métodos da classe. Nas linhas 4 a 6 são definidos os atributos privados da classe, sendo acessados pelos métodos *get* e *set* apresentados entre as linhas 34 e 57.

A classe possui os atributos *OID*, *fNome* e *fDepartamento*. Esse último é a implementação da associação apresentada no modelo, no qual um funcionário é lotado em um *Departamento*. Na classe ainda são definidos os métodos *Persistir*, *ObterTodos*, *ObterFuncionariosPorOIDDepartamento*, *Destruir* e *calcularSalario*.

O *Persistir* é virtual e abstrato, o que significa que ele não possui implementação na classe, devendo ser codificado nas classes filhas. Da mesma forma, o *calcularSalario* também é virtual e abstrato já que o mesmo é redefinido nas subclasses, de acordo com as necessidades específicas de implementação de cada especialização.

Isso é uma das características de um conceito chamado polimorfismo, que é a capacidade de métodos de mesma assinatura comportar-se de maneiras diferentes, em uma mesma hierarquia. Nas linhas 25 a 28 é implementado o método construtor da classe *Funcionario* e das linhas 29 à 33 o destrutor.

O objetivo é permitir a instanciação do *Departamento* no momento da instanciação de um objeto *Funcionario*, como apresentado na linha 27. Para o destrutor o objetivo é destruir o objeto *Departamento* associado juntamente com o objeto *Funcionario*. Os demais métodos definidos entre as linhas 58 a

Listagem 2. Definição da Classe Funcionario

```

1.type
2. Funcionario = class
3.private
4. OID: string;
5. fNome: string;
6. fDepartamento: Departamento;
7.public
8. constructor Create;
9. destructor Destroy;
10. function getOID: string;
11. procedure setOID(pOID: string);
12. function getNome: string;
13. procedure setNome(pNome: string);
14. function getDepartamento: Departamento;
15. procedure setDepartamento(pDepartamento: Departamento);
16. function Persistir: Boolean; virtual; abstract;
17. class function ObterTodos: TList; virtual;
18. class function ObterFuncionariosPorOIDDepartamento(
    pOIDDepartamento: string): TList;
19.class function Destruir(pOID: string): Boolean; virtual;
20.function calcularSalario: Real; virtual; abstract;
21.end;
22.
23.implementation
24.
25.constructor Funcionario.Create;
26.begin
27. fDepartamento := Departamento.Create;
28.end;
29.destructor Funcionario.Destroy;
30.begin
31. fDepartamento.Free;
32. inherited;
33.end;
34.function Funcionario.getOID: string;
35.begin
36. result := OID;
37.end;
38.procedure Departamento.setOID(pOID: string);
39.begin
40. OID := pOID;
41.end;
42.function Funcionario.getNome: string;
43.begin
44. result := fNome;
45.end;
46.procedure Funcionario.setNome(pNome: string);
47.begin
48. fNome := pNome;
49.end;
50.function Funcionario.getDepartamento: Departamento;
51.begin
52. result := fDepartamento;
53.end;
54.procedure Funcionario.setDepartamento(pDepartamento: Departamento);
55.begin
56. fDepartamento := pDepartamento;
57.end;
58.class function Funcionario.ObterTodos: TList;
59.var
60. objPersistente: FuncionarioDAO;
61.begin
62. objPersistente := FuncionarioDAO.Create;
63. result := objPersistente.ObterTodos;
64. objPersistente.Free;
65.end;
66.class function Funcionario.ObterFuncionariosPorOIDDepartamento(
    pOIDDepartamento: string): TList;
67.var
68. objPersistente: FuncionarioDAO;
69. objPersistente := FuncionarioDAO.Create;
70.begin
71. objPersistente := FuncionarioDAO.Create;
72. result := objPersistente.ObterFuncionariosPorOIDDepto(pOIDDepartamento);
73. objPersistente.Free;
74.end;
75.class function Funcionario.Destruir(pOID: string): Boolean;
76.var
77. objPersistente: FuncionarioDAO;
78.begin
79. objPersistente := FuncionarioDAO.Create;
80. result := objPersistente.Destruir(pOID);
81. objPersistente.Free;
82.end;

```

82 são os de manipulação e persistência da classe.

Esses, assim como os definidos na classe *Departamento* apresentada, fazem chamada à classe *FuncionarioDAO* de acesso aos dados do funcionário.

Por fim, na **Listagem 3**, a classe *FuncionarioDiarista* é apresentada. Logo na linha 2, a classe é definida especificando a sua superclasse *Funcionario*. Além de herdar todos os atributos da superclasse, o funcionário diarista apresenta o atributo *fNumDias*, representando o número de dias trabalhados em um mês e o atributo *fValorDia*, representando o valor recebido por dia trabalhado.

Listagem 3. Definição da Classe FuncionarioDiarista

```

1.type
2. FuncionarioDiarista = class (Funcionario)
3.private
4. fNumDias: Integer;
5. fValorDia: Currency;
6.public
7.function getNumDias: Integer;
8.procedure setNumDias(pNumDias: Integer);
9.function getValorDia: Currency;
10.procedure setValorDia(pValorDia: Currency);
11.function Persistir: Boolean; override;
12.class function Obter(pOID: string): FuncionarioDiarista;
13.class function ObterTodos: TList; override;
14.function Destruir: Boolean; reintroduce; overload;
15.function calcularSalario: Real; override;
16.end;
17.
18.implementation
19.
20.function FuncionarioDiarista.getNumDias: Integer;
21.begin
22. result := fNumDias;
23.end;
24.procedure FuncionarioDiarista.setNumDias(pNumDias: Integer);
25.begin
26. fNumDias := pNumDias;
27.end;
28.function FuncionarioDiarista.getValorDia: Currency;
29.begin
30. result := fValorDia;
31.end;
32.procedure FuncionarioDiarista.setValorDia(pValorDia: Currency);
33.begin
34. fValorDia := pValorDia;
35.end;
36.function FuncionarioDiarista.Persistir: Boolean;
37.var
38. objPersistente: FuncionarioDiaristaDAO;
39.begin
40. objPersistente := FuncionarioDiaristaDAO.Create;
41. result := objPersistente.Persistir(self);
42. objPersistente.Free;
43.end;
44.class function FuncionarioDiarista.Obter(
    pOID: string): FuncionarioDiarista;
45.var
46. objPersistente: FuncionarioDiaristaDAO;
47.begin
48. objPersistente := FuncionarioDiaristaDAO.Create;
49. result := objPersistente.Obter(pOID);
50. objPersistente.Free;
51.end;
52.class function FuncionarioDiarista.ObterTodos: TList;
53.var
54. objPersistente: FuncionarioDiaristaDAO;
55.begin
56. objPersistente := FuncionarioDiaristaDAO.Create;
57. result := objPersistente.ObterTodos;
58. objPersistente.Free;
59.end;
60.function FuncionarioDiarista.Destruir: Boolean;
61.var
62. objPersistente: FuncionarioDiaristaDAO;
63.begin
64. objPersistente := FuncionarioDiaristaDAO.Create;
65. result := objPersistente.Destruir(self.getOID);
66. objPersistente.Free;
67.end;
68.function FuncionarioDiarista.calcularSalario: Real;
69.begin
70. result := fNumDias * fValorDia;
71.end;

```

A classe apresenta a definição e implementação do *Persistir* (linhas 36 a 43), abstrato na superclasse, além dos métodos *Obter*, *ObterTodos*, *Destruir* e *calcularSalario*. O *Persistir* é responsável por persistir e/ou atualizar os objetos na base de dados. O *Obter* tem a funcionalidade de recuperar um objeto na base de dados.

O *ObterTodos* tem a função de recuperar todos os objetos do tipo *FuncionarioDiarista* e o *Destruir* de excluir o objeto referenciado da base de dados. Por fim, o *calcularSalario*, implementa a operação de cálculo de salário para um funcionário diarista, como apresentado a partir da linha 68.

Mapeamento Objeto-Relacional e banco de dados

O mapeamento objeto-relacional é uma abordagem que permite a construção de sistemas utilizando o paradigma Orientado a Objetos com a persistência desses objetos em bancos de dados relacionais. Utilizando-se de técnicas e estratégias específicas, é possível mapear classes com seus atributos e associações para o modelo relacional.

Utiliza-se uma abstração bastante intuitiva no sentido de que uma classe pode ser mapeada para uma tabela no banco de dados relacional e atributos da classe para campos da tabela. Porém, algumas diferenças entre os dois modelos, como o OID, tipos de dados, herança e associações, demandam um estudo mais detalhado das estratégias de mapeamento.

Para o estudo de caso, a construção da base de dados seguiu esse raciocínio. A classe *Departamento* foi mapeada para a tabela *departamento*, com seus atributos sendo os campos da tabela. A hierarquia *Funcionario* foi mapeada para uma única tabela *funcionario*, com todos os atributos (como campos da tabela), acrescido de um campo chamado *tipo*, que representa a classe que instanciou cada objeto.

Existem outras duas possíveis alternativas: criar uma nova tabela para cada classe da hierarquia e relacioná-las utilizando o conceito de chave estrangeira ou criar uma nova tabela para cada classe concreta, contendo os seus atributos, mais os atributos da classe abstrata. Neste artigo isso não foi feito em função da simplicidade do modelo.

Assim, a **Listagem 4** apresenta o *script* para criação do banco, resultante do mapeamento objeto-relacional realizado com as classes do estudo de caso.

Listagem 4. Script para a criação das tabelas no banco de dados

```
CREATE TABLE DEPARTAMENTO (
  OID          CHAR(38) NOT NULL,
  DESCRICAO   VARCHAR(50));

CREATE TABLE FUNCIONARIO (
  OID          CHAR(38) NOT NULL,
  NOME         VARCHAR(50),
  DEPARTAMENTO CHAR(38),
  TIPO         CHAR(1) NOT NULL,
  VALORMES    DECIMAL(13,2),
  VALORDIA    DECIMAL(13,2),
  NUMDIAS     INTEGER);

ALTER TABLE DEPARTAMENTO ADD CONSTRAINT
  PK_DEPARTAMENTO PRIMARY KEY (OID);
ALTER TABLE FUNCIONARIO ADD CONSTRAINT
  PK_FUNCIONARIO PRIMARY KEY (OID);
ALTER TABLE FUNCIONARIO ADD CONSTRAINT
  FK_FUNCIONARIO FOREIGN KEY (DEPARTAMENTO)
  REFERENCES DEPARTAMENTO (OID);
```

Com a base de dados criada no Firebird, é possível configurar a forma de acesso da aplicação ao banco. O acesso aos dados é realizado por meio de um Data Module. Nesse estão presentes os componentes que permitem a realização da conexão com banco de dados, recuperação e atualização de dados por meio de expressões em SQL.

A **Figura 3** mostra os componentes que fazem parte do Data Module denominado *dtmdlDados*.

O *sqlcnctnDados* é do tipo *TSQLConnection*, que permite através de drivers específicos estabelecer conexão com bancos de dados. Neste exemplo é utilizado o driver do InterBase, que também é compatível com a versão do Firebird que foi utilizada nessa aplicação.

Os parâmetros de conexão utilizados na aplicação são apresentados na **Figura 4**.

Os demais componentes: *sqldstPrincipal*, *dtstprvdrPrincipal* e *clntdtstPrincipal* são do tipo *TSQLDataset*, *TDataSetProvider* e *TClientDataSet*. São ligados ao *sqlcnctnDados* bem como interligados entre si por meio das propriedades *SQLConnection*, *DataSet* e *ProviderName*, respectivamente.

Tais componentes são responsáveis por manipular conjuntos de registros recuperados e destinados ao banco de dados.

Classes de Persistência

As classes de persistência e manipulação de objetos são tratadas em uma unit separada, no exemplo chamada de *untPersistencia*. Dessa forma, isolam-se em uma camada todas as questões relativas à persistência e, por conseqüência, questões relativas ao acesso a banco de dados. Isso facilita manutenções futuras na aplicação, além de obter outras vantagens do desenvolvimento em camadas.

Essas classes representam um padrão de projeto chamado *Data Access Object*, que, basicamente, define o encapsulamento do acesso aos dados por meio de classes especificamente construídas para essa tarefa. Assim, para cada classe persistente do modelo existe uma classe DAO correspondente.

Sendo assim, as classes DAO são responsáveis por utilizar o Data Module definido anteriormente, para realizar a manipulação dos dados. O Data Module não é referenciado diretamente, mas sim instanciados objetos a partir desse.

A **Figura 5** exhibe a estrutura de classes da *untPersistente*, onde são implementadas as classes *TPersistente* e as classes DAO correspondentes às classes de domínio.

A hierarquia de classes apresentada na **Figura 5** demonstra uma forma possível de implementar classes DAO que tratam da persistência de objetos. Uma outra forma possível, não tratada neste artigo, seria o uso de interface no lugar da classe *TPersistente*, o que evitaria a construção hierarquizada das classes DAO.

De uma forma ou de outra, as classes DAO implementam somente métodos responsáveis pela recuperação e manutenção dos dados armazenados em banco de dados, de suas correspondentes classes de domínio. A classe *TPersistente*, que é ancestral das demais classes, possui implementação em apenas um método (*GerarOID*).

Os demais são abstratos, caracterizando essa classe como uma

classe abstrata, exigindo a implementação em suas classes descendentes. Dessa forma garante-se que as classes DAO possuam implementações básicas de persistência e manipulação de objetos, através dos métodos *Persistir*, *Obter*, *ObterTodos* e *Destruir*.

A **Listagem 5** apresenta a definição da classe *TPersistente*.

A classe apresenta ainda os métodos *Persistir*, *Obter*, *ObterTodos* e *Destruir*, que além de serem abstratos, sofrem sobrecarga nas classes descendentes. Já o *GerarOID*, possui implementação, visto que o OID (identificador do objeto) possui uma regra única para sua formação.

É utilizado o algoritmo GUID, disponível no Delphi para formação do OID. Isso garante a unicidade do OID gerado. Outros algoritmos podem ser utilizados aqui caso seja necessário, bem como algoritmos diferentes podem ser implementados para cada classe DAO. O GUID foi escolhido por ser de fácil implementação.

O identificador gerado, embora seja um conjunto de caracteres do sistema hexadecimal como, por exemplo `{81B35AC9-F6C5-411E-9176-3D0A4D2C8E04}`, não será problema visto que o usuário não necessita ter contato direto com ele. O OID é utilizado para identificar um objeto bem como atuar como chave primária em tabelas do banco de dados relacional.

A **Listagem 6** apresenta o código correspondente à implementação do *GerarOID*.

A **Listagem 7** mostra a definição das demais classes DAO.

Na definição das classes DAO, nota-se principalmente a presença dos métodos *Persistir*, *Obter*, *ObterTodos* e *Destruir*. Tais métodos redefinem a implementação dos métodos abstratos da classe *TPersistente* e são os responsáveis pela manipulação dos objetos na base de dados relacional. Percebe-se que *Persistir* e *Obter* possuem assinaturas diferentes da classe ancestral, por isso utilizam a diretiva *overload*.

A classe *FuncionarioDAO* é ancestral das classes *FuncionarioDiaristaDAO*, espelhando o domínio. Para essa hierarquia de classes de funcionário foi definida uma única tabela no banco de dados para armazenamento das informações, tanto de funcionários diaristas quanto funcionários mensalistas. Outras formas também seriam possíveis como uma tabela para cada classe da hierarquia ou então uma tabela para cada classe concreta.

Da forma que estão definidas as classes da hierarquia de *FuncionarioDAO*, nem todos métodos são implementados em todas as classes. Por exemplo, a classe *FuncionarioDAO* não possui implementação de *Persistir* e *Obter*, visto que cada caso (mensalista e diarista) são tratados separadamente.

Já o *Destruir* possui implementação na classe *FuncionarioDAO* e não possui nas classes descendentes, pois a exclusão de um funcionário independe

do seu tipo. Ainda, o *ObterTodos* está implementado em todas as classes da hierarquia, tendo semântica diferente. Na classe *FuncionarioDAO* o método recupera todos os objetos de funcionários independente do tipo, e nas classes descendentes recupera todos os objetos do respectivo tipo.

Além dos métodos padrões definidos na classe *TPersistente*, a classe *FuncionarioDAO* implementa o *ObterFuncionariosPorOIDDepto*, que permite a recuperação de objetos de funcionários a partir do identificador de um departamento, que está associado.

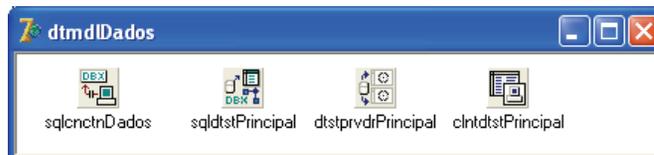


Figura 3. Data Module de conexão ao banco de dados

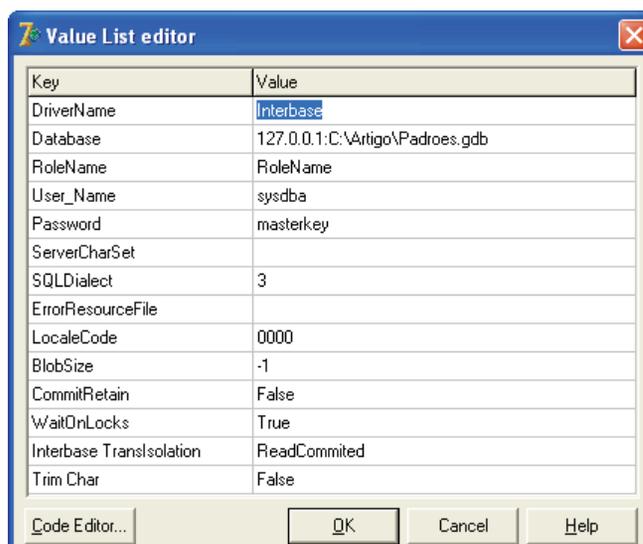


Figura 4. Configuração do sqlcnctnDados

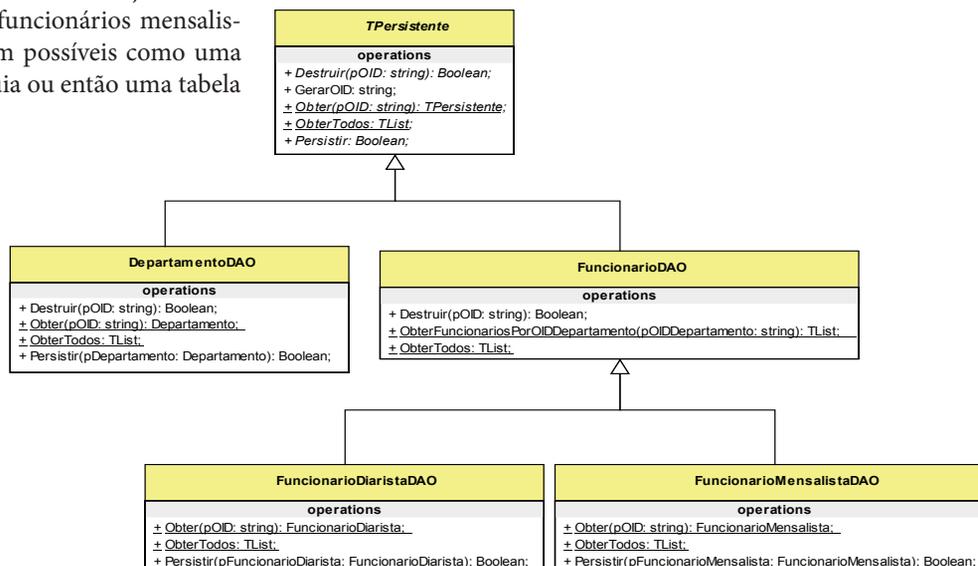


Figura 5. Estrutura de classes responsáveis pela persistência dos objetos

Listagem 5. Definição da Classe TPersistente

```
TPersistente = class (TObject)
public
    function GerarOID: string;
    function Persistir: Boolean; virtual; abstract;
```

Listagem 6. Código-fonte do Método GerarOID

```
function TPersistente.GerarOID: string;
var
    vGUID: TGUID;
begin
    CreateGUID(vGUID);
    result := GUIDToString(vGUID);
end;
class function Obter(pOID: string): TPersistente;
virtual; abstract;
class function ObterTodos: TList; virtual;
abstract;
function Destruir(pOID: string): Boolean; virtual;
abstract;
end;
```

Listagem 7. Definição das classes DAO

```
DepartamentoDAO = class (TPersistente)
public
    function Persistir(pDepto: Departamento): Boolean;
    reintroduce; overload;
    class function Obter(pOID: string): Departamento;
    reintroduce; overload;
    class function ObterTodos: TList; override;
    function Destruir(pOID: string): Boolean; override;
end;
FuncionarioDAO = class (TPersistente)
public
    class function ObterTodos: TList; override;
    class function ObterFuncionariosPorOIDDepto(
        pOIDDepartamento: string): TList;
    function Destruir(pOID: string): Boolean; override;
end;
FuncionarioDiaristaDAO = class (FuncionarioDAO)
public
    function Persistir(
        pFuncDia: FuncionarioDiarista): Boolean;
    reintroduce; overload;
    class function Obter(
        pOID: string): FuncionarioDiarista;
    reintroduce; overload;
    class function ObterTodos: TList; override;
end;
```

Conclusão

Esta primeira parte do artigo teve como objetivo abordar de forma prática a utilização do paradigma Orientado a Objetos no contexto do desenvolvimento de software no ambiente Delphi e de técnicas aliadas, como mapeamento objeto relacional e utilização de padrões de desenvolvimento.

Foi possível observar na prática a criação de classes e a definição de atributos e métodos. Foi abordado também o tema de mapeamento objeto-relacional, que é de suma importância quando da persistência de objetos utilizando um banco de dados relacional. Ao final, foi definida a estratégia utilizada para conexão com o banco de dados e foi apresentada a definição das classes responsáveis pela manipulação e persistência dos objetos (Classes DAO). Foi possível também definir o padrão de projeto DAO (*Data Access Object*), que busca estreitar os benefícios da orientação a objetos com a maturidade do banco de dados relacional.

Na próxima edição, será apresentada a implementação das classes de persistência com o objetivo de persistir e manipular os objetos no banco de dados relacional, além de encapsular toda a comunicação com o banco

de dados. Serão apresentados alguns exemplos práticos no contexto do estudo de caso aqui apresentado e definido. ■

ClubeDelphi PLUS

Acesse agora mesmo o Portal do Assinante ClubeDelphi e assista a uma vídeo-aula de Luciano Pimenta que mostra como usar um driver free e específico do Firebird para a tecnologia dbExpress.

www.devmedia.com.br/articles/viewcomp.asp?comp=1941

ClubeDelphi PLUS

Acesse agora mesmo o portal do assinante ClubeDelphi e assista a uma vídeo-aula de Guinther Pauli que mostra como utilizar GUID para gerar chaves primárias para o banco de dados através do Delphi.

www.devmedia.com.br/articles/viewcomp.asp?comp=3321

PENSE...
 QUANTO TEMPO
 VOCÊ GASTARIA
 PARA DESENVOLVER
 COBRANÇA COM BOLETOS
 BANCÁRIOS PARA
 APENAS UM BANCO
 NO SEU SOFTWARE



COBREBEMX

-  56 BANCOS E MAIS DE 430 CARTEIRAS DE COBRANÇA PARA IMPRESSÃO E/OU ENVIO DE BOLETO BANCÁRIO POR EMAIL;
-  GERAÇÃO DE BOLETOS ON LINE;
-  GERAÇÃO E LEITURA DE ARQUIVOS (REMESSA/RETORNO) NOS PADRÕES FEBRABAN E CNAB;
-  MAIS DE 40 EXEMPLOS EM DIVERSAS LINGUAGENS DE PROGRAMAÇÃO



DOWNLOADS E INFORMAÇÕES EM WWW.COBREBEM.COM